4th Dimension®

Language Reference
Windows and Mac OS Versions

4th Dimension
by
Laurent Ribardière
Adapted by Bernard Gallet

4th Dimension Language Reference
Version 6.0 for Windows® and Mac™ OS

IMPORTANT LICENSE INFORMATION

Contents

1 Introduction

Introduction

version 6.0

4th Dimension has its own programming language. This built-in language, consisting of over 500 commands, makes 4th Dimension a powerful development tool for database applications on desktop computers. You can use the 4th Dimension language for many different tasks—from performing simple calculations to creating complex custom user interfaces. For example, you can:

• Programmatically access any of the editors available to the user in the User environment,
• Create and print complex reports and labels with the information from the database,
• Communicate with other devices,
• Manage documents,
• Import and export data between 4th Dimension databases and other applications,
• Incorporate procedures written in other languages into the 4th Dimension programming language.

The flexibility and power of the 4th Dimension programming language make it the ideal tool for all levels of users and developers to accomplish a complete range of information management tasks. Novice users can quickly perform calculations. Experienced users without programming experience can customize their databases. Experienced developers can use this powerful programming language to add sophisticated features and capabilities to their databases, including file transfer and communications. Developers with programming experience in other languages can add their own commands to the 4th Dimension language.

The 4th Dimension programming language is expanded when any of the 4th Dimension modules are added to the application. Each module includes language commands that are specific to the capabilities they provide.

About the Manuals

The manuals described here provide a guide to the features of both 4th Dimension and 4D Server. The only exception is the *4D Server Reference*, which describes features exclusive to 4D Server and is included only in the 4D Server documentation package.

• The *Language Reference* is a guide to using the 4th Dimension language. Use this manual to learn how to customize your database by incorporating 4th Dimension commands and functions.
• The *Design Reference* provides detailed descriptions of the operations you can perform in the Design environment to create forms for managing data.
• The *User Reference* provides a description of the User environment, in which users enter and manipulate data in forms.
• The *Discover 4D* manual leads you through example lessons in which you create and use a 4th Dimension database. These examples provide hands-on experience and help you become familiar with the concepts and features of 4th Dimension and 4D Server.
• The *4D Server Reference,* which is included only in the 4D Server package, is a guide to managing multi-user databases with 4D Server.

About this Manual

This manual describes the 4th Dimension language. It assumes that you are familiar with terms such as table, field, and form. Before you read this manual, you should:

• Use the *Discover 4D* manual to work through the database example.
• Begin creating your own databases, referring to the *Design Reference* manual when necessary.
• Be comfortable with managing your database in the User environment. See the *User Reference* manual for more information on the User environment.

Where to go from here?

If you read this manual for the first time, read the section Introduction.

Introduction

This topic introduces you to the 4th Dimension programming language. The following topics are discussed:

• What the language is and what it can do for you,
• How you will use methods,
• How to develop an application with 4th Dimension.

These topics are covered here in general terms; they are covered in greater detail in other sections.

What is a Language?

The 4th Dimension language is not very different from the spoken language we use every day. It is a form of communication used to express ideas, inform, and instruct. Like a spoken language, 4th Dimension has its own vocabulary, grammar, and syntax; you use it to tell 4th Dimension how to manage your database and data.

You do not need to know everything in the language in order to work effectively with 4th Dimension. In order to speak, you do not need to know the entire English language; in fact, you can have a small vocabulary and still be quite eloquent. The 4th Dimension language is much the same—you only need to know a small part of the language to become productive, and you can learn the rest as the need arises.

Why Use a Language?

At first it may seem that there is little need for a programming language in 4th Dimension. The Design and User environments provide flexible tools, which require no programming to perform a wide variety of data management tasks. Fundamental tasks, such as data entry, queries, sorting, and reporting are handled with ease. In fact, many extra capabilities are available, such as data validation, data entry aids, graphing, and label generation.

Then why do we need a 4th Dimension language? Here are some of its uses:

• Automate repetitive tasks: These tasks include data modification, generation of complex reports, and unattended completion of long series of operations.
• Control the user interface: You can manage windows and menus, and control forms and interface objects.
• Perform sophisticated data management: These tasks include transaction processing, complex data validation, multi-user management, sets, and named selection operations.
• Control the computer: You can control serial port communications, document management, and error management.
• Create applications: You can create easy-to-use, customized databases that use the Runtime environment.
• Add functionality to the built-in 4D Web Services: Create dynamic HTML pages in addition to those automatically translated from forms by 4D.

The language lets you take complete control over the design and operation of your database. While the User environment gives you powerful "generic" tools, the language lets you customize your database to whatever degree you require.

Taking Control of Your Data

The 4th Dimension language lets you take complete control of your data in a powerful and elegant manner. The language is easy enough for a beginner, and sophisticated enough for an experienced application developer. It provides smooth transitions from built-in database functions to a completely customized database.

The commands in the 4th Dimension language provide access to the User environment editors, with which you are already familiar. For example, when you use the QUERY command, you are presented with the Query Editor. Using this language command is almost as easy as choosing the Query command from the Queries menu, but the QUERY command is even more useful. You can tell the QUERY command to search for explicitly described data. For example, QUERY ([People];[People]Last Name="Smith") will find all the people named Smith in your database.

The 4th Dimension language is very powerful—one command often replaces hundreds or even thousands of lines of code written in traditional computer languages. Surprisingly enough, with this power comes simplicity—commands have plain English names. For example, to perform a query, you use the QUERY command; to add a new record, you use the ADD RECORD command.

The language is designed for you to easily accomplish almost any task. Adding a record, sorting records, searching for data, and similar operations are specified with simple and direct commands. But the language can also control the serial ports, read disk documents, control sophisticated transaction processing, and much more.

The 4th Dimension language accomplishes even the most sophisticated tasks with relative simplicity. Performing these tasks without using the language would be unimaginable for many.

Even with the language’s powerful commands, some tasks can be complex and difficult. A tool by itself does not make a task possible; the task itself may be challenging and the tool can only ease the process. For example, a word processor makes writing a book faster and easier, but it will not write the book for you. Using the 4th Dimension language will make the process of managing your data easier and will allow you to approach complicated tasks with confidence.

Is it a “Traditional” Computer Language?

If you are familiar with traditional computer languages, this section may be of interest. If not, you may want to skip it.

The 4th Dimension language is not a traditional computer language. It is one of the most innovative and flexible languages available on a computer today. It is designed to work the way you do, and not the other way around.

To use traditional languages, you must do extensive planning. In fact, planning is one of the major steps in development. 4th Dimension allows you to start using the language at any time and in any part of your database. You may start by adding a method to a form, then later add a few more methods. As your database becomes more sophisticated, you might add a project method controlled by a menu. You can use as little or as much of the language as you want. It is not “all or nothing,” as is the case with many other databases.

Traditional languages force you to define and pre-declare objects in formal syntactic terms. In 4th Dimension, you simply create an object, such as a button, and use it. 4th Dimension automatically manages the object for you. For example, to use a button, you draw it on a form and name it. When the user clicks the button, the language automatically notifies your methods.

Traditional languages are often rigid and inflexible, requiring commands to be entered in a very formal and restrictive style. The 4th Dimension language breaks with tradition, and the benefits are yours.

Methods are the Gateway to the Language

A method is a series of instructions that causes 4th Dimension to perform a task. Each line of instruction in a method is called a statement. Each statement is composed of parts of the language.

Because you have already worked through the *Discover 4D* tutorials (you did go through *Discover 4D,* didn’t you?), you have already written and used methods.

You can create five types of methods with 4th Dimension:

• **Object Methods**: Usually short methods used to control form objects.
• **Form Methods**: Manage the display or printing of a form.
• **Table Methods/Triggers**: Used to enforce the rules of your database.
• **Project methods**: Methods that are available for use throughout your database. For example, methods that can be attached to menus.
• **Database methods**: Execute initializations or special actions when a database is opened or closed, or when a Web browser connects to your database published as a Web Server on Internet an Intranet.

The following sections introduce each of these method types and give you a feel for how you can use them to automate your database.

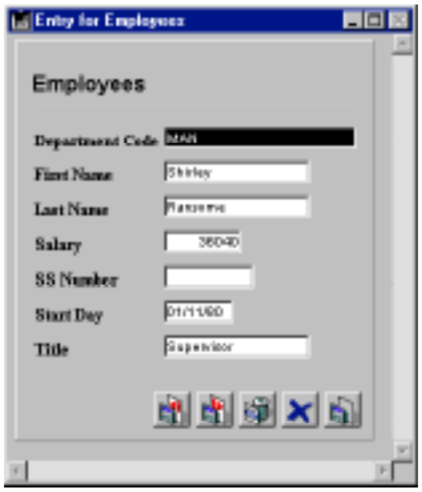
Getting started with object methods

Any form object that can perform an action (that is, any active object) can have a method associated with it. An object method monitors and manages the active object during data entry and printing. A object method is bound to its active object even when the object is copied and pasted. This allows you to create reusable libraries of scripted objects. The object method takes control exactly when needed.

Object methods are the primary tools for managing the user interface, which is the doorway to your database. The user interface consists of the procedures and conventions by which a computer communicates with the user. The goal is to make the user interface of your database as simple and easy to use as possible. The user interface should make interaction with the computer a pleasant process, one that the user enjoys or does not even notice.

There are two basic types of active objects in a form:
• Those for entering, displaying, and storing data; such as fields and subfields
• Those for control; such as enterable areas, buttons, scrollable areas, hierarchical lists, and meters

4th Dimension enables you to build classic forms, such as the one shown here:

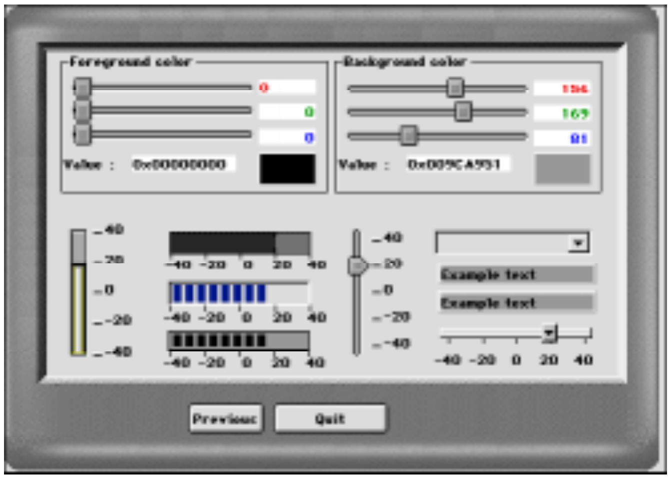


Entry for Employees

Employees

Department Code	MAN
First Name	Shirley
Last Name	Ransome
Salary	38040
SS Number	
Start Day	01/11/90
Title	Supervisor

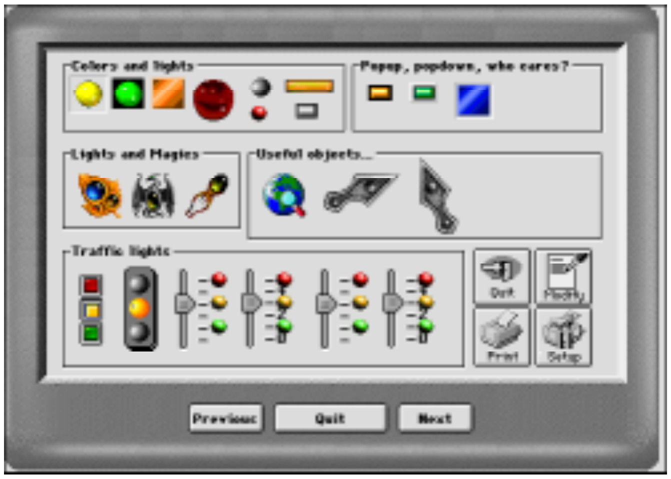
You can also build forms with multiple graphic controls, such as this one:



Foreground color: 0, 0, 0 — Value : 0x00000000

Background color: 156, 169, 81 — Value : 0x009CA951

You can even build forms that incorporate a graphical flair limited only by your imagination:

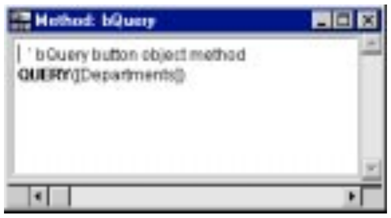


Whatever your style in building forms, all active objects have built-in aids, like range checking and entry filters for data entry areas, and automatic actions for controls, menus, and buttons. Always use these aids before adding object methods. The built-in aids are similar to methods in that they remain associated with the active object and are active only when the active object is being used. You will typically use a combination of built-in aids and object methods to control the user interface.

An object method associated with an active object used for data entry typically performs a data-management task specific to the field or variable. The method can perform data validation, data formatting, or calculations. It may even get related information from other files. Some of these tasks can, of course, also be performed with the built-in data entry aids for objects. Use object methods when the task is too complex for the built-in data entry aids to manage. For more information about the built-in data entry aids, refer to the *4th Dimension Design Reference*.

Object methods are also associated with active objects used for control, such as buttons. Active objects used for control are essential to navigating within your database. Buttons allow you to move from record to record, move to different forms, and add and delete data. These active objects simplify the use of a database and reduce the time required to learn it. Buttons also have built-in aids and, as with data entry, you should use these built-in aids before adding methods. Object methods enable you to add actions that are not built-in, to your controls.

For example, the following window is the object method for a button that, when clicked, displays the Query editor.



```
` bQuery button object method
QUERY([Departments])
```

As you become more proficient with scripts, you will find that you can create libraries of objects with associated methods. You can copy and paste these objects and their methods between forms, tables, and databases. You can even keep them in the Clipbook (Windows) or Scrapbook (Macintosh), ready to be used when you need them.

Controlling forms with form methods

In the same way that object methods are associated with the active objects in a form, a form method is associated with a form. Each form can have one form method. A form is the means through which you can enter, view, and print your data. Forms allow you to present the data to the user in different ways. Through the use of forms, you can create attractive and easy-to-use data entry screens and printed reports. A form method monitors and manages the use of an individual form both for data entry and for printing.

Form methods manage forms at a higher level than do object methods. Object methods are activated only when the object is used, whereas a form method is activated when anything in the form is used. Form methods are typically used to control the interaction between the different objects and the form as a whole.

As forms are used in so many different ways, it is informative to monitor what is happening while your form is in use. You use the various **form events** for this purpose. They tell you what is currently happening with the form. Each type of event (i.e., clicks, double-clicks, keystrokes...) enables or disables the execution of the form method as well as the object method of each object of the form.

For more information about form, objects, events and methods, see the section Form event.

Enforcing the rules of your database using the table methods/triggers

A Trigger is attached to a table; for this reason, it is also called a Table Method. Triggers are automatically invoked by the 4D database engine each you manipulate the records of a table (Add, Delete, Modify and Load). Triggers are methods that can prevent "illegal" operations with the records of your database. For example, in an invoicing system, you can prevent anyone from adding an invoice without specifying the customer to whom the invoice is billed. Triggers are a very powerful tool to restrict operations on a table as well as to prevent accidental data loss or tampering. You can write very simple triggers, then make them more and more sophisticated.

For detailed information about Triggers, see the section Triggers.

Using project method throughout the database

Unlike object methods, form methods, and triggers, which are all associated with a particular object, form, or table, project methods are available for use throughout your database. Project methods are reusable, and available for use by any other method. If you need to repeat a task, you do not have to write identical methods for each case. You can call project methods wherever you need them—from other project methods or from object or form methods. When you call a project method, it acts as if you had written the method at the location where you called it. Project methods called from other method are often referred to as "subroutines."

There is one other way to use project methods—associating them with menu commands. When you associate a project method with a menu command, the method is executed when the menu is chosen. You can think of the menu command as calling the project method.

Handling working sessions with database methods

In the same way object and form methods are invoked when events occur in a form, there are methods associated with the database which are invoked when a working session event occurs. These are the **database methods**. For example, each time you open a database, you may want to initialize some variables that will be used during the whole working session. To do so, you use the On Startup Database Method, automatically executed by 4D when you open the database.

For more information about Database Methods, see the section Database Methods.

Developing Your Database

Development is the process of customizing a database using the language and other built-in tools.

By simply creating a database, you have already taken the first steps to using the language. All the parts of your database—the tables and fields, the forms and their objects, and the menus—are tied to the language. The 4th Dimension language “knows” about all of these parts of your database.

Perhaps your first use of the language is to add a method to a form object in order to control data entry. Later, you might add a form method to control the display of your form. As the database becomes more complex, you can add a menu bar with project methods to completely customize your database.

As with other aspects of 4th Dimension, development is a very flexible process. There is no formal path to take during development—you can develop in a manner with which you are comfortable. There are, of course, some general patterns in the process.

• Implementation: Implement your design in the Design environment.
• Testing: You try out the design in the User environment and perhaps stay there to use your customized database.
• Usage: When your database is fully customized, you use it in the Custom Menus environment.
• Corrections: If you find errors, you return to the Design environment to fix them.

Special development support tools, hidden until needed, are built into 4th Dimension. As you use the language more frequently, you will find that these tools facilitate the development process. For example, the Method Editor catches typing errors and formats your work; the Interpreter (the engine that runs the language) catches errors in syntax and shows you where and what they are; and the Debugger lets you monitor the execution of your methods to catch errors in design.

Building Applications

By now you are familiar with the general uses of a database—data entry, searching, sorting, and reporting. You have performed these tasks in the User environment, using the built-in menus and editors.

As you use a database, you perform some sequences of tasks repeatedly. For example, in a database of personal contacts, you might search for your business associates, order them by last name, and print a specific report each time information about them is changed. These tasks may not seem difficult, but they can certainly be time-consuming after you have done them 20 times. In addition, if you don't use the database for a couple of weeks, you may return to find that the steps used to generate the report are not so fresh in your mind. The steps in methods are chained together, so a single command automatically performs all the tasks linked to it. Consequently, you do not have to worry about the specific steps.

Applications have custom menus and perform tasks that are specific to the needs of the person using your database. An application is composed of all the pieces of your database: the structure, the forms, the object, form and project methods, the menus, and the passwords.

You can use 4D Compiler to compile your databases and create stand-alone Windows and Macintosh applications. Compiling databases increases the execution speed of the language, protects your databases, and allows you to create applications that are completely independent. It also checks the syntax and the types of variables in methods for consistency.

An application can be as simple as a single menu that lets you enter people's names and print a report, or as complex as an invoicing, inventory, and control system. There are no limits to the uses of database applications. Typically, an application grows from a database used in the User environment to a database controlled completely by custom menus.

WHERE TO GO FROM HERE?
• Developing applications can be as simple or complex as you like. For a quick overview about building a simple 4D application, see the section Building a 4D application.
• If you are new to 4D, refer to the **Language Definition** sections to learn about the basics of the 4D language: start with Introduction to the 4D Language.

Building a 4D Application

An application is a database designed to fill a specific need. It has a user interface designed specifically to facilitate its use. The tasks that an application performs are limited to those appropriate for its purpose. Creating applications with 4th Dimension is smoother and easier than with traditional programming. 4th Dimension can be used to create a variety of applications, including:

- An invoice system
- An inventory control system
- An accounting system
- A payroll system
- A personnel system
- A customer tracking system
- A database shared over the Internet or an Intranet

It is possible that a single application could even contain all of these systems. Applications like these are typical uses of databases. In addition, the tools in 4th Dimension allow you to create innovative applications, such as:

- A document tracking system
- A graphic image management system
- A catalog publishing application
- A serial device control and monitoring system
- An electronic mail system (E-mail)
- A multi-user scheduling system
- A list such as a menu list, video collection, or music collection

An application typically starts as a database used in the User environment. The database "evolves" into an application as it is customized. What differentiates an application is that the systems required to manage the database are hidden from the user. Database management is automated, and users use menus to perform specific tasks.

When you use a 4th Dimension database in the User environment, you must know the steps to take to achieve a result. In an application, you use the Custom Menus environment, in which you need to manage all the aspects that are automatic in the User Environment.

These include:

•Table Navigation: The **Choose Table/Form** dialog box and **List of Tables** window are not available to the user. You can use menu commands and methods to control navigation between tables.

• Menus: In the Custom Menus environment, you only have the default File menu with the Quit menu command, Edit menu, and the Help menu (Windows only) or the Apple menu (Macintosh only). If the application requires more menus, you have to create and manage them using 4D methods.

• Editors: The editors, such as the Query and Order By editors, are no longer automatically available in the Custom Menus environment. If you want to use them in the Custom Menus environment, you have to call them using 4D methods.

The following sections include examples showing how the language can automate the use of a database.

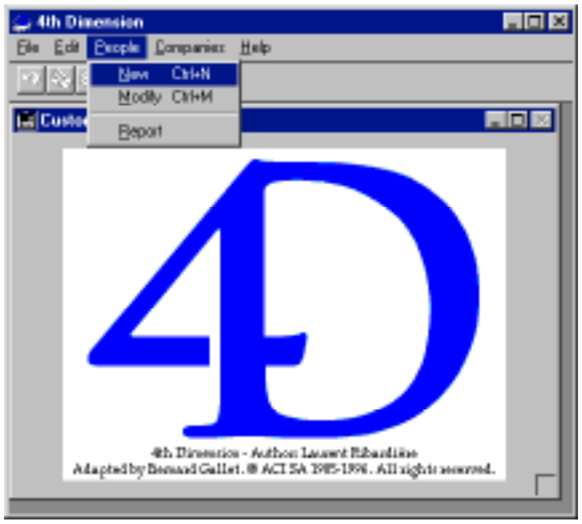
Custom Menus: an Example

Custom Menus are the primary interface in an application. They make it easier for users to learn and use a database. Creating custom menus is very simple—you associate methods with each menu command (also called menu items) in the Menu editor.

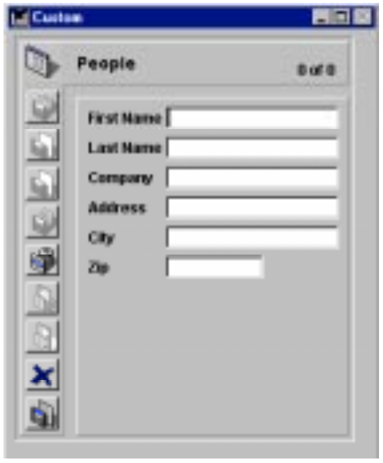
“The User's Perspective” section describes what happens when the user chooses a menu command. Next, “Behind the Scenes” describes the design work that made it happen. Although the example is simple, it should be apparent how custom menus make the database easier to use and learn. Rather than the “generic” tools and menu commands in the User environment, the user sees only things that are appropriate to his or her needs.

The User's Perspective

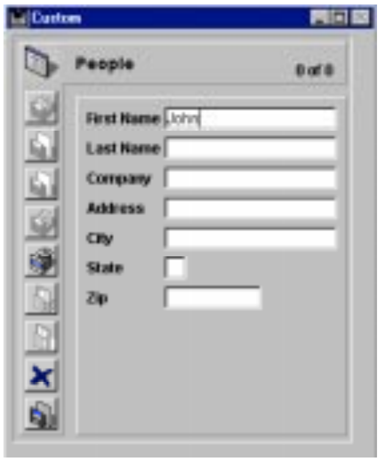
The user chooses a menu item called **New** from the **People** menu to add a new person to the database.


4th Dimension
File Edit People Companies Help
New Ctrl+N
Modify Ctrl+M
Report
Custo
4th Dimension - Author: Laurent Ribardière
Adapted by Bernard Gallet. © ACI SA 1985-1996. All rights reserved.

The Input form for the People table is displayed.

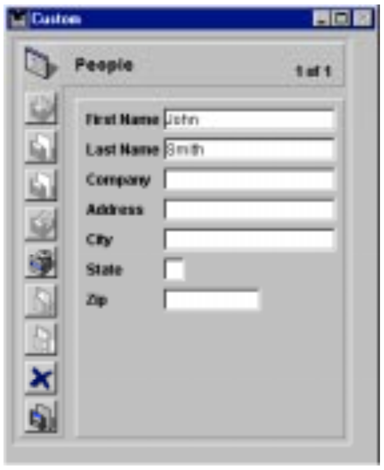


The user enters the person’s first name and then tabs to the next field.



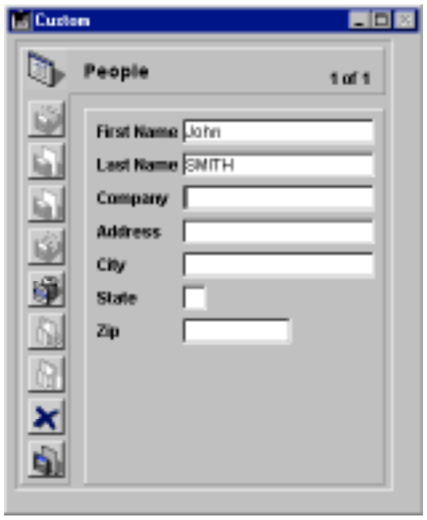
First Name	John
Last Name	
Company	
Address	
City	
State	
Zip	

The user enters the person’s last name and then tabs to the next field.



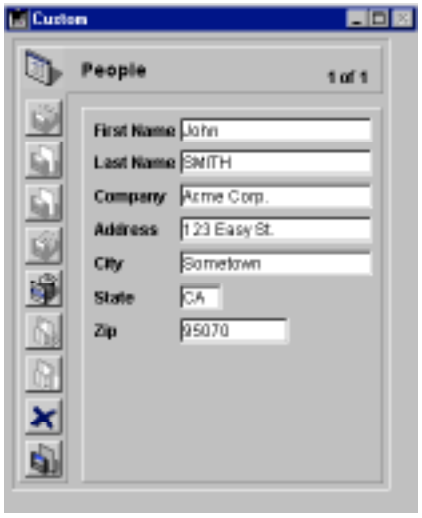
First Name	John
Last Name	Smith
Company	
Address	
City	
State	
Zip	

The user sees that the last name has been converted to uppercase.



First Name	John
Last Name	SMITH
Company	
Address	
City	
State	
Zip	

The user finishes entering the record and clicks the validation button (the last button in the vertical row of buttons).

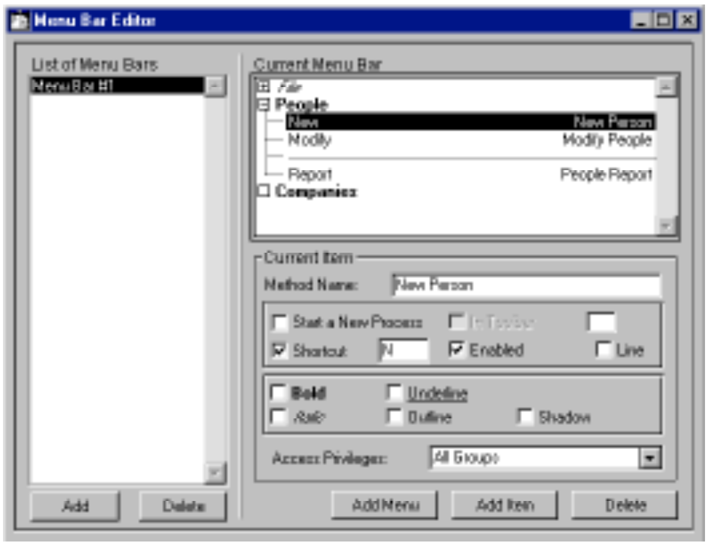


First Name	John
Last Name	SMITH
Company	Acme Corp.
Address	123 Easy St.
City	Sometown
State	CA
Zip	95070

Another blank record appears, and the user clicks the Cancel button (the one with the “X”) to terminate the “data entry loop.” The user is returned to the menu bar.

Behind the Scenes

The menu bar was created in the Design environment, using the Menu Bar Editor.



Menu Bar #1

```
⊞ File
⊟ People
   New                New Person
   Modify             Modify People
   Report             People Report
☐ Companies
```

Current Item

Method Name: New Person

☐ Start a New Process ☐ In Toolbar

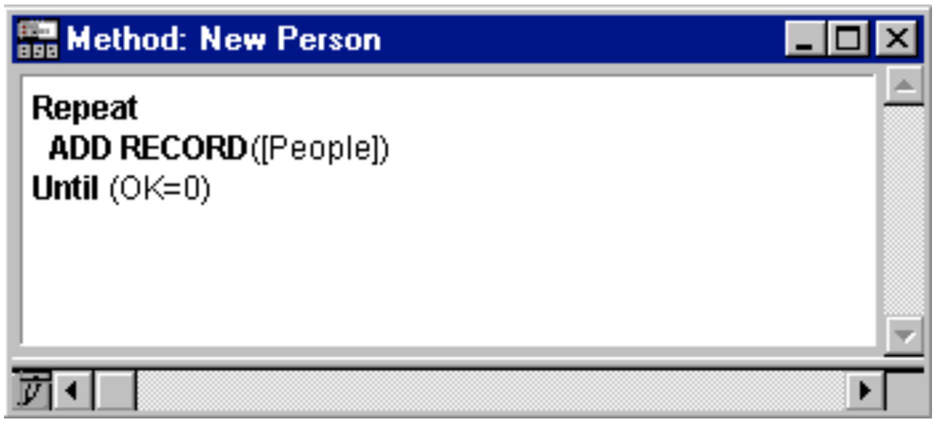
☑ Shortcut N ☑ Enabled ☐ Line

☐ Bold ☐ Underline

☐ Italic ☐ Outline ☐ Shadow

Access Privileges: All Groups

The menu item **New** has a project method named New Person associated with it. This method was created in the Design environment, using the Method editor.



```
Repeat
  ADD RECORD([People])
Until (OK=0)
```

When the user chooses this menu item, the New Person method executes:

```
Repeat
    ADD RECORD([People])
Until (OK=0)
```

The Repeat...Until loop with an ADD RECORD command within the loop acts just like the **New Record** menu item in the User environment. It displays the input form to the user, so that he or she can add a new record. When the user saves the record, another new blank record appears. This ADD RECORD loop continues to execute until the user clicks the Cancel button.

When a record is entered, the following occurs:
• There is no method for the First Name field, so nothing executes.
• There is a method for the Last Name field. This Object Method was created in the Design environment, using the Form and Method editors. The method executes:

```
Last Name:=Uppercase(Last Name)
```

This line converts the Last Name field to uppercase characters.

After a record has been entered, when the user clicks the Cancel button for the next one, the OK variable is set to zero, thus ending the execution of the ADD RECORD loop.

As there are no more statements to execute, the New Person method stops executing and control returns to the menu bar.

Comparing an Automated Task with the Actions to be performed in the User environment

Let's compare the way a task is performed in the User environment and the way the same task is performed using the language. The task is a common one:

• Find a group of records
• Sort them
• Print a report

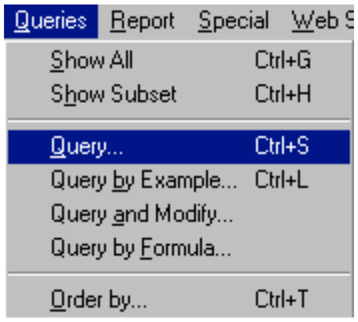
The next section, "Using a Database in the User Environment," displays the tasks performed in the User environment.

The following section, "Using the Built-in Editors within the Custom Menus environment," displays the same tasks performed in an application.

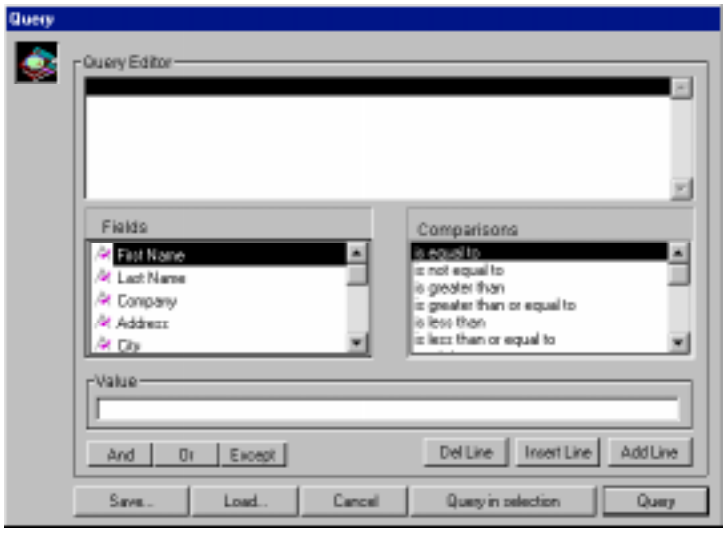
Note that although both methods perform the same task, the steps in the second section are automated using the language.

Using a database in the User environment

The user chooses **Query** from the **Queries** menu.

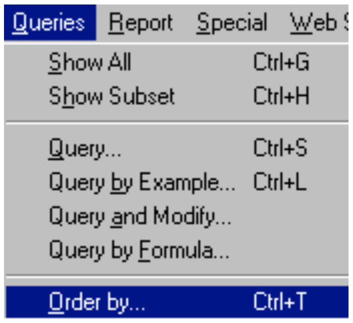


The **Query** editor is displayed.

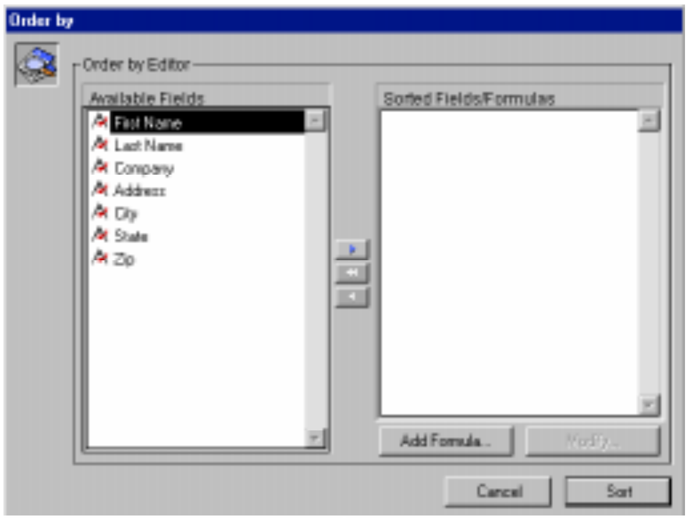


The user enters the criteria and clicks the **Query** button. The search is performed.

The user chooses **Order by** from the **Queries** menu.



The **Order By** editor is displayed.



The user enters the criteria and clicks the **Sort** button. The sort is performed.

Then, to print the records, these additional steps are required:
• The user chooses **Print** from the **File** menu.
• The **Choose Print Form** dialog box is displayed, because users need to know which form to print.
• The Printing dialog boxes are displayed. The user chooses the settings, and the report is printed.

Using the built-in editors within the Custom Menus environment

Let’s examine how this can be performed in the Custom Menus environment.

The User chooses **Report** from the **People** menu.

Even at this point, using an application is easier for the users—they did not need to know that querying is the first step!

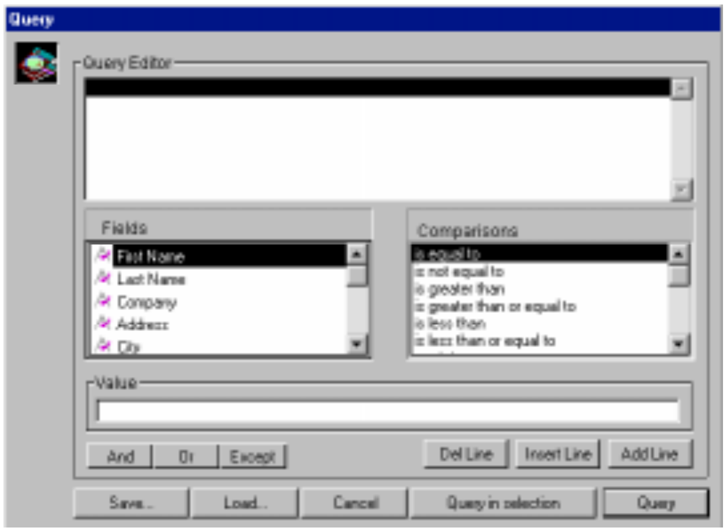
A method called My Report is attached to the menu command; it looks like this:

```
QUERY ([People])
ORDER BY ([People])
OUTPUT FORM ([People]; "Report")
PRINT SELECTION ([People])
```

The first line is executed:

```
QUERY ([People])
```

The **Query** editor is displayed.



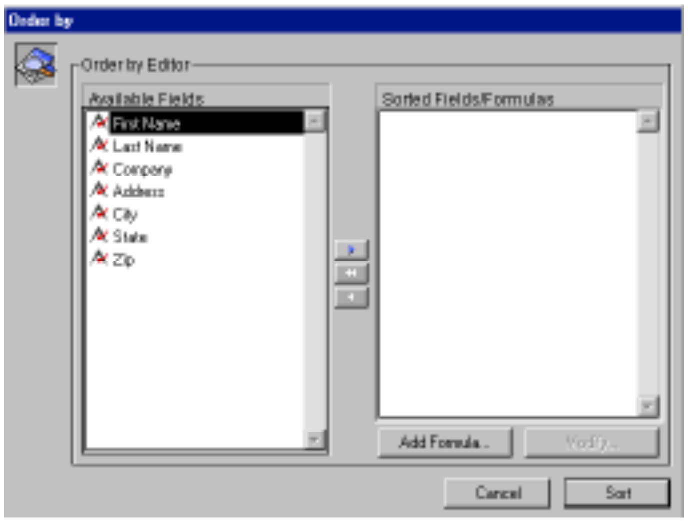
The user enters the criteria and clicks the **Query** button. The query is performed.

The second line of the My Report method is executed:

```
ORDER BY ([People])
```

Note that the user did not need to know that ordering the records was the next step.

The **Order By** Editor is displayed.



The user enters the criteria and clicks the **Sort** button. The sort is performed.

The third line of the My Report method is executed:

```
OUTPUT FORM ([People]; "Report")
```

Once again, the user did not need to know what to do next; the method takes care of that.

The final line of the My Report method is executed:

```
PRINT SELECTION ([People])
```

The **Printing** dialog boxes are displayed. The User chooses the settings, and the report is printed.

Automating the Application Further

The same commands used in the previous example can be used to further automate the database.

Let’s take a look at the new version of the My Report method.

The user chooses **Report** from the **People** menu. A method called My Report2 is attached to the menu command. It looks like this:

```
QUERY([People];[People]Company="Acme")
ORDER BY([People]; [People]Last Name;>;[People]First Name;>)
OUTPUT FORM([People];"Report")
PRINT SELECTION([People];*)
```

The first line is executed:

```
QUERY([People];[People]Company="Acme")
```

The **Query** editor is not displayed. Instead, the query is specified and performed by the QUERY command. The user does not need to do anything.

The second line of the My Report2 method is executed:

```
ORDER BY([People];[People]Last Name;>;[People]First Name;>)
```

The **Order By** editor is not displayed, and the sort is immediately performed. Once again, no user actions are required.

The final lines of the My Report2 method are executed:

```
OUTPUT FORM ([People]; "Report")
PRINT SELECTION ([People]; *)
```

The **Printing** dialog boxes are not displayed. The PRINT SELECTION command accepts an optional asterisk (*) parameter that instructs the command to use the print settings that were in effect when the report form was created. The report is printed.

This additional automation saved the user from having to enter options in three dialog boxes. Here are the benefits :

• The query is automatically performed: users may select wrong criteria when making a query.
• The sort is automatically performed: users may select wrong criteria when defining a sort.
• The printing is automatically performed: users may select the wrong form to print.

Tools for Developing 4D Applications

As you develop a 4D application, you will discover many capabilities that you did not notice when you started. You can even augment the standard version of 4D by adding other tools and plug-ins to your 4D development environment.

Development tools

ACI provides several tools that can be used for developing applications. These tools help you move objects from one database to another, compile your databases, and check your database syntactically. These tools include:

• **4D Insider** allows you to cross-reference your 4th Dimension databases. You can use it to view and print methods, variables, commands, externals, structures, lists, and forms. The cross-referencing utility tells you where each of these objects is used throughout your database. It also helps you to move objects like tables, forms, methods, menu bars, lists, packages, and styles from one database to another.

• **4D Compiler** translates your methods and scripts into assembly-level instructions. This increases the execution speed of your databases, checks the consistency of the code, and detects logical and syntactical conflicts. Furthermore, it protects your database from being viewed or modified, deliberately or inadvertently.

4D Plug-ins

You can extend the capabilities of your 4D applications by adding professional **Plug-ins** to your 4D development environment.

ACI provides the following **Productivity** Plug-ins:

• **4D Write**: Word-processor
• **4D Calc**: Spreadsheet
• **4D Draw**: Graphical drawing program

ACI also provides the following **Connectivity** Plug-ins:

• **4D ODBC**: Connectivity via ODBC
• **4D ORACLE**: Connectivity with ORACLE databases
• **4D SQL SERVER**: Connectivity with SYBASE SQL Server and Microsoft SQL Server
• **4D Open**: Connectivity (from 4D to 4D) for building distributed 4D information systems.

For more information, contact ACI or its Partners. Visit our Web Sites:

USA and International	http://www.acius.com
France and International	http://www.aci.fr
Japan and Asia	http://www.aci.co.jp

The 4D community and third party tools

There is a very active worldwide 4D community, composed of User Groups, Electronic Forums, and ACI Partners. ACI Partners produce **Third Party Tools**, such as Area List Pro from Foresight Technology, Inc. (http://www.fsti.com).

Browse your 4D CD—it contains demos and information from ACI Partners. Find out about them on the Web. Subscribe to **Dimensions** magazine (Mark Yelich, Publisher, myelichcps@aol.com).

The 4D community offers access to tips and tricks, solutions, information, and additional tools that will save you time and energy, and increase your productivity.

2 Language Definition

Introduction to the 4D Language

Language Definition

version 6.0

The 4th Dimension language is made up of various components that help you perform tasks and manage your data.

• **Data types**: Classifications of data in a database. See discussion in this section as well as the detailed discussion in the section Data Types.
• **Variables**: Temporary storage places for data in memory. See detailed discussion in the section Variables.
• **Operators**: Symbols that perform a calculation between two values. See discussion in this section as well as the detailed discussion in the section Operators and its subsections.
• **Expressions**: Combinations of other components that result in a value. See discussion in this section.
• **Commands**: Built-in instructions to perform an action. All 4D commands, such as ADD RECORD, are described in this manual, grouped by theme; when necessary, the theme is preceded by an introductory section. You can use 4D Plug-ins to add new commands to your 4D development environment. For example, once you have added the 4D Write Plug-in to your 4D system, the 4D Write commands become available for creating and manipulating word-processing documents.
• **Methods**: Instructions that you write using all parts of the language listed here. See discussion in the section Methods and its subsections.

This section introduces Data Types, Operators, and Expressions. For the other components, refer to the sections cited above.

In addition:
• Language components, such as variables, have names called Identifiers. For a detailed discussion about identifiers and the rules for naming objects, refer to the section Identifiers.
• To learn more about array variables, refer to the section Arrays.
• To learn more about BLOB variables, refer to the section BLOB commands.
• If you plan to compile your database, refer to the section Compiler Commands as well as the *4D Compiler Reference Guide*.

Data Types

In the language, the various types of data that can be stored in a 4th Dimension database are referred to as data types. There are seven basic data types: string, numeric, date, time, Boolean, picture, and pointer.

• **String**: A series of characters, such as “Hello there”. Alpha and Text fields, and string and text variables, are of the string data type.
• **Numeric**: Numbers, such as 2 or 1,000.67. Integer, Long Integer, and Real fields and variables are of the numeric data type.
• **Date**: Calendar dates, such as 1/20/89. Date fields and variables are of the date data type.
• **Time**: Times, including hours, minutes, and seconds, such as 1:00:00 or 4:35:30 PM. Time fields and variables are of the time data type.
• **Boolean**: Logical values of TRUE or FALSE. Boolean fields and variables are of the Boolean data type.
• **Picture**: Picture fields and variables are of the picture data type.
• **Pointer**: A special type of data used in advanced programming. Pointer variables are of the pointer data type. There is no corresponding field type.

Note that in the list of data types, the string and numeric data types are associated with more than one type of field. When data is put into a field, the language automatically converts the data to the correct type for the field. For example, if an integer field is used, its data is automatically treated as numeric. In other words, you need not worry about mixing similar field types when using the language; it will manage them for you.

However, when using the language it is important that you do not mix different data types. In the same way that it makes no sense to store “ABC” in a Date field, it makes no sense to put “ABC” in a variable used for dates. In most cases, 4th Dimension is very tolerant and will try to make sense of what you are doing. For example, if you add a number to a date, 4th Dimension will assume that you want to add that number of days to the date, but if you try to add a string to a date, 4th Dimension will tell you that the operation cannot work.

There are cases in which you need to store data as one type and use it as another type. The language contains a full complement of commands that let you convert from one data type to another. For example, you may need to create a part number that starts with a number and ends with characters such as “abc”. In this case, you might write:

```
[Products]Part Number:=String(Number)+"abc"
```

If Number is 17, then [Products]Part Number will get the string “17abc”.

The data types are fully defined in the section Data Types.

Operators

When you use the language, it is rare that you will simply want a piece of data. It is more likely that you will want to do something to or with that data. You perform such calculations with operators. Operators, in general, take two pieces of data and perform an operation on them that results in a new piece of data. You are already familiar with many operators. For example, 1 + 2 uses the addition (or plus sign) operator to add two numbers together, and the result is 3. This table shows some familiar numeric operators:

Operator	Operation	Example
+	Addition	1 + 2 results in 3
–	Subtraction	3 – 2 results in 1
*	Multiplication	2 * 3 results in 6
/	Division	6 / 2 results in 3

Numeric operators are just one type of operator available to you. 4th Dimension supports many different types of data, such as numbers, text, dates, and pictures, so there are operators that perform operations on these different data types.

The same symbols are often used for different operations, depending on the data type. For example, the plus sign (+) performs different operations with different data:

Data Type	Operation	Example
Number	Addition	1 + 2 adds the numbers and results in 3
String	Concatenation	"Hello " + "there" concatenates (joins together) the strings and results in "Hello there"
Date and Number	Date addition	!1/1/1989! + 20 adds 20 days to the date January 1, 1989, and results in the date January 21, 1989

The operators are fully defined in the section Operators and its subsections.

Expressions

Simply put, expressions return a value. In fact, when using the 4th Dimension language, you use expressions all the time and tend to think of them only in terms of the value they represent. Expressions are also sometimes referred to as formulas.

Expressions are made up of almost all the other parts of the language: commands, operators, variables, and fields. You use expressions to build statements (lines of code), which in turn are used to build methods. The language uses expressions wherever it needs a piece of data.

Expressions rarely “stand alone.” There are only a few places in 4th Dimension where an expression can be used by itself:
• Query by Formula dialog box in the User environment
• Debugger where the value of expressions can be checked
• Apply Formula dialog box
• Quick Report editor as a formula for a column

An expression can simply be a constant, such as the number 4 or the string “Hello.” As the name implies, a constant’s value never changes. It is when operators are introduced that expressions start to get interesting. In preceding sections you have already seen expressions that use operators. For example, 4 + 2 is an expression that uses the addition operator to add two numbers together and return the result 6.

You refer to an expression by the data type it returns. There are seven expression types:
• String expression
• Numeric expression (also referred to as number)
• Date expression
• Time expression
• Boolean expression
• Picture expression
• Pointer expression

The following table gives examples of each of the seven types of expressions.

Expression	**Type**	**Explanation**
“Hello”	String	The word Hello is a string constant, indicated by the double quotation marks.
“Hello ” + “there”	String	Two strings, “Hello ” and “there”, are added together (concatenated) with the string concatenation operator (+). The string “Hello there” is returned.
“Mr. ” + [People]Name	String	Two strings are concatenated: the string “Mr. ” and the current value of the Name field in the People table. If the field contains “Smith”, the expression returns “Mr. Smith”.
Uppercase (“smith”)	String	This expression uses “Uppercase”, a command from the language, to convert the string “smith” to uppercase. It returns “SMITH”.
4	Number	This is a number constant, 4.
4 * 2	Number	Two numbers, 4 and 2, are multiplied using the multiplication operator (*). The result is the number 8.

My Button	Number	This is the name of a button. It returns the current value of the button: 1 if it was clicked, 0 if not.
!1/25/97!	Date	This is a date constant for the date 1/25/97 (January 25, 1997).
Current date + 30	Date	This is a date expression that uses the command “Current date” to get today’s date. It adds 30 days to today’s date and returns the new date.
?8:05:30?	Time	This is a time constant that represents 8 hours, 5 minutes, and 30 seconds.
?2:03:04? + ?1:02:03?	Time	This expression adds two times together and returns the time 3:05:07.
True	Boolean	This command returns the Boolean value TRUE.
10 # 20	Boolean	This is a logical comparison between two numbers. The number sign (#) means “is not equal to”. Since 10 “is not equal to” 20, the expression returns TRUE.
“ABC” = “XYZ”	Boolean	This is a logical comparison between two strings. They are not equal, so the expression returns FALSE.
My Picture + 50	Picture	This expression takes the picture in My Picture, moves it 50 pixels to the right, and returns the resulting picture.
->[People]Name	Pointer	This expression returns a pointer to the field called [People]Name.
Table (1)	Pointer	This is a command that returns a pointer to the first table.

See Also

Arrays, Constants, Data Types, Methods, Operators, Pointers, Variables.

Data Types

4th Dimension fields, variables, and expressions can be of the following data types:

Data Type	Field	Variable	Expression
String (see note 1)	Yes	Yes	Yes
Number (see note 2)	Yes	Yes	Yes
Date	Yes	Yes	Yes
Time	Yes	Yes	Yes
Boolean	Yes	Yes	Yes
Picture	Yes	Yes	Yes
Pointer	No	Yes	Yes
BLOB (see note 3)	Yes	Yes	No
Array (see note 4)	No	Yes	No
Subtable	Yes	No	No
Undefined	No	Yes	Yes

Notes

1. String includes alphanumeric field, fixed length variable, and text field or variable.
2. Number includes Real, Integer, and Long Integer field and variable.
3. BLOB is an abbreviation for Binary Large OBject. For more information about BLOBs, see the section BLOB Commands.
4. Array includes all types of arrays. For more information, see the section Arrays.

String

String is a generic term that stands for:
• Alphanumeric field
• Fixed length variable
• Text field or variable
• Any string or text expression

A string is composed of characters. Each character can be any of the 256 ASCII codes. For more information about ASCII codes and how 4D handles them in a cross-platform environment, see the section ASCII Codes.

• An Alphanumeric field may contain from 0 to 80 characters (the limit depends on the field definition).
• A Fixed length variable may contain from 0 to 255 (the limit depends on the variable declaration).
• A Text field, variable, or expression may contain from 0 to 32,000 characters.

You can assign a string to a text field and vice-versa; 4D does the conversion, truncating if necessary. You can mix string and text in an expression.

Note: In the *4D Language Reference* manual, both string and text parameters in command descriptions are denoted as String, except when marked otherwise.

Number

Number is a generic term that stands for:
• Real Field, variable or expression
• Integer field, variable or expression
• Long Integer field, variable or expression

The range for the Real data type is ±1.7e±308 (15 digits)
The range for the Integer data type (2-byte Integer) is -32,768..32,767 (2^15..(2^15)-1)
The range for the Long Integer data type (4-byte Integer) is -2^31..(2^31)-1

You can assign any Number data type to another one; 4D does the conversion, truncating or rounding, if necessary. Note however, that when values are out of range, the conversion will not return a valid value. You can mix Number data types in expression.

Note: In the *4D Language Reference* manual, no matter the actual data type, Real, Integer, and Long Integer parameters in command descriptions are denoted as Number, except when marked otherwise.

Date

• A Date field, variable, or expression can be in the range of 1/1/100 to 12/31/32,767.
• Using the US English version of 4D, a date is ordered month/day/year.
• If a year is given as two digits, it is assumed to be in the 1900's (unless this default has been changed using the command SET DEFAULT CENTURY).

Note: In the *4D Language Reference* manual, Date parameters in command descriptions are denoted as Date, except when marked otherwise.

Time

• A Time field, variable or expression can be in the range of 00:00:00 to 596,000:00:00.
• Using the US English version of 4D, time is ordered hour:minute:second.
• Times are in 24-hour format.
• A time value can be treated as a number. The number returned from a time is the number of seconds that time represents. For more information, see the section Time Operators.

Note: In the *4D Language Reference* manual, Time parameters in command descriptions are denoted as Time, except when marked otherwise.

Boolean

A Boolean field, variable or expression can be either TRUE or FALSE.

Note: In the *4D Language Reference* manual, Boolean parameters in command descriptions are denoted as Boolean, except when marked otherwise.

Picture

A Picture field, variable, or expression can be any Windows or Macintosh picture. In general, this includes any picture that can be put on the Clipboard or read from the disk using the 4D commands or Plug-Ins commands.

Note: In the *4D Language Reference* manual, Picture parameters in command descriptions are denoted as Picture, except when marked otherwise.

Pointer

A Pointer variable or expression is a reference to another variable (including arrays and array elements), table, or field. There is no field of type Pointer.

For more information about Pointers, see the section Pointers.

Note: In the *4D Language Reference* manual, Pointer parameters in command descriptions are denoted as Pointer except when marked otherwise.

BLOB

A BLOB field or variable is a series of bytes (from 0 to 2 GB in length) that you can address individually or by using the BLOB Commands. There is no expression of type BLOB.

Note: In the *4D Language Reference* manual, BLOB parameters in command descriptions are denoted as BLOB.

Array

Array is not actually a data type. The various types of arrays (such as Integer Array, Text Array, and so on) are grouped under this title. Arrays are variables—there is no field of type Array, and there is no expression of type Array. For more information about arrays, see the section Arrays.

Note: In the *4D Language Reference* manual, Array parameters in command descriptions are denoted as Array, except when marked otherwise (i.e., String Array, Numeric Array, ...).

Subtable

Subtable is not actually a data type. Only fields can be of type Subtable. There is no variable or expression of type Subtable. For more information about subtables, see the *4th Dimension Design Reference* manual as well as the commands regrouped under the theme Subrecords.

Undefined

Undefined is not actually a data type. Undefined denotes a variable that has not yet been defined. A function (a project method that returns a result) can return an undefined value if, within the method, the function result ($0) is assigned an undefined expression (an expression calculated with at least one undefined variable). A field cannot be undefined.

Converting Data Types

The 4D language contains operators and commands to convert between data types, where such conversions are meaningful. The 4D language enforces data type checking. For example, you cannot write: "abc"+0.5+!12/25/96!-?00:30:45?. This will generate syntax errors.

The following table lists the basic data types, the data types to which they can be converted, and the commands used to do so:

Data Type	**Convert to** String	**Convert to** Number	**Convert to** Date	**Convert to** Time
String		Num	Date	Time
Number (*)	String			
Date	String			
Time	String			
Boolean		Num		

(*) Time values can be be treated as numbers.

Note: In addition to the data conversions listed inthis table, more sophisticated data conversions can be obtained by combining operators and other commands.

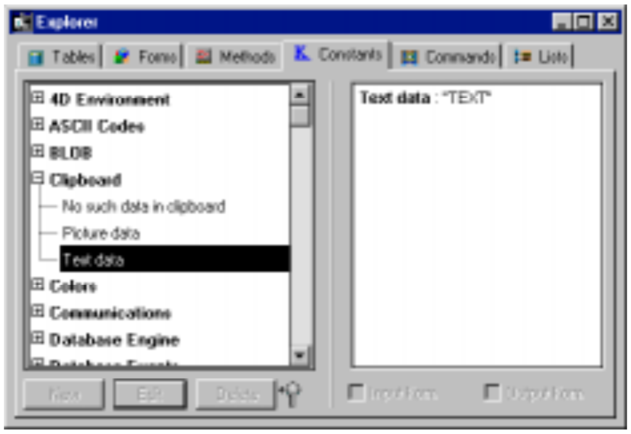
See Also

Arrays, Constants, Control Flow, Identifiers, Methods, Operators, Pointers, Type, Variables.

A constant is an expression that has a fixed value. There are two types of constants: **predefined constants** that you select by name, and **literal constants** for which you type the actual value.

Predefined Constants

Version 6 of 4th Dimension introduces **predefined constants.** These constants are listed in the Explorer Window:



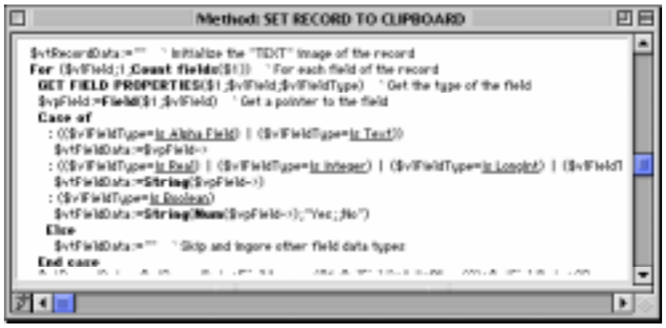
The predefined constants are listed by theme. To use a predefined constant in a **Method editor** window:
• Drag and drop the constant from the Explorer window to the Method Editor window.
• Directly type its name in the Method Editor window.

Predefined constant names can contain up to 31 characters.

Tip: If you directly enter the name of a predefined constant, you can use the @ symbol (at sign) to avoid typing the entire constant name. For example, if you type "No such da@", 4D will fill the line with the constant "No such data in clipboard" when you press Return or Enter to validate the line of code.

Note: The predefined constants (about 500) are listed by theme in this manual. See the section About this manual for more information. When appropriate, predefined constants are also listed in the command descriptions.

Predefined constants appear underlined within the Method Editor and Debugger windows:



Method: SET RECORD TO CLIPBOARD

```
$vtRecordData:=""   ` Initialize the "TEXT" image of the record
For ($vlField;1;Count fields($1))   ` For each field of the record
 GET FIELD PROPERTIES($1;$vlField;$vlFieldType)   ` Get the type of the field
 $vpField:=Field($1;$vlField)   ` Get a pointer to the field
 Case of
   : (($vlFieldType=Is Alpha Field) | ($vlFieldType=Is Text))
    $vtFieldData:=$vpField->
   : (($vlFieldType=Is Real) | ($vlFieldType=Is Integer) | ($vlFieldType=Is LongInt) | ($vlFieldT
    $vtFieldData:=String($vpField->)
   : ($vlFieldType=Is Boolean)
    $vtFieldData:=String(Num($vpField->);"Yes;;No")
  Else
    $vtFieldData:=""   ` Skip and ignore other field data types
 End case
```

In the window shown here, Is Alpha Field, for example, is a predefined constant.

Literal Constants

Literal Constants can be of four data types:
• String
• Numeric
• Date
• Time

String Constants

A string constant is enclosed in double, straight quotation marks ("..."). Here are some examples of string constants:

"Add Records"
"No records found."
"Invoice"

An empty string is specified by two quotation marks with nothing between them ("").

Numeric Constants

A numeric constant is written as a real number. Here are some examples of numeric constants:

```
27
123.76
0.0076
```

Negative numbers are specified with the minus sign(–). For example:

```
–27
–123.76
–0.0076
```

Date Constants

A date constant is enclosed by exclamation marks (!...!). In the US English version of 4D, a date is ordered month/day/year, with a slash (/) setting off each part. Here are some examples of date constants:

```
!1/1/76!
!4/4/04!
!12/25/96!
```

A null date is specified by !00/00/00!

Tip: The Method Editor includes a shortcut for entering a null date. To type a null date, enter the exclamation (!) character and press Enter.

Note: A two-digit year is assumed to be in the 1900's. Unless this default setting has been changed using the command SET DEFAULT CENTURY.

Time Constants

A time constant is enclosed by question marks (?...?).

Note: This syntax can be used on both Windows and Macintosh. On Macintosh, you can also use the Dagger symbol (Option-T on a US keyboard).

In the US English version of 4D, a time constant is ordered hour:minute:second, with a colon (:) setting off each part. Times are specified in 24-hour format.

Here are some examples of time constants:

```
?00:00:00? ` midnight
?09:30:00? ` 9:30 am
?13:01:59? ` 1 pm, 1 minute, and 59 seconds
```

A null time is specified by ?00:00:00?

Tip: The Method Editor includes a shortcut for entering a null time. To type a null time, enter the question mark (?) character and press Enter.

See Also

Control Flow, Data Types, Identifiers, Methods, Operators, Pointers, Variables.

Data in 4th Dimension is stored in two fundamentally different ways. Fields store data permanently on disk; variables store data temporarily in memory.

When you set up your 4th Dimension database, you specify the names and types of fields that you want to use. Variables are much the same—you also give them names and different types.

The following variable types correspond to each of the data types:

- String: Fixed alphanumeric string of up to 255 characters
- Text: Alphanumeric string of up to 32,000 characters
- Integer: Integer from -32768 to 32767
- Long Integer: Integer from -2^31 to (2^31)-1
- Real: A number to ±1.7e±308 (15 digits)
- Date: 1/1/100 to 12/31/32767
- Time: 00:00:00 to 596000:00:00 (seconds from midnight)
- Boolean: True or False
- Picture: Any Windows or Macintosh picture
- BLOB (Binary Large OBject): Series of bytes up to 2 GB in size
- Pointer: A pointer to a table, field, variable, array, or array element

You can display variables (except Pointer and BLOB) on the screen, enter data into them, and print them in reports. In these ways, enterable and non-enterable area variables act just like fields, and the same built-in controls are available when you create them:

- Display formats
- Data validation, such entry filters and default values
- Character filters
- Choice lists (hierarchical lists)
- Enterable or non-enterable values

Variables can also do the following:

- Control buttons (buttons, check boxes, radio buttons, 3D buttons, and so on)
- Control sliders (meters, rulers, and dials)
- Control scrollable areas, pop-up menus, and drop-down list boxes
- Control hierarchical lists and hierarchical pop-up menus
- Control button grids, tab controls, picture buttons, and so on
- Display results of calculations that do not need to be saved.

Creating Variables

You create variables simply by using them; you do not need to formally define them as you do with fields. For example, if you want a variable that will hold the current date plus 30 days, you write:

```
MyDate:=Current date+30
```

4th Dimension creates MyDate and holds the date you need. The line of code reads “MyDate gets the current date plus 30 days.” You could now use MyDate wherever you need it in your database. For example, you might need to store the date variable in a field of same type:

```
[MyTable]MyField:=MyDate
```

Sometimes you may want a variable to be explicitly defined as a certain type. For more information about typing variables for a database that you intend to compile, see the section Compiler Commands.

Assigning Data to Variables

Data can be put into and copied out of variables. Putting data into a variable is called **assigning the data to the variable** and is done with the assignment operator (:=). The assignment operator is also used to assign data to fields.

The assignment operator is the primary way to create a variable and to put data into it. You write the name of the variable that you want to create on the left side of the assignment operator. For example:

```
MyNumber:=3
```

creates the variable MyNumber and puts the number 3 into it. If MyNumber already exists, then the number 3 is just put into it.

Of course, variables would not be very useful if you could not get data out of them. Once again, you use the assignment operator. If you need to put the value of MyNumber in a field called [Products]Size, you would write MyNumber on the right side of the assignment operator:

```
[Products]Size:=MyNumber
```

In this case, [Products]Size would be equal to 3. This example is rather simple, but it illustrates the fundamental way that data is transferred from one place to another by using the language.

Important: Be careful not to confuse the assignment operator (:=) with the comparison operator, equal (=). Assignment and comparison are very different operations. For more information about the comparison operators, see the section Operators.

Local, Process, and Interprocess Variables

You can create three types of variables: **local** variables, **process** variables, and **interprocess** variables. The difference between the three types of variables is their scope, or the objects to which they are available.

Local variables

A local variable is, as its name implies, local to a method—accessible only within the method in which it was created and not accessible outside of that method. Being local to a method is formally referred to as being "local in scope." Local variables are used to restrict a variable so that it works only within the method.

You may want to use a local variable to:
• Avoid conflicts with the names of other variables
• Use data temporarily
• Reduce the number of process variables

The name of a local variable always starts with a dollar sign ($) and can contain up to 31 additional characters. If you enter a longer name, 4th Dimension truncates it to the appropriate length.

When you are working in a database with many methods and variables, you often find that you need to use a variable only within the method on which you are working. You can create and use a local variable in the method without worrying about whether you have used the same variable name somewhere else.

Frequently, in a database, small pieces of information are needed from the user. The Request command can obtain this information. It displays a dialog box with a message prompting the user for a response. When the user enters the response, the command returns the information the user entered. You usually do not need to keep this information in your methods for very long. This is a typical way to use a local variable. Here is an example:

```
$vsID:=Request("Please enter your ID:")
If (OK=1)
   QUERY ([People];[People]ID =$vsID)
End if
```

This method simply asks the user to enter an ID. It puts the response into a local variable, $vsID, and then searches for the ID that the user entered. When this method finishes, the $vsID local variable is erased from memory. This is fine, because the variable is needed only once and only in this method.

Process variables
A process variable is available only within a process. It is accessible to the process method and any other method called from within the process.

A process variable does not have a prefix before its name. A process variable name can contain up to 31 characters.

In interpreted mode, variables are maintained dynamically, they are created and erased from memory “on the fly.” In compiled mode, all processes you create (user processes) share the same definition of process variables, but each process has a different instance for each variable. For example, the variable myVar is one variable in the process P_1 and another one in the process P_2.

Starting with version 6, a process can “peek and poke” process variables from another process using the commands GET PROCESS VARIABLE and SET PROCESS VARIABLE. It is good programming practice to restrict the use of these commands to the situation for which they were added to 4D:
• Interprocess communication at specific places or your code
• Handling of interprocess drag and drop
• In Client/Server, communication between processes on client machines and the stored procedures running on the server machines

For more information, see the section Processes and the description of these commands.

Interprocess variables
Interprocess variables are available throughout the database and are shared by all processes. They are primarily used to share information between processes.

The name of an interprocess variable always begins with the symbols (<>) — a “less than” sign followed by a “greater than” sign— followed by 31 characters.

Note: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

In Client/Server, each machine (Client machines and Server machine) share the same definition of interprocess variables, but each machine has a different instance for each variable.

Form Object Variables

In the Form editor, naming an active object—button, radio button, check box, scrollable area, meter bar, and so on—automatically creates a variable with the same name. For example, if you create a button named MyButton, a variable named MyButton is also created. Note that this variable name is not the label for the button, but is the name of the button.

The form object variables allow you to control and monitor the objects. For example, when a button is clicked, its variable is set to 1; at all other times, it is 0. The variable associated with a meter or dial lets you read and change the current setting. For example, if you drag a meter to a new setting, the value of the variable changes to reflect the new setting. Similarly, if a method changes the value of the variable, the meter is redrawn to show the new value.

For more information about variables and forms, see the 4th Dimension Design Reference Manual as well as the section Form event.

System Variables

4th Dimension maintains a number of variables called **system variables**. These variables let you monitor many operations. System variables are all process variables, accessible only from within a process.

The most important system variable is the **OK** system variable. As its name implies, it tells you if everything is OK in the particular process. Was the record saved? Has the importing operation been completed? Did the user click the OK button? The OK system variable is set to 1 when a task is completed successfully, and to 0 when it is not.

For more information about system variables, see the section System Variables.

See Also

Arrays, Constants, Control Flow, Data Types, Identifiers, Methods, Operators, Pointers.

Pointers

Language Definition

version 6.0

Pointers provide an advanced way (in programming) to refer to data.

When you use the language, you access various objects—in particular, tables, fields, variables, and arrays—by simply using their names. However, it is often useful to refer to these elements and access them without knowing their names. This is what pointers let you do.

The concept behind pointers is not that uncommon in everyday life. You often refer to something without knowing its exact identity. For example, you might say to a friend, “Let’s go for a ride in your car” instead of “Let’s go for a ride in the car with license plate 123ABD.” In this case, you are referencing the car with license plate 123ABD by using the phrase “your car.” The phrase “car with license plate 123ABD” is like the name of an object, and using the phrase “your car” is like using a pointer to reference the object.

Being able to refer to something without knowing its exact identity is very useful. In fact, your friend could get a new car, and the phrase “your car” would still be accurate—it would still be a car and you could still take a ride in it. Pointers work the same way. For example, a pointer could at one time refer to a numeric field called Age, and later refer to a numeric variable called Old Age. In both cases, the pointer references numeric data that could be used in a calculation.

You can use pointers to reference tables, fields, variables, arrays, and array elements. The following table gives an example of each data type:

Object	**To Reference**	**To Use**	**To Assign**
Table	vpTable:=->[Table]	DEFAULT TABLE(vpTable->)	n/a
Field	vpField:=->[Table]Field	ALERT(vpField->)	vpField->:="John"
Variable	vpVar:=->Variable	ALERT(vpVar->)	vpVar->:="John"
Array	vpArr:=->Array	SORT ARRAY(vpArr->;>)	COPY ARRAY (Arr;vpArr->)
Array element	vpElem:=->Array{1}	ALERT (vpElem->)	vpElem->:="John"

Using Pointers: An Example

It is easiest to explain the use of pointers through an example. This example shows how to access a variable through a pointer. We start by creating a variable:

```
MyVar:="Hello"
```

MyVar is now a variable containing the string "Hello." We can now create a pointer to MyVar:

```
MyPointer:=->MyVar
```

The -> symbol means "get a pointer to." This symbol is formed by a dash followed by a "greater than" sign. In this case, it gets the pointer that references or "points to" MyVar. This pointer is assigned to MyPointer with the assignment operator.

MyPointer is now a variable that contains a pointer to MyVar. MyPointer does not contain "Hello", which is the value in MyVar, but you can use MyPointer to get this value. The following expression returns the value in MyVar:

```
MyPointer->
```

In this case, it returns the string "Hello". The -> symbol, when it follows a pointer, references the object pointed to. This is called **dereferencing**.

It is important to understand that you can use a pointer followed by the -> symbol anywhere that you could have used the object that the pointer points to. This means that you could use the expression MyPointer-> anywhere that you could use the original MyVar variable.

For example, the following line displays an alert box with the word Hello in it:

```
ALERT(MyPointer->)
```

You can also use MyPointer to change the data in MyVar. For example, the following statement stores the string "Goodbye" in the variable MyVar:

```
MyPointer->:="Goodbye"
```

If you examine the two uses of the expression MyPointer->, you will see that it acts just as if you had used MyVar instead. In summary, the following two lines perform the same action—both display an alert box containing the current value in the variable MyVar:

```
ALERT(MyPointer->)
ALERT(MyVar)
```

The following two lines perform the same action— both assign the string "Goodbye" to MyVar:

```
MyPointer->:="Goodbye"
MyVar:="Goodbye"
```

Using Pointers to Buttons

This section describes how to use a pointer to reference a button. A button is (from the language point of view) nothing more than a variable. Although the examples in this section use pointers to reference buttons, the concepts presented here apply to the use of all types of objects that can be referenced by a pointer.

Let's say that you have a number of buttons in your forms that need to be enabled or disabled. Each button has a condition associated with it that is TRUE or FALSE. The condition says whether to disable or enable the button. You could use a test like this each time you need to enable or disable the button:

```
If (Condition) ` If the condition is TRUE…
   ENABLE BUTTON (MyButton) ` enable the button
Else ` Otherwise…
   DISABLE BUTTON (MyButton) ` disable the button
End if
```

You would need to use a similar test for every button you set, with only the name of the button changing. To be more efficient, you could use a pointer to reference each button and then use a subroutine for the test itself.

You must use pointers if you use a subroutine, because you cannot refer to the button's variables in any other way. For example, here is a project method called SET BUTTON, which references a button with a pointer:

```
   ` SET BUTTON project method
   ` SET BUTTON ( Pointer ; Boolean )
   ` SET BUTTON ( -> Button ; Enable or Disable )
   `
   ` $1 – Pointer to a button
   ` $2 – Boolean. If TRUE, enable the button. If FALSE, disable the button

If ($2) ` If the condition is TRUE…
   ENABLE BUTTON($1->) ` enable the button
Else ` Otherwise…
   DISABLE BUTTON($1->) ` disable the button
End if
```

You can call the SET BUTTON project method as follows:

```
    ` ...
SET BUTTON (->bValidate;True)
    ` ...
SET BUTTON (->bValidate;False)
    ` ...
SET BUTTON (->bValidate;([Employee]Last Name#""))
    ` ...
For ($vlRadioButton;1;20)
    $vpRadioButton:=Get pointer("r"+String($vlRadioButton))
    SET BUTTON ($vpRadioButton;False)
End for
```

Using Pointers to Tables

Anywhere that the language expects to see a table, you can use a dereferenced pointer to the table.
You create a pointer to a table by using a line like this:

```
TablePtr:=->[anyTable]
```

You can also get a pointer to a table by using the Table command. For example:

```
TablePtr:=Table(20)
```

You can use the dereferenced pointer in commands, like this:

```
DEFAULT TABLE(TablePtr->)
```

Using Pointers to Fields

Anywhere that the language expects to see a field, you can use a dereferenced pointer to reference the field. You create a pointer to a field by using a line like this:

```
FieldPtr:=->[aTable]ThisField
```

You can also get a pointer to a field by using the Field command. For example:

```
FieldPtr:=Field(1; 2)
```

You can use the dereferenced pointer in commands, like this:

```
FONT(FieldPtr->; "Arial")
```

Using Pointers to Array Elements

You can create a pointer to an array element. For example, the following lines create an array and assign a pointer to the first array element to a variable called ElemPtr:

```
ARRAY REAL(anArray; 10) ` Create an array
ElemPtr:=->anArray{1} ` Create a pointer to the array element
```

You could use the dereferenced pointer to assign a value to the element, like this:

```
ElemPtr->:=8
```

Using Pointers to Arrays

You can create a pointer to an array. For example, the following lines create an array and assign a pointer to the array to a variable called ArrPtr:

```
ARRAY REAL(anArray; 10) ` Create an array
ArrPtr := ->anArray ` Create a pointer to the array
```

It is important to understand that the pointer points to the array; it does not point to an element of the array. For example, you can use the dereferenced pointer from the preceding lines like this:

```
SORT ARRAY(ArrPtr->; >) ` Sort the array
```

If you need to refer to the fourth element in the array by using the pointer, you do this:

```
ArrPtr->{4} := 84
```

Using an Array of Pointers

It is often useful to have an array of pointers that reference a group of related objects.

One example of such a group of objects is a grid of variables in a form. Each variable in the grid is sequentially numbered, for example: Var1,Var2,..., Var10. You often need to reference these variables indirectly with a number. If you create an array of pointers, and initialize the pointers to point to each variable, you can then easily reference the variables. For example, to create an array and initialize each element, you could use the following lines:

```
ARRAY POINTER(apPointers; 10)   ` Create an array to hold 10 pointers
For ($i; 1; 10)   ` Loop once for each variable
   apPointers{$i}:=Get pointer("Var"+String($i))   ` Initialize the array element
End for
```

The Get pointer function returns a pointer to the named object.

To reference any of the variables, you use the array elements. For example, to fill the variables with the next ten dates (assuming they are variables of the date type), you could use the following lines:

```
For ($i; 1; 10) ` Loop once for each variable
    apPointers{$i}->:=Current date+$i ` Assign the dates
End for
```

Setting a Button Using a Pointer

If you have a group of related radio buttons in a form, you often need to set them quickly. It is inefficient to directly reference each one of them by name. Let's say you have a group of radio buttons named Button1, Button2,..., Button5.

In a group of radio buttons, only one radio button is on. The number of the radio button that is on can be stored in a numeric field. For example, if the field called [Preferences]Setting contains 3, then Button3 is selected. In your form method, you could use the following code to set the button:

```
Case of
    :(Form event=On Load)
            ` ...
        Case of
            : ([Preferences]Setting = 1)
                Button1:=1
            : ([Preferences]Setting = 2)
                Button2:=1
            : ([Preferences]Setting = 3)
                Button3:=1
            : ([Preferences]Setting = 4)
                Button4:=1
            : ([Preferences]Setting = 5)
                Button5:=1
        End case
            ` ...
End case
```

A separate case must be tested for each radio button. This could be a very long method if you have many radio buttons in your form. Fortunately, you can use pointers to solve this problem. You can use the Get pointer command to return a pointer to a radio button. The following example uses such a pointer to reference the radio button that must be set. Here is the improved code:

```
Case of
   :(Form event=On Load)
         ` ...
      $vpRadio:=Get pointer("Button"+String([Preferences]Setting))
      $vpRadio->:=1
         ` ...
End case
```

The number of the set radio button must be stored in the field called [Preferences]Setting. You can do so in the form method for the On Clicked event:

```
[Preferences]Setting:=Button1+(Button2*2)+(Button3*3)+(Button4*4)+(Button5*5)
```

Passing Pointers to Methods

You can pass a pointer as a parameter to a method. Inside the method, you can modify the object referenced by the pointer. For example, the following method, TAKE TWO, takes two parameters that are pointers. It changes the object referenced by the first parameter to uppercase characters, and the object referenced by the second parameter to lowercase characters. Here is the method:

```
   ` TAKE TWO project method
   ` $1 – Pointer to a string field or variable. Change this to uppercase.
   ` $2 – Pointer to a string field or variable. Change this to lowercase.
$1->:=Uppercase($1->)
$2->:=Lowercase($2->)
```

The following line uses the TAKE TWO method to change a field to uppercase characters and to change a variable to lowercase characters:

```
TAKE TWO (->[My Table]My Field; ->MyVar)
```

If the field [My Table]My Field contained the string "jones", it would be changed to the string "JONES". If the variable MyVar contained the string "HELLO", it would be changed to the string "hello".

In the TAKE TWO method, and in fact, whenever you use pointers, it is important that the data type of the object being referenced is correct. In the previous example, the pointers must point to an object that contains a string or text.

Pointers to Pointers

If you really like to complicate things, you can use pointers to reference other pointers. Consider this example:

```
MyVar := "Hello"
PointerOne := ->MyVar
PointerTwo := ->PointerOne
(PointerTwo->)-> := "Goodbye"
ALERT((Point Two->)->)
```

It displays an alert box with the word “Goodbye” in it.

Here is an explanation of each line of the example:

- MyVar:="Hello"

→ This line puts the string "Hello" into the variable MyVar.

- PointerOne:=->MyVar

→ PointerOne now contains a pointer to MyVar.

- PointerTwo:=->PointerOne

→ PointerTwo (a new variable) contains a pointer to PointerOne, which in turn points to MyVar.

- (PointerTwo->)->:="Goodbye"

→ PointerTwo-> references the contents of PointerOne, which in turn references MyVar. Therefore (PointerTwo->)-> references the contents of MyVar. So in this case, MyVar is assigned "Goodbye".

- ALERT ((PointerTwo->)->)

→ Same thing: PointerTwo-> references the contents of PointerOne, which in turn references MyVar. Therefore (PointerTwo->)-> references the contents of MyVar. So in this case, the alert box displays the contents of myVar.

The following line puts "Hello" into MyVar:

```
(PointerTwo->)->:="Hello"
```

The following line gets "Hello" from MyVar and puts it into NewVar:

```
NewVar:=(PointerTwo->)->
```

Important: Multiple dereferencing requires parentheses.

See Also

Arrays, Arrays and Pointers, Constants, Control Flow, Data Types, Identifiers, Methods, Operators, Variables.

Identifiers

This section describes the conventions for naming various objects in the 4th Dimension language. The names for all objects follow these rules:

• A name must begin with an alphabetic character.
• Thereafter, the name can include alphabetic characters, numeric characters, the space character, and the underscore character.
• Periods, slashes, and colons are not allowed.
• Characters reserved for use as operators, such as * and +, are not allowed.
• 4th Dimension ignores any trailing spaces.

Tables

You denote a table by placing its name between brackets: [...]. A table name can contain up to 31 characters.

Examples

```
DEFAULT TABLE ([Orders])
INPUT FORM ([Clients]; "Entry")
ADD RECORD ([Letters])
```

Fields

You denote a field by first specifying the table to which the field belongs. The field name immediately follows the table name. A field name can contain up to 31 characters.

Do not start a field name with the underscore character (_). The underscore character is reserved for plug-ins. When 4th Dimension encounters this character at the beginning of a field in the Method editor, it removes the underscore.

Examples

```
[Orders]Total:=Sum([Line]Amount)
QUERY([Clients];[Clients]Name="Smith")
[Letters]Text:=Capitalize text ([Letters]Text)
```

It is a good programming technique to specify the table name before the field, even though it is not absolutely necessary in a table, form, or object method.

Subtables

You denote a subtable by first specifying the parent table to which the subtable belongs. The subtable name immediately follows the table name. A subtable name can contain up to 31 characters.

Examples

```
ALL SUBRECORDS ([People]Children)
ADD SUBRECORD ([Clients]Phones;"Add One")
NEXT SUBRECORD ([Letters]Keywords)
```

A subtable is treated as a type of field; therefore, it follows the same rules as a field when used in a form. If you are specifying a subtable in the table, form, or object method of the parent table, you do not need to specify the parent table name. However, it is a good programming technique to specify the name of the table before the subtable name.

Subfields

You denote a subfield in the same way as a field. You denote the subfield by first specifying the subtable to which the subfield belongs. The subfield name follows, and is separated from the subtable name by an apostrophe ('). A subfield name can contain up to 31 characters.

Examples

```
[People]Children'First Name:=Uppercase([People]Children'First Name)
[Clients]Phones'Number:="408 555–1212"
[Letters]Keywords'Word:=Capitalize text ([Letters]Keywords'Word)
```

If you are specifying a subfield in a subtable, form, or object method of the subfile, you do not need to specify the subtable name. However it is a good programming technique to specify the table name and the subtable name before the name of the subfield.

Interprocess Variables

You denote an interprocess variable by preceding the name of the variable with the symbols (<>) — a “less than” sign followed by a “greater than” sign.

Note: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess variable can have up to 31 characters, not including the <> symbols.

Examples

```
<>vlProcessID:=**Current process**
<>vsKey:=**Char**(KeyCode)
**If** (<>vtName#"")
```

Process Variables

You denote a process variable by using its name (which cannot start with the <> symbols nor the dollar sign $). A process variable name can contain up to 31 characters.

Examples

```
<>vrGrandTotal:=Sum([Accounts]Amount)
If (bValidate=1)
vsCurrentName:=""
```

Local Variables

You denote a local variable with a dollar sign ($) followed by its name. A local variable name can contain up to 31 characters, not including the dollar sign.

Examples

```
For ($vlRecord; 1; 100)
If ($vsTempVar="No")
$vsMyString:="Hello there"
```

Arrays

You denote an array by using its name, which is the name you passed to the array declaration (such as ARRAY LONGINT) when you created the array. Arrays are variables, and from the scope point of view, like variables, there are three different types of arrays:

• Interprocess arrays,
• Process arrays,
• Local arrays.

Interprocess Arrays
The name of an interprocess array is preceded by the symbols (<>) — a “less than” sign followed by a “greater than” sign.

Note: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess array name can contain up to 31 characters, not including the <> symbols.

Examples

```
ARRAY TEXT(<>atSubjects;Records in table([Topics]))
SORT ARRAY (<>asKeywords; >)
ARRAY INTEGER(<>aiBigArray;10000)
```

Process Arrays
You denote a process array by using its name (which cannot start with the <> symbols nor the dollar sign $). A process array name can contain up to 31 characters.

Examples

```
ARRAY TEXT(atSubjects;Records in table([Topics]))
SORT ARRAY (asKeywords; >)
ARRAY INTEGER(aiBigArray;10000)
```

Local Arrays
The name of a local array is preceded by the dollar sign ($). An local array name can contain up to 31 characters, not including the dollar sign.

Examples

```
ARRAY TEXT($atSubjects;Records in table([Topics]))
SORT ARRAY ($asKeywords; >)
ARRAY INTEGER($aiBigArray;10000)
```

Elements of arrays
You reference an element of an interprocess, process or local array by using the curly braces({…}). The element referenced is denoted by a numeric expression.

Examples

```
	` Adressing an element of an interprocess array
If (<>asKeywords{1}="Stop")
<>atSubjects{$vlElem}:=[Topics]Subject
$viNextValue:=<>aiBigArray{Size of array(<>aiBigArray)}

	` Adressing an element of a process array
If (asKeywords{1}="Stop")
atSubjects{$vlElem}:=[Topics]Subject
$viNextValue:=aiBigArray{Size of array(aiBigArray)}
```

```
    ` Adressing an element of a local array
If ($asKeywords{1}="Stop")
$atSubjects{$vlElem}:=[Topics]Subject
$viNextValue:=$aiBigArray{Size of array($aiBigArray)}
```

Elements of two-dimensional arrays
You reference an element of a two-dimensional array by using the curly braces ({…}) twice. The element referenced is denoted by two numeric expressions in two sets of curly braces.

Examples

```
    ` Adressing an element of a two-dimensional interprocess array
If (<>asKeywords{$vlNextRow}{1}="Stop")
<>atSubjects{10}{$vlElem}:=[Topics]Subject
$viNextValue:=<>aiBigArray{$vlSet}{Size of array(<>aiBigArray{$vlSet})}

    ` Adressing an element of a two-dimensional process array
If (asKeywords{$vlNextRow}{1}="Stop")
atSubjects{10}{$vlElem}:=[Topics]Subject
$viNextValue:=aiBigArray{$vlSet}{Size of array(aiBigArray{$vlSet})}

    ` Adressing an element of a two-dimensional local array
If ($asKeywords{$vlNextRow}{1}="Stop")
$atSubjects{10}{$vlElem}:=[Topics]Subject
$viNextValue:=$aiBigArray{$vlSet}{Size of array($aiBigArray{$vlSet})}
```

Forms

You denote a form by using a string expression that represents its name. A form name can contain up to 31 characters.

Examples

```
INPUT FORM([People];"Input")
OUTPUT FORM([People]; "Output")
DIALOG([Storage];"Note box"+String($vlStage))
```

Methods

You denote a method (procedure and function) by using its name. A method name can contain up to 31 characters.

Note: A method that does not return a result is also called a **procedure**. A method that returns is a result is also called a **function**.

Examples

```
If (New client)
DELETE DUPLICATED VALUES
APPLY TO SELECTION ([Employees];INCREASE SALARIES)
```

Tip: It is a good programming technique to adopt the same naming convention as the one used by 4D for built-in commands. Use uppercase characters for naming your methods; however if a method is function, capitalize the first character of its name. By doing so, when you reopen a database for maintenance after a few months, you will already know if a method returns a result by simply looking at its name in the Explorer window.

Note: When you call a method, you just type its name. However, some 4D built-in commands, such as ON EVENT CALL, as well as all the Plug-In commands, expect the name of a method as a string when a method parameter is passed. Example:

Examples

```
   ` This command expects a method (function) or formula
QUERY BY FORMULA ([aTable];Special query)
   ` This command expects a method (procedure) or statement
APPLY TO SELECTION ([Employees];INCREASE SALARIES)
   ` But this command expects a method name
ON EVENT CALL ("HANDLE EVENTS")
   ` And this Plug-In command expects a method name
WR ON ERROR ("WR HANDLE ERRORS")
```

Methods can accept parameters (arguments). The parameters are passed to the method in parentheses, following the name of the method. Each parameter is separated from the next by a semicolon (;). The parameters are available within the called method as consecutively numbered local variables: $1, $2,..., $n. In addition, multiple consecutive (and last) parameters can be addressed with the syntax ${n}where n, numeric expression, is the number of the parameter.

Inside a function, the $0 local variable contains the value to be returned.

Examples

```
     ` Within DROP SPACES $1 is a pointer the field [People]Name
DROP SPACES (->[People]Name)

     ` Within Calc creator:
     ` - $1 is numeric and equal to 1
     ` - $2 is numeric and equal to 5
     ` - $3 is text or string and equal to "Nice"
     ` - The result value is assigned to $0
$vsResult:=Calc creator (1; 5; "Nice")

     ` Within Dump:
     ` - The three parameters are text or string
     ` - They can be addressed as $1, $2 or $3
     ` - They can also be addressed as, for instance, ${$vlParam} where $vlParam is 1, 2 or
3
     ` - The result value is assigned to $0
vtClone:=Dump ("is"; "the"; "it")
```

Plug-In Commands (External Procedures, Functions and Areas)

You denote a plug-in command by using its name as defined by the plug-in. A plug-in command name can contain up to 31 characters.

Examples

```
WR BACKSPACE  (wrArea; 0)
$spNewArea:=SP New offscreen area
```

Sets

From the scope point of view, there are two types of sets:
• Interprocess sets
• Process sets

4D Server also includes:
• Client sets

Interprocess Sets
A set is an interprocess set if the name of the set is preceded symbols (<>) — a “less than” sign followed by a “greater than” sign.

Note: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess set name can contain up to 80 characters, not including the <> symbols.

Process Sets
You denote a process set by using a string expression that represents its name (which cannot start with the <> symbols or the dollar sign $). A set name can contain up to 80 characters.

Client Sets
The name of a client set is preceded by the dollar sign ($). A client set name can contain up to 80 characters, not including the dollar sign.

Note: In 4D Client/Server up to version 6, a set was maintained on the Client machine where it was created. Starting with version 6, sets are maintained on the Server machine. In certain cases, for efficiency or special purposes, you may need to work with sets locally on the Client machine. To do so, you use Client sets.

Examples
```
    ` Interprocess sets
USE SET("<>Deleted Records")
CREATE SET([Customers];"<>Customer Orders")
If (Records in set("<>Selection"+String($i))>0)
    ` Process sets
USE SET("Deleted Records")
CREATE SET([Customers];"Customer Orders")
If (Records in set("<>Selection"+String($i))>0)
    ` Client sets
USE SET("$Deleted Records")
CREATE SET([Customers];"$Customer Orders")
If (Records in set("$Selection"+String($i))>0)
```

Named Selections

From the scope point of view, there are two types of named selections:
• Interprocess named selections
• Process named selections

Interprocess Named Selections
A named selection is an interprocess named selection if its name is preceded by the symbols (<>) — a "less than" sign followed by a "greater than" sign.

Note: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

An interprocess named selection name can contain up to 80 characters, not including the <> symbols.

Process Named Selections
You denote a process named selection by using a string expression that represents its name (which cannot start with the <> symbols nor the dollar sign $). A named selection name can contain up to 80 characters.

Examples

```
	` Interprocess Named Selection
USE NAMED SELECTION([Customers];"<>ByZipcode")
	` Process Named Selection
USE NAMED SELECTION([Customers];"<>ByZipcode")
```

Processes

In the single-user version, or in Client/Server on the Client side, there are two types of processes:
• Global processes
• Local processes

Global Processes
You denote a global process by using a string expression that represents its name (which cannot start with the dollar sign $). A process name can contain up to 31 characters.

Local Processes
You denote a local process if the name of the process is preceded by a dollar ($) sign. The process name can contain up to 31 characters, not including the dollar sign.

Example

```
    ` Starting the global process "Add Customers"
$vlProcessID:=New process("P_ADD_CUSTOMERS";48*1024;"Add Customers")
    ` Starting the local process "$Follow Mouse Moves"
$vlProcessID:=New process("P_MOUSE_SNIFFER";16*1024;"$Follow Mouse Moves")
```

Summary of Naming Conventions

The following table summarizes 4th Dimension naming conventions.

Type	Max. Length	Example
Table	31	[Invoices]
Field	31	[Employees]Last Name
Subtable	31	[Friends]Kids
Subfield	31	[Documents]Keyword'Keyword
Interprocess Variable	<> + 31	<>vlNextProcessID
Process Variable	31	vsCurrentName
Local Variable	$ + 31	$vlLocalCounter
Form	31	"My Custom Web Input"
Interprocess Array	<> + 31	<>apTables
Process Array	31	asGender
Local Array	$ + 31	$atValues
Method	31	M_ADD_CUSTOMERS
Plug-in Routine	31	WR INSERT TEXT
Interprocess Set	<> + 80	"<>Records to be Archived"
Process Set	80	"Current selected records"
Client Set	$ + 80	"$Previous Subjects"
Named Selection	80	"Employees A to Z"
Interprocess Named Selection	<> + 80	"<>Employees Z to A"
Local Process	$ + 31	"$Follow Events"
Global Process	31	"P_INVOICES_MODULE"

Resolving Naming Conflicts

If a particular object has the same name as another object of a different type (for example, if a field is named Person and a variable is also named Person), 4th Dimension uses a priority system to identify the object. It is up to you to ensure that you use unique names for the parts of your database.

4th Dimension identifies names used in procedures in the following order:

1. Fields
2. Commands
3. Methods
4. Plug-in routines
5. Predefined constants
6. Variables

For example, 4th Dimension has a built-in command called Date. If you named a method Date, 4th Dimension would recognize it as the built-in Date command, and not as your method. This would prevent you from calling your method. If, however, you named a field “Date”, 4th Dimension would try to use your field instead of the Date command.

See Also

Arrays, Constants, Data Types, Methods, Operators, Pointers, Variables.

Regardless of the simplicity or complexity of a method, you will always use one or more of three types of programming structures. Programming structures control the flow of execution, whether and in what order statements are executed within a method. There are three types of structures:

• Sequential
• Branching
• Looping

The 4th Dimension language contains statements that control each of these structures.

Sequential structure

The sequential structure is a simple, linear structure. A sequence is a series of statements that 4th Dimension executes one after the other, from first to last. For example:

```
OUTPUT FORM([People]; "Listing")
ALL RECORDS([People])
DISPLAY SELECTION([People])
```

A one-line routine, frequently used for object methods, is the simplest case of a sequential structure. For example:

```
[People]Last Name:=Uppercase([People]Last Name)
```

Branching structures

A branching structure allows methods to test a condition and take alternative paths, depending on the result. The condition is a Boolean expression, an expression that evaluates TRUE or FALSE. One branching structure is the If...Else...End if structure, which directs program flow along one of two paths. The other branching structure is the Case of...Else...End case structure, which directs program flow to one of many paths.

Looping structures

When writing methods, it is very common to find that you need a sequence of statements to repeat a number of times. To deal with this need, the language provides three looping structures:

- While...End while
- Repeat...Until
- For...End for

The loops are controlled in two ways: either they loop until a condition is met, or they loop a specified number of times. Each looping structure can be used in either way, but While loops and Repeat loops are more appropriate for repeating until a condition is met, and For loops are more appropriate for looping a specified number of times.

See Also

Logical Operators, Methods.

The formal syntax of the If...Else...End if control flow structure is:

```
If (Boolean_Expression)
   statements(s)
Else
   statement(s)
End if
```

Note that the Else part is optional; you can write:

```
If (Boolean_Expression)
   statements(s)
End if
```

The If...Else...End if structure lets your method choose between two actions, depending on whether a test (a Boolean expression) is TRUE or FALSE.

When the Boolean expression is TRUE, the statements immediately following the test are executed. If the Boolean expression is FALSE, the statements following the Else statement are executed. The Else statement is optional; if you omit Else, execution continues with the first statement (if any) following the End if.

Example

```
   ` Ask the user to enter the name
$Find:=Request(“Type a name:”)
If (OK=1)
   QUERY([People]; [People]LastName=$Find)
Else
   ALERT("You did not enter a name.")
End if
```

Tip: Branching can be performed without statements to be executed in one case or the other. When developing an algorithm or a specialized application, nothing prevents you from writing:

```
If (Boolean_Expression)
Else
    statement(s)
End if
```

or:

```
If (Boolean_Expression)
    statements(s)
Else
End if
```

See Also

Case of...Else...End case, Control Flow, For...End for, Repeat...Until, While...End while.

The formal syntax of the Case of...Else...End case control flow structure is:

```
Case of
    : (Boolean_Expression)
        statement(s)
    : (Boolean_Expression)
        statement(s)
    .
    .
    .

    : (Boolean_Expression)
        statement(s)
Else
    statement(s)
End case
```

Note that the Else part is optional; you can write:

```
Case of
    : (Boolean_Expression)
        statement(s)
    : (Boolean_Expression)
        statement(s)
    .
    .
    .

    : (Boolean_Expression)
        statement(s)
End case
```

As with the If...Else...End if structure, the Case of...Else...End case structure also lets your method choose between alternative actions. Unlike the If...Else...End if structure, the Case of...Else...End case structure can test a reasonable unlimited number of Boolean expressions and take action depending on which one is TRUE.

Each Boolean expression is prefaced by a colon (:). This combination of the colon and the Boolean expression is called a case. For example, the following line is a case:

```
: (bValidate=1)
```

Only the statements following the first TRUE case (and up to the next case) will be executed. If none of the cases are TRUE, none of the statements will be executed (if no Else part is included).

You can include an Else statement after the last case. If all of the cases are FALSE, the statements following the Else will be executed.

Example

This example tests a numeric variable and displays an alert box with a word in it:

```
Case of
   : (vResult = 1)   ` Test if the number is 1
      ALERT("One.")   ` If it is 1, display an alert
   : (vResult = 2)   ` Test if the number is 2
      ALERT("Two.")   ` If it is 2, display an alert
   : (vResult = 3) ` Test if the number is 3
      ALERT("Three.")   ` If it is 3, display an alert
Else   ` If it is not 1, 2, or 3, display an alert
   ALERT("It was not one, two, or three.")
End case
```

For comparison, here is the If...Else...End if version of the same method:

```
If (vResult = 1)   ` Test if the number is 1
   ALERT("One.")   ` If it is 1, display an alert
Else
   If (vResult = 2)   ` Test if the number is 2
      ALERT("Two.")   ` If it is 2, display an alert
   Else
      If (vResult = 3)   ` Test if the number is 3
         ALERT("Three.")   ` If it is 3, display an alert
      Else   ` If it is not 1, 2, or 3, display an alert
         ALERT("It was not one, two, or three.")
      End if
   End if
End if
```

Remember that with a Case of...Else...End case structure, only the first TRUE case is executed. Even if two or more cases are TRUE, only the statements following the first TRUE case will be executed.

Tip: Branching can be performed without statements to be executed in one case or another. When developing an algorithm or a specialized application, nothing prevents you from writing:

```
Case of
    : (Boolean_Expression)
    : (Boolean_Expression)

    .
    .
    .

    : (Boolean_Expression)
        statement(s)
Else
    statement(s)
End case
```

or:

```
Case of
    : (Boolean_Expression)
    : (Boolean_Expression)
        statement(s)
    .
    .
    .

    : (Boolean_Expression)
        statement(s)
Else
End case
```

or:

```
Case of
Else
    statement(s)
End case
```

See Also

Control Flow, For...End for, If...Else...End if, Repeat...Until, While...End while.

The formal syntax of the While...End while control flow structure is:

```
While (Boolean_Expression)
   statement(s)
End while
```

A While...End while loop executes the statements inside the loop as long as the Boolean expression is TRUE. It tests the Boolean expression at the beginning of the loop and does not enter the loop at all if the expression is FALSE.

It is common to initialize the value tested in the Boolean expression immediately before entering the While...End while loop. Initializing the value means setting it to something appropriate, usually so that the Boolean expression will be TRUE and While...End while executes the loop.

The Boolean expression must be set by something inside the loop or else the loop will continue forever. The following loop continues forever because NeverStop is always TRUE:

```
NeverStop:=True
While (NeverStop)
End while
```

If you find yourself in such a situation, where a method is executing uncontrolled, you can use the trace facilities to stop the loop and track down the problem. For more information about tracing a method, see the section Debugging.

Example

```
CONFIRM ("Add a new record?")   ` The user wants to add a record?
While (OK = 1)   ` Loop as long as the user wants to
   ADD RECORD([aTable])   ` Add a new record
End while   ` The loop always ends with End while
```

In this example, the OK system variable is set by the CONFIRM command before the loop starts. If the user clicks the OK button in the confirmation dialog box, the OK system variable is set to 1 and the loop starts. Otherwise, the OK system variable is set to 0 and the loop is skipped. Once the loop starts, the ADD RECORD command keeps the loop going because it sets the OK system variable to 1 when the user saves the record. When the user cancels (does not save) the last record, the OK system variable is set to 0 and the loop stops.

See Also

Case of...Else...End case, Control Flow, For...End for, If...Else...End if, Repeat...Until.

The formal syntax of the Repeat...Until control flow structure is:

```
Repeat
    statement(s)
Until (Boolean_Expression)
```

A Repeat...Until loop is similar to a While...End while loop, except that it tests the Boolean expression after the loop rather than before. Thus, a Repeat...Until loop always executes the loop once, whereas if the Boolean expression is initially False, a While...End while loop does not execute the loop at all.

The other difference with a Repeat...Until loop is that the loop continues until the Boolean expression is TRUE.

Example

Compare the following example with the example for the While...End while loop. Note that the Boolean expression does not need to be initialized—there is no CONFIRM command to initialize the OK variable.

```
Repeat
    ADD RECORD([aTable])
Until (OK=0)
```

See Also

Case of...Else...End case, Control Flow, For...End for, If...Else...End if, While...End while.

The formal syntax of the For...End for control flow structure is:

```
For (Counter_Variable; Start_Expression; End_Expression {; Increment_Expression})
    statement(s)
End for
```

The For...End for loop is a loop controlled by a counter variable:

• The counter variable Counter_Variable is a numeric variable (Real, Integer, or Long Integer) that the For...End for loop initializes to the value specified by Start_Expression.

• Each time the loop is executed, the counter variable is incremented by the value specified in the optional value Increment_Expression. If you do not specify Increment_Expression, the counter variable is incremented by one (1), which is the default.

• When the counter variable passes the End_Expression value, the loop stops.

Important: The numeric expressions Start_Expression, End_Expression and Increment_Expression are evaluated once at the beginning of the loop. If these expressions are variables, changing one of these variables **within** the loop **will not** affect the loop.

Tip: However, for special purposes, you can change the value of the counter variable Counter_Variable **within** the loop; this **will** affect the loop.

• Usually Start_Expression is less than End_Expression.

• If Start_Expression and End_Expression are equal, the loop will execute only once.

• If Start_Expression is greater than End_Expression, the loop will not execute at all unless you specify a negative Increment_Expression. See the examples.

Basic Examples

1. The following example executes 100 iterations:

```
For (vCounter;1;100)
    ` Do something
End for
```

2. The following example goes through all elements of the array anArray:

```
For ($vlElem;1;Size of array(anArray))
        ` Do something with the element
    anArray{$vlElem}:=...
End for
```

3. The following example goes through all the characters of the text vtSomeText:

```
For ($vlChar;1;Length(vtSomeText))
        ` Do something with the character if it is a TAB
    If (Ascii(vtSomeText≤$vlChar≥)=Char(Tab))
        ` ...
    End if
End for
```

4. The following example goes through the selected records for the table [aTable]:

```
FIRST RECORD([aTable])
For ($vlRecord;1;Records in selection([aTable]))
        ` Do something with the record
    SEND RECORD([aTable])
        ` ...
        ` Go to the next record
    NEXT RECORD([aTable])
End for
```

Most of the For...End for loops you will write in your databases will look like the ones listed in these examples.

Decrementing variable counter

In some cases, you may want to have a loop whose counter variable is decreasing rather than increasing. To do so, you must specify Start_Expression greater than End_Expression and a negative Increment_Expression. The following examples do the same thing as the previous examples, but in reverse order:

5. The following example executes 100 iterations:

```
For (vCounter;100;1;-1)
    ` Do something
End for
```

6. The following example goes through all elements of the array anArray:

```
For ($vlElem;Size of array(anArray);1;-1)
        ` Do something with the element
    anArray{$vlElem}:=...
End for
```

7. The following example goes through all the characters of the text vtSomeText:

```
For ($vlChar;Length(vtSomeText);1;-1)
      ` Do something with the character if it is a TAB
   If (Ascii(vtSomeText≤$vlChar≥)=Char(Tab))
      ` ...
   End if
End for
```

8. The following example goes through the selected records for the table [aTable]:

```
LAST RECORD([aTable])
For ($vlRecord;Records in selection([aTable]);1;-1)
      ` Do something with the record
   SEND RECORD([aTable])
      ` ...
      ` Go to the previous record
   PREVIOUS RECORD([aTable])
End for
```

Incrementing the counter variable by more than one

If you need to, you can use an Increment_Expression (positive or negative) whose absolute value is greater than one.

9. The following loop addresses only the even elements of the array anArray:

```
For ($vlElem;2;((Size of array(anArray)+1)\2)*2;2)
      ` Do something with the element #2,#4...#2n
   anArray{$vlElem}:=...
End for
```

Note that the ending expression ((Size of array(anArray)+1)\2)*2 takes care of even and odd array sizes.

Getting out of a loop by changing the counter variable

In some cases, you may want to execute a loop for a specific number of iterations, but then get out of the loop when another condition becomes TRUE. To do so, you can test this condition within the loop and if it becomes TRUE, explicitly set the counter variable to a value that exceeds the end expression.

10. In the following example, a selection of the records is browsed until this is actually done or until the interprocess variable <>vbWeStop, intially set to FALSE, becomes TRUE. This variable is handled by an ON EVENT CALL project method that allows you to interrupt the operation:

```
	<>vbWeStop:=False
	ON EVENT CALL ("HANDLE STOP")
	   ` HANDLE STOP sets <>vbWeStop to True if Ctrl-period (Windows) or Cmd-Period
(Macintosh) is pressed
	$vlNbRecords:=Records in selection([aTable])
	FIRST RECORD([aTable])
	For ($vlRecord;1;$vlNbRecords)
	      ` Do something with the record
	   SEND RECORD([aTable])
	      ` ...
	      ` Go to the next record
	   If (<>vbWeStop)
	      $vlRecord:=$vlNbRecords+1 ` Force the counter variable to get out of the loop
	   Else
	      NEXT RECORD([aTable])
	   End if
	End for
	ON EVENT CALL("")
	If (<>vbWeStop)
	   ALERT("The operation has been interrupted.")
	Else
	   ALERT("The operation has been successfully completed.")
	End if
```

Comparing looping structures

Let's go back to the first For...End for example:

The following example executes 100 iterations:

```
For (vCounter;1;100)
    ` Do something
End for
```

It is interesting to see how the While...End while loop and Repeat...Until loop would perform the same action.

Here is the equivalent While...End while loop:

```
$i := 1 ` Initialize the counter
While ($i<=100) ` Loop 100 times
    ` Do something
    $i := $i + 1 ` Need to increment the counter
End while
```

Here is the equivalent Repeat...Until loop:

```
$i := 1 ` Initialize the counter
Repeat
    ` Do something
    $i := $i + 1 ` Need to increment the counter
Until ($i=100) ` Loop 100 times
```

Tip: The For...End for loop is usually faster than the While...End while and Repeat...Until loops, because 4th Dimension tests the condition internally for each cycle of the loop and increments the counter. Therefore, use the For...End for loop whenever possible.

Optimizing the execution of the For...End for loops

You can use Real, Integer, and Long Integer variables as well as interprocess, process, and local variable counters. For lengthy repetitive loops, especially in compiled mode, use local Long Integer variables.

11. Here is an example:

```
C_LONGINT($vlCounter)    ` use local Long Integer variables
For ($vlCounter;1;10000)
    ` Do something
End for
```

Nested For...End for **looping structures**

You can nest as many control structures as you (reasonably) need. This includes nesting For...End for loops. To avoid mistakes, make sure to use different counter variables for each looping structure.

Here are two examples:

12. The following example goes through all the elements of a two-dimensional array:

```
For ($vlElem;1;Size of array(anArray))
     ` ...
     ` Do something with the row
     ` ...
   For ($vlSubElem;1;Size of array(anArray{$vlElem}))
     ` Do something with the element
     anArray{$vlElem}{$vlSubElem}:=...
   End for
End for
```

13. The following example builds an array of pointers to all the date fields present in the database:

```
ARRAY POINTER($apDateFields;0)
$vlElem:=0
For ($vlTable;1;Count table)
   For($vlField;1;Count fields($vlTable))
      $vpField:=Field($vlTable;$vlField)
      If (Type($vpField->)=Is Date)
         $vlElem:=$vlElem+1
         INSERT ELEMENT($apDateFields;$vlElem)
         $apDateFields{$vlElem}:=$vpField
      End if
   End for
End for
```

See Also

Case of...Else...End case, Control Flow, If...Else...End if, Repeat...Until, While...End while.

Methods

Language Definition

version 6.0

In order to make the commands, operators, and other parts of the language work, you put them in methods. There are several kinds of methods: Object methods, Form methods, Table methods (Triggers), Project methods, and Database methods. This section describes features common to all types of methods.

A method is composed of **statements**; each statement consists of one line in the method. A statement performs an action, and may be simple or complex. Although a statement is always one line, that one line can be as long as needed (up to 32,000 characters, which is probably enough for most tasks).

For example, the following line is a statement that will add a new record to the [People] table:

```
ADD RECORD([People])
```

A method also contains **tests** and **loops** that control the flow of the execution. For a detailed discussion about the control flow programming structures, see the section Control Flow.

Types of Methods

There are five types of methods in 4th Dimension:

• **Object methods**: An object method is a property of an object. It is a usually a short method associated with an active form object. Object methods generally “manage” the object while the form is displayed or printed. You do not call an object method—4D calls it automatically when an event involves the object to which the object method is attached.

• **Form methods**: A form method is a property of a form. You can use a form method to manage data and objects, but it is generally simpler and more efficient to use an object method for these purposes. You do not call a form method—4D calls it automatically when an event involves the form to which the form method is attached.

For more information about Object methods and Form methods, see the 4th Dimension Design Reference Manual as well as the section Form event.

• **Table methods (Triggers)**: A Trigger is a property of a table. You do not call a Trigger. Triggers are automatically invoked by the 4D database engine each time that you manipulate the records of a table (Add, Delete, Modify and Load). Triggers are methods that can prevent “illegal” operations with the records of your database. For example, in an invoicing system, you can prevent anyone from adding an invoice without specifying the customer to whom the invoice is billed. Triggers are a very powerful tool to restrict operations on a table, as well as to prevent accidental data loss or tampering. You can write very simple triggers, and then make them more and more sophisticated.

For detailed information about Triggers, see the section Triggers.

• **Project methods**: Unlike object methods, form methods, and triggers, which are all associated with a particular object, form, or table, project methods are available for use throughout your database. Project methods are reusable, and available for use by any other method. If you need to repeat a task, you do not have to write identical methods for each case. You can call project methods wherever you need them—from other project methods or from object or form methods. When you call a project method, it acts as if you had written the method at the location where you called it. Project methods called from other method are often referred to as “subroutines.” A project method that returns a result can also be called a **function**.

There is one other way to use project methods—associating them with menu commands. When you associate a project method with a menu command, the method is executed when the menu is chosen. You can think of the menu command as calling the project method.

For detailed information about Project methods, see the section Project Methods.

• **Database methods**: In the same way object and form methods are invoked when events occur in a form, there are methods associated with the database that are invoked when a working session event occurs. These are the **database methods**. For example, each time you open a database, you may want to initialize some variables that will be used during the whole working session. To do so, you use the On Startup Database Method, automatically executed by 4D when you open the database.

For more information about Database Methods, see the section Database Methods.

Compatiblity with previous versions of 4D

You can skip these compatibility notes if you work with brand-new databases created with version 6 of 4th Dimension.

1. Version 6 introduces many new object and form events (such as On Double Clicked, On Getting Focus, and so on) that replace the execution cycles from the previous versions. If you have converted a version 3 database to version 6, your forms have been converted in order to preserve as much as the “expected behavior” of your forms and objects.

If you want to take advantage of the new events for forms and objects created with a previous version of 4D, you must enable the new events in the **Form Properties** and **Object Properties** windows for the forms and the objects.

2. Table method, also called trigger, is a new type of method introduced in version 6. In previous versions of 4th Dimension, table methods (called file procedures) were executed by 4D only when a form for a table was used for data entry, display, or printing. They were rarely used. Note that triggers execute at a much lower level that the old file procedures. No matter what you do to a record via user actions (like data entry) or programmatically (like a call to SAVE RECORD), the trigger of a table will be invoked by 4D. Triggers are truly quite different from the old file procedures. If you have converted a version 3 database to version 6, and if you want to take advantage of the new Trigger capability, you must deselect the **Use Old File Procedures Scheme** property in the **Database Properties** dialog box (shown in this section).

3. Database methods is a new type of method introduced in version 6. In previous versions of 4th Dimension, there was only one method (procedure) that 4D automatically executed when you opened a database. This procedure had to be called **STARTUP** (US English INTL version) or **DEBUT** (French version) in order to be invoked. If you have converted a version 3 database to version 6, and if you want to take advantage of the new On Startup Database Method capability, you must deselect the **Use Old Startup Method** property in the **Database Properties** dialog box (shown in this section). This property only affects the STARTUP/On Startup Database Method alternative. If you do not deselect this property and add, for instance, an On Exit Database Method, this latter will be invoked by 4D.

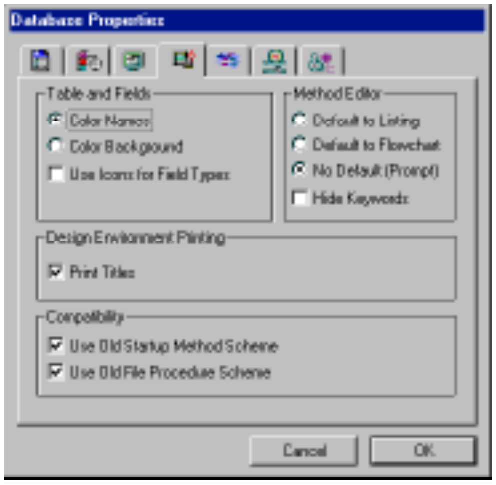

Database Properties

Table and Fields
[✓ Color Names]
Color Background
☐ Use Icons for Field Types

Method Editor
Default to Listing
Default to Flowchart
[✓ No Default (Prompt)]
☐ Hide Keywords

Design Environment Printing
☑ Print Titles

Compatibility
☑ Use Old Startup Method Scheme
☑ Use Old File Procedure Scheme

An Example Project Method

All methods are fundamentally the same—they start at the first line and work their way through each statement until they reach the last line (i.e., they execute sequentially). Here is an example project method:

```
QUERY ([People])   ` Display the Query editor
If (OK=1)   ` The user clicked OK, not cancel
    If (Records in selection([People])=0)   ` If no record was found…
        ADD RECORD([People])   ` Let the user add a new record
    End if
End if ` The end
```

Each line in the example is a **statement** or line of code. Anything that you write using the language is loosely referred to as code. Code is executed or run; this means that 4th Dimension performs the task specified by the code.

We will examine the first line in detail and then move on more quickly:

```
QUERY([People]) ` Display the Query editor
```

The first element in the line, QUERY, is a command. A command is part of the 4th Dimension language—it performs a task. In this case, QUERY displays the Query editor. This is similar to choosing Query from the Queries menu in the User environment.

The second element in the line, specified between parantheses, is an argument to the QUERY command. An argument (or **parameter**) is data required by a command in order to complete its task. In this case, [People] is the name of a table. Table names are always specified inside square brackets ([…]). In our example, the People table is an argument to the QUERY command. A command can accept several parameters.

The third element is a **comment** at the end of the line. A comment tells you (and anyone else who might read your code) what is happening in the code. It is indicated by the reverse apostrophe (`). Anything (on the line) following the comment mark will be ignored when the code is run. A comment can be put on a line by itself, or you can put comments to the right of the code, as in the example. Use comments generously throughout your code; this makes it easier for you and others to read and understand the code.

Note: A comment can be up to 80 characters long.

The next line of the method checks to see if any records were found:

```
	If (Records in selection([People]) = 0) ` If no record was found…
```

The If statement is a **control-of-flow statement**—a statement that controls the step-by-step execution of your method. The If statement performs a test, and if the statement is true, execution continues with the subsequent lines. Records in selection is a function—a command that returns a value. Here, Records in selection returns the number of records in the current selection for the table passed as argument.

Note: Notice that only the first letter of the function name is capitalized. This is the naming convention for 4th Dimension functions.

You should already know what the current selection is—it is the group of records you are working on at any one time. If the number of records is equal to 0 (in other words, if no record was found), then the following line is executed:

```
	ADD RECORD([People]) ` Let the user add a new record
```

The ADD RECORD command displays a form so that the user can add a new record. 4th Dimension formats your code automatically; notice that this line is indented to show you that it is dependent on the control-of-flow statement (If).

```
	End if  ` The end
```

The End if statement concludes the If statement's section of control. Whenever there is a control-of-flow statement, you need to have a corresponding statement telling the language where the control stops.

Be sure you feel comfortable with the concepts in this section. If they are all new, you may want to review them until they are clear to you.

Where to go from here?

To learn more about:
• Object methods and Form methods, see the section Form event.
• Triggers, see the section Triggers.
• Project methods, see the section Project Methods.
• Database methods, see the section Database Methods.

See Also

Arrays, Constants, Control Flow, Data Types, Database Methods, Identifiers, Operators, Pointers, Triggers, Variables.

Project Methods

Project methods are aptly named. Whereas form and object methods are bound to forms and objects, a project method is available anywhere; it is not specifically attached to any particular object of the database. A project method can have one of the following roles, depending on how it is executed and used:

• Menu method
• Subroutine and function
• Process method
• Event catching method
• Error catching method

These terms do not distinguish project methods by what they are, but by what they do.

A **menu method** is a project method called from a custom menu. It directs the flow of your application. The menu method takes control—branching where needed, presenting forms, generating reports, and generally managing your database.

The **subroutine** is a project method that can be thought of as a servant. It performs those tasks that other methods request it to perform. A function is a subroutine that returns a value to the method that called it.

A **process method** is a project method that is called when a process is started. The process lasts only as long as the process method continues to execute. For more information about processes, see the section Processes. Note that a menu method attached to a menu command whose property **Start a New Process** is selected, is also the process method for the newly started process.

An **event catching method** runs in a separate process as the process method for catching events. Usually, you let 4D do most of the event handling for you. For example, during data entry, 4D detects keystrokes and clicks, then calls the correct object and form methods so you can respond appropriately to the events from within these methods. In other circumstances, you may want to handle events directly. For example, if you run a lengthy operation (such as For...End For loop browsing records), you may want to be able to interrupt the operation by typing Ctrl-Period (Windows) or Cmd-Period (Macintosh). In this case, you should use an event catching method to do so. For more information, see the description of the command ON EVENT CALL.

An **error catching method** is an interrupt-based project method. Each time an error or an exception occurs, it executes within the process in which it was installed. For more information, see the description of the command ON ERR CALL.

Menu Methods

A menu method is invoked in the Custom Menus environment when you select the custom menu command to which it is attached. You assign the method to the menu command using the Menu editor. The menu executes when the menu command is chosen. This process is one of the major aspects of customizing a database. By creating custom menus with menu methods that perform specific actions, you personalize your database. Refer to the *4th Dimension Design Reference* manual for more information about the Menu editor.

Custom menu commands can cause one or more activities to take place. For example, a menu command for entering records might call a method that performs two tasks: displaying the appropriate input form, and calling the ADD RECORD command until the user cancels the data entry activity.

Automating sequences of activities is a very powerful capability of the programming language. Using custom menus, you can automate task sequences that would otherwise be carried out manually in the User environment. With custom menus, you provide more guidance to users of the database.

Subroutines

When you create a project method, it becomes part of the language of the database in which you create it. You can then call the project method in the same way that you call 4th Dimension's built-in commands. A project method used in this way is called a subroutine.

You use subroutines to:
• Reduce repetitive coding
• Clarify your methods
• Facilitate changes to your methods
• Modularize your code

For example, let's say you have a database of customers. As you customize the database, you find that there are some tasks that you perform repeatedly, such as finding a customer and modifying his or her record. The code to do this might look like this:

```
   ` Look for a customer
QUERY BY EXAMPLE([Customers])
   ` Select the input form
INPUT FORM([Customers];"Data Entry")
   ` Modify the customer's record
MODIFY RECORD([Customers])
```

If you do not use subroutines, you will have to write the code each time you want to modify a customer's record. If there are ten places in your custom database where you need to do this, you will have to write the code ten times. If you use subroutines, you will only have to write it once. This is the first advantage of subroutines—to reduce the amount of code.

If the previously described code was a method called MODIFY CUSTOMER, you would execute it simply by using the name of the method in another method. For example, to modify a customer's record and then print the record, you would write this method:

```
MODIFY CUSTOMER
PRINT SELECTION([Customers])
```

This capability simplifies your methods dramatically. In the example, you do not need to know how the MODIFY CUSTOMER method works, just what it does. This is the second reason for using subroutines—to clarify your methods. In this way, your methods become extensions to the 4th Dimension language.

If you need to change your method of finding customers in this example database, you will need to change only one method, not ten. This is the next reason to use subroutines—to facilitate changes to your methods.

Using subroutines, you make your code modular. This simply means dividing your code into modules (subroutines), each of which performs a logical task. Consider the following code from a checking account database:

```
FIND CLEARED CHECKS   ` Find the cleared checks
RECONCILE ACCOUNT   ` Reconcile the account
PRINT CHECK BOOK REPORT   ` Print a checkbook report
```

Even for someone who doesn't know the database, it is clear what this code does. It is not necessary to examine each subroutine. Each subroutine might be many lines long and perform some complex operations, but here it is only important that it performs its task.

We recommend that you divide your code into logical tasks, or modules, whenever possible.

Passing Parameters to Methods

You'll often find that you need to pass data to your methods. This is easily done with parameters.

Parameters (or arguments) are pieces of data that a method needs in order to perform its task. The terms parameter and argument are used interchangeably throughout this manual. Parameters are also passed to built-in 4th Dimension commands. In this example, the string "Hello" is an argument to the ALERT command:

```
ALERT("Hello")
```

Parameters are passed to methods in the same way. For example, if a method named DO SOMETHING accepted three parameters, a call to the method might look like this:

```
DO SOMETHING(WithThis;AndThat;ThisWay)
```

The parameters are separated by semicolons (;).

In the subroutine (the method that is called), the value of each parameter is automatically copied into sequentially numbered local variables: $1, $2, $3, and so on. The numbering of the local variables represents the order of the parameters.

The local variables/parameters are not the actual fields, variables, or expressions passed by the **calling method**; they only contain the values that have been passed.

Within the subroutine, you can use the parameters $1, $2... in the same way you would use any other local variable.

Since they are local variables, they are available only within the subroutine and are cleared at the end of the subroutine. For this reason, a subroutine cannot change the value of the actual fields or variables passed as parameters at the calling method level. For example:

```
   ` Here is some code from the method MY METHOD
   ` ...
DO SOMETHING ([People]Last Name) ` Let's say [People]Last Name is equal to "williams"
ALERT([People]Last Name)

   ` Here the code of the method DO SOMETHING
$1:=Uppercase($1)
ALERT($1)
```

The alert box displayed by DO SOMETHING will read "WILLIAMS" and the alert box displayed by MY METHOD will read "williams". The method locally changed the value of the parameter $1, but this does not affect the value of the field [People]Last Name passed as parameter by the method MY METHOD.

There are two ways to make the method DO SOMETHING change the value of the field:

1. Rather than passing the field to the method, you pass a pointer to it, so you would write:

```
    ` Here is some code from the method MY METHOD
    ` ...
    ` Let's say [People]Last Name is equal to "williams"
DO SOMETHING (->[People]Last Name)
ALERT([People]Last Name)

    ` Here the code of the method DO SOMETHING
$1->:=Uppercase($1->)
ALERT($1->)
```

Here the parameter is not the field, but a pointer to it. Therefore, within the DO SOMETHING method, $1 is no longer the value of the field but a pointer to the field. The object **referenced** by $1 ($1-> in the code above) is the actual field. Consequently, changing the referenced object goes beyond the scope of the subroutine, and the actual field is affected. In this example, both alert boxes will read "WILLIAMS".

For more information about Pointers, see the section Pointers.

2. Rather than having the method DO SOMETHING "doing something," you can rewrite the method so it returns a value. Thus you would write:

```
    ` Here is some code from the method MY METHOD
    ` ...
    ` Let's say [People]Last Name is equal to "williams"
[People]Last Name:=DO SOMETHING ([People]Last Name)
ALERT([People]Last Name)

    ` Here the code of the method DO SOMETHING
$0:=$1
ALERT($0)
```

This second technique of returning a value by a subroutine is called "using a function." This is described in the next paragraphs.

Advanced note: Parameters within the subroutine are accessible through the local variables $1, $2... In addition, parameters can be optional and can be referred to using the syntax ${...}. For more information on parameters, see the description of the command Count parameters.

Functions: Project Methods that return a value

Data can be returned from methods. A method that returns a value is called a **function**.

4D or 4D Plug-in commands that return a value are also called functions.

For example, the following line is a statement that uses the built-in function, Length, to return the length of a string. The statement puts the value returned by Length in a variable called MyLength. Here is the statement:

```
MyLength:=Length("How did I get here?")
```

Any subroutine can return a value. The value to be returned is put into the local variable $0.

For example, the following function, called Uppercase4, returns a string with the first four characters of the string passed to it in uppercase:

```
$0:=Uppercase(Substring($1; 1; 4))+Substring($1; 5)
```

The following is an example that uses the Uppercase4 function:

```
NewPhrase:=Uppercase4 ("This is good.")
```

In this example, the variable NewPhrase gets “THIS is good.”

The **function result**, $0, is a local variable within the subroutine. It can be used as such within the subroutine. For example, in the previous DO SOMETHING example, $0 was first assigned the value of $1, then used as parameter to the ALERT command. Within the subroutine, you can use $0 in the same way you would use any other local variable. It is 4D that returns the value of $0 (as it is when the subroutine ends) to the called method.

Recursive Project Methods

Project methods can call themselves. For example:
• The method A may call the method B which may call A, so A will call B again and so on.
• A method can call itself.

This is called **recursivity**. The 4D language fully supports recursivity.

Here is an example. Let’s say you have a [Friends and Relatives] table composed of this extremely simplified set of fields:
- [Friends and Relatives]Name
- [Friends and Relatives]Children'Name

For this example, we assume the values in the fields are unique (there are no two persons with the same name). Given a name, you want to build the sentence “A friend of mine, John who is the child of Paul who is the child of Jane who is the child of Robert who is the child of Eleanor, does this for a living!”:

1. You can build the sentence in this way:

```
$vsName:=Request("Enter the name:";"John")
If (OK=1)
    QUERY([Friends and Relatives];[Friends and Relatives]Name=$vsName)
    If (Records in selection([Friends and Relatives])>0)
        $vtTheWholeStory:="A friend of mine, "+$vsName
        Repeat
            QUERY([Friends and Relatives];[Friends and Relatives]Children'Name=$vsName)
            $vlQueryResult:=Records in selection([Friends and Relatives])
            If ($vlQueryResult>0)
                $vtTheWholeStory:=$vtTheWholeStory+" who is the child of "
                                                              +[Friends and Relatives]Name
                $vsName:=[Friends and Relatives]Name
            End if
        Until ($vlQueryResult=0)
        $vtTheWholeStory:=$vtTheWholeStory+", does this for a living!"
        ALERT($vtTheWholeStory)
    End if
End if
```

2. You can also build it this way:

```
$vsName:=Request("Enter the name:";"John")
If (OK=1)
    QUERY([Friends and Relatives];[Friends and Relatives]Name=$vsName)
    If (Records in selection([Friends and Relatives])>0)
        ALERT("A friend of mine, "+Genealogy of ($vsName)+", does this for a living!")
    End if
End if
```

with the recursive function Genealogy of listed here:

```
    ` Genealogy of project method
    ` Genealogy of ( String ) -> Text
    ` Genealogy of ( Name ) -> Part of sentence

$0:=$1
QUERY([Friends and Relatives];[Friends and Relatives]Children'Name=$1)
If (Records in selection([Friends and Relatives])>0)
    $0:=$0+" who is the child of "+Genealogy of ([Friends and Relatives]Name)
End if
```

Note the Genealogy of method which calls itself.

The first way is an **iterative algorithm**. The second way is a **recursive algorithm**.

When implementing code for cases like the previous example, it is important to note that you can always write methods using iteration or recursivity. Typically, recursivity provides more concise, readable, and maintainable code, but using it is not mandatory.

Some typical uses of recursivity in 4D are:
• Treating records within tables that relate to each other in the same way as in the example.
• Browsing documents and folders on your disk, using the commands FOLDER LIST and DOCUMENT LIST. A folder may contain folders and documents, the subfolders can themselves contain folders and documents, and so on.

Important: Recursive calls should always end at some point. In the example, the method Genealogy of stops calling itself when the query returns no records. Without this condition test, the method would call itself indefinitely; eventually, 4D would return a "Stack Full" error becuase it would no longer have space to "pile up" the calls (as well as parameters and local variables used in the method).

See Also

Control Flow, Database Methods, Methods.

3 4D Environment

Application type

version 6.0

Application type → Long Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Long Integer	←	Numeric value denoting the type of the application

Description

The Application type command returns a numeric value that denotes the type of 4D environment that you are running. 4D provides the following predefined constants:

Constant	Type	Value
4th Dimension	Long Integer	0
4D Engine	Long Integer	1
4D Runtime	Long Integer	2
4D Runtime Classic	Long Integer	3
4D Client	Long Integer	4
4D Server	Long Integer	5
4D First	Long Integer	6

Example

Somewhere in your code, other than in the On Server Startup database method, you need to check if you are running 4D Server. You can write:

```
⇒    If (Application type=4D Server)
        ` Perform appropriate actions
     End if
```

See Also

Application version, Version type.

Version type

Version type → Long Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Long Integer	←	0 -> Full version 1 -> Demo Limited version

Description

The Version type command returns a numeric value that denotes the type of 4D environment version that you are running. 4D provides the following predefined constants:

Constant	Type	Value
Full Version	Long Integer	0
Demo Version	Long Integer	1

Example

Your 4D application includes some features that are not available when a demo version of the 4D environment is used. Surround these features with a test that calls Version type:

```
⇒       If (Version type=Full Version)
           ` Perform appropriate operations
        Else
           ALERT("This feature is not available in the Demo version of"
                                                       +" Super Management Systems™.")
        End if
```

See Also

Application type, Application version.

Application version

Application version {(*)} → String

Parameter	Type		Description
*	*	→	Long version number if passed, otherwise Short version number
Function result	String	←	Version number encoded string

Description

The Application version command returns an encoded string value that expresses the version number of the 4D environment you are running.

• If you do not pass the optional * parameter, a 4-character string is returned, formatted as follows:

Characters	Description
1-2	Version number
3	Update number
4	Revision number

Example: The string "0600" stands for version 6.0.0.

• If you pass the optional * parameter, an 8-character string is returned, formatted as follows:

Characters	Description
1	"F" denotes a final version "B" denotes a beta version Other characters denote an ACI internal version
2-3-4	Internal ACI compilation number
5-6	Version number
7	Update number
8	Revision number

Example: The string "B0120602" would stand for the Beta 12 of version 6.0.2.

Examples

1. This example displays the 4D environment version number:

```
⇒      $vs4Dversion:=Application version
       ALERT("You are using the version "+String(Num(Substring($vs4Dversion;1;2)))
                                   +"."+$vs4Dversion[[3]]+"."+$vs4Dversion[[4]])
```

2. This example tests to verify that you are using a final version:

```
⇒      If(Subtring(Application version(*);1;1)#"F")
          ALERT("Please make sure you are using a Final Production version of 4D
                                                        with this database!")
          QUIT 4D
       End if
```

See Also

Application type, Version type.

Compiled application

version 6.0

Compiled application → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	Compiled (True), Interpreted (False)

Description

Compiled application tests whether you are running in compiled mode (True) or interpreted mode (False).

Example

In one of your routines, you include debugging code useful only when you are running in interpreted mode, so surround this debugging code with a test that calls Compiled application:

```
          ` ...
⇒    If (Not(Compiled application))
               ` Include debugging code here
     End if
          ` ...
```

See Also

IDLE, Undefined.

PLATFORM PROPERTIES

4D Environment

version 3

PLATFORM PROPERTIES (platform; system; machine)

Parameter	Type		Description
platform	Number	←	1 68K-based Macintosh 2 Power Macintosh 3 Windows
system	Number	←	Depends on the version you are running
machine	Number	←	Depends on the version you are running

Description

The PLATFORM PROPERTIES command returns information about the type of platform you are running, the version of the operating system, and the processor installed on your machine.

PLATFORM PROPERTIES returns environment information in the parameters platform, system, and machine.

Platform indicates whether you are running a 68K or PowerPC-based Macintosh, or Windows version of 4 th Dimension. This parameter returns one the following predefined constants:

Constant	Type	Value
Macintosh 68K	Long Integer	1
Power Macintosh	Long Integer	2
Windows	Long Integer	3

The information returned in system and machine depends on the version of 4th Dimension you are running.

Macintosh (both 68K and PowerPC versions)

If you are running a MacOS version of 4th Dimension, the system and machine parameters return the following information.

• The system parameter returns a 32-bit (Long Integer) value, for which the high level word is unused and the low level word is structured like this:

- The high byte contains the major version number,
- The low byte is composed of two nibbles (4 bits each). The high nibble is the major update version number and the low nibble is the minor update version. Example: System 7.5.1 is coded as $0751, so you receive the decimal value 1873.

Note: In 4D, you can extract these values using the % (modulo) and \ (integer division) numeric operators or the bitwise operators introduced in version 6.

• The Machine parameter returns a unique ID number identifying the model of Macintosh.

Note: An update list of these unique ID numbers is published by Apple Computer, Inc. in its Developer and Technical documentation. New values may be added when Apple or other manufacturers release new models of the Macintosh.

Windows version

If you are running the Windows version of 4 th Dimension, system and machine return the following information.

• The system parameter returns a 32-bit (Long Integer) value, the bits and bytes of which are structured as follows:

If the high level bit is set to 1, it means you are running one of the types of Windows NT. If the bit is set to 0, it means you are running Windows 3.1 or Window 95.

Note: The high level bit fixes the sign of the long integer value. Therefore, in 4D, you just need to test the sign of the value; if it is positive you are running Windows NT. You can also use the bitwise operators introduced in version 6.

The low byte gives the major Windows version number. If it returns 3, you are running version 3.x of Windows or Window NT. If it returns 4, you are running Windows 95 or Windows NT 4. In both cases, the sign of the value tells whether or not you are running NT.

The next low byte gives the minor Windows version number.

Note: In 4D, you can extract these values using the % (modulo) and \ (integer division) numeric operators or the bitwise operators introduced in version 6.

• The machine parameter returns one the following predefined constants:

Constant	Type	Value
INTEL 386	Long Integer	386
INTEL 486	Long Integer	486
Pentium	Long Integer	586
PowerPC 601	Long Integer	601
PowerPC 603	Long Integer	603
PowerPC 604	Long Integer	604

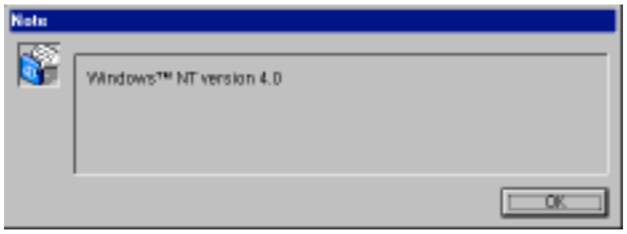
Note: Under Windows 3.1.x, machine returns 486 even though your machine is equipped with a Pentium processor.

Example

The following project method displays an alert box showing the OS software you are using:

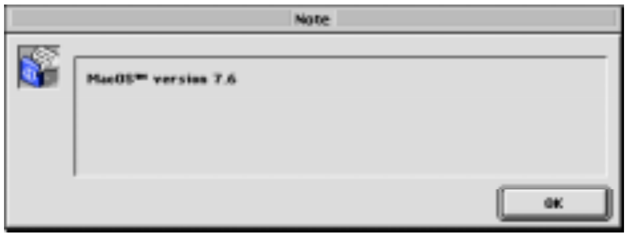
```
          ` SHOW OS VERSION project method
⇒      PLATFORM PROPERTIES($vlPlatform;$vlSystem;$vlMachine)
       If (($vlPlatform<1) | (3<$vlPlatform))
          $vsPlatformOS:=""
       Else
          If ($vlPlatform=3)
             $vsPlatformOS:=""
             If ($vlSystem<0)
                $winMajVers:=((2^31)+$vlSystem)%256
                $winMinVers:=(((2^31)+$vlSystem)\256)%256
                If ($winMajVers>=4)
                   $vsPlatformOS:="Windows™ 95"
                Else
                   $vsPlatformOS:="Windows™ (with Win32s)"
                End if
             Else
                $winMajVers:=$vlSystem%256
                $winMinVers:=($vlSystem\256)%256
                $vsPlatformOS:="Windows™ NT"
             End if
             $vsPlatformOS:=$vsPlatformOS+" version "+
                                          String($winMajVers)+"."+String($winMinVers)
          Else
             $vsPlatformOS:="MacOS™ version "+String($vlSystem\256)+
                                                    "."+String(($vlSystem\16)%16)+
                                 (("."+String($vlSystem%16))*Num(($vlSystem%16) # 0))
          End if
       End if
       ALERT($vsPlatformOS)
```

On Windows, you get an alert box similar to this:



Windows™ NT version 4.0

On Macintosh, you get an alert box similar to this:



MacOS™ version 7.6

See Also

Bitwise Operators.

Application file

Application file → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Long name of the 4D executable file or application

Description

The Application file command returns the long name of the 4D executable file or application you are running.

On Windows
If, for example, you are running 4th Dimension located at \4DWIN600\PROGRAM on the volume E, the command returns E:\4DWIN600\PROGRAM\4D.EXE.

On Macintosh
If, for example, you are running 4th Dimension in the folder 4th Dimension® 6.0*f* on the disk Macintosh HD, the command returns Macintosh HD:4th Dimension® 6.0*f*:4th Dimension® 6.0.

Example

At startup on Windows, you need to check if a DLL Library is correctly located at the same level as the 4D executable file. In the On Startup database method of your application you can write:

```
     If (On Windows & (Application type#4D Server))
⇒       If (Test path name (Long name to path name (
                                     Application file)+"XRAYCAPT.DLL")#Is a document)
           ` Display a dialog box explaining that the library XRAYCAPT.DLL
           ` is missing. Therefore, the X-rays capture capabilitity will not be available.
        End if
     End if
```

Note: The project methods On Windows and Long name to path name are listed in the section System Documents.

See Also

Data file, DATA SEGMENT LIST, Structure file.

Structure file

4D Environment

version 6.0

Structure file → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Long name of the database structure file

Description

The Structure file command returns the long name of structure file for the database with which you are currently working.

On Windows
If, for example, you are working with the database MyCDs located in \DOCS\MyCDs on the volume G, the command returns G:\DOCS\MyCDs\MyCDs.4DB.

On Macintosh
If, for example, you are are working with the database located in the folder Documents:MyCDsƒ: on the disk Macintosh HD, the command returns Macintosh HD:Documents:MyCDsƒ:MyCDs.

WARNING: If you call this command while running 4D Client, only the name of the structure file is returned; the long name is not returned.

Example

This example displays the name and the location of the structure file currently in use:

```
     If (Application type#4D Client)
⇒       $vsStructureFilename:=Long name to file name (Structure file)
⇒       $vsStructurePathname:=Long name to path name (Structure file)
        ALERT("You are currently using the database "+
                            Char(34)+$vsStructureFilename+Char(34)+" located at "+
                                        Char(34)+$vsStructurePathname+Char(34)+".")
     Else
⇒       ALERT("You are connected to the database "+Char(34)+Structure file+Char(34))
     End if
```

Note: The project methods Long name to file name and Long name to path name are listed in the section System Documents.

See Also

Application file, Data file, DATA SEGMENT LIST.

Data file

Data file {(segment)} → String

Parameter	Type		Description
segment	Number	→	Segment number
Function result	String	←	Long name of the data file for the database

Description

The Data file command returns the long name of the data file or one data segment for the database with which you are currently working.

If you do not pass the segment parameter, it returns the long name of the data file or the first segment (if the database is segmented). If you pass the segment parameter, it returns the long name of the corresponding data segment. If you pass a segment number greater than the number of data segments, it returns an empty string.

On Windows
If, for example, you are working with the database MyCDs located at \DOCS\MyCDs on the volume G, a call to Data file returns G:\DOCS\MyCDs\MyCDs.4DD (provided that you accepted the default location and name proposed by 4D when you created the database).

On Macintosh
If, for example, you are working with the database located in the folder Documents:MyCDsƒ: on the disk Macintosh HD, a call to Data file returns Macintosh HD:Documents:MyCDsƒ:MyCDs.data (provided that you accepted the default location and name proposed by 4D when you created the database).

WARNING: If you call this command while running 4D Client, only the name of the data file or the first data segment is returned, not the long name. In addition, even though the database is segmented, the command returns an empty string for the other data segments. If you need (for adminstrative purposes) to display a list of the data segments on a 4D Client station, use a Stored Procedure to build the data segment list and store it in a variable on the server machine, then get the contents of this variable using the GET PROCESS VARIABLE command.

Example

The following code goes through the data segments of a database.

```
        If (Application type#4D Client)
            $vlDataSegNum:=0
            Repeat
                $vlDataSegNum:=$vlDataSegNum+1
⇒               $vsDataSegName:=Data file($vlDataSegNum)
                If ($vsDataSegName#"")
                    ALERT ("Data segment "+String($vlDataSegNum)+":"+Char(34)+
                                                          $vsDataSegName+Char(34)+".")
                End if
            Until ($vsDataSegName="")
            ALERT("There is/are "+String($vlDataSegNum-1)+"data segment(s).")
        End if
```

See Also

Application file, DATA SEGMENT LIST, Structure file.

ACI folder

ACI folder → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Pathname to ACI Folder

Description

The **ACI folder** command returns the pathname to the ACI folder located in the active system folder or directory.

On Windows

The ACI directory (folder) is named ACI and is located in the active WINDOWS directory (usually C:\WINDOWS). This is why the **ACI folder** command usually returns the pathname C:\WINDOWS\ACI\. However, PC computers can be set up with multi-boot configurations, and, the location and name of the active WINDOWS directory can be customized during installation. Therefore, if you want to save your own files (documents) in the ACI folder, this command enables you to get the actual pathname to that directory.

On Macintosh

The ACI folder is named ACI and is located in the Preferences folder of the active system folder. Typically, the pathname Macintosh HD:System folder:Preferences:ACI: is the value returned by the **ACI folder** command after a fresh installation of 4D. Because Macintosh users can rename their disks and system folders, the pathname to the ACI folder can vary. Therefore, if you want to save your own documents (files) in the ACI folder, this command enables you to get the actual pathname to this folder.

Platform Independence and International: By using the **ACI folder** command to get the actual pathname to that folder, you also ensure that your code will work on any platform running any localized system.

The 4D environment uses the ACI folder to store the following information:

• User registration files
• Preferences files used by the 4D environment applications, tools, and utility programs
• 4D Client/Server or Internet/Intranet Network Components (on Windows only, within the ...\ACI\NETWORK directory) as well as their option files
• .rex and res files created by 4D Client for storing resources downloaded from 4D Server
• Local database folders created by 4D Client for storing the 4D Extensions downloaded from 4D Server

WARNING: You are free to store whatever files or documents you wish into the ACI folder, however, it is good idea not to move or modify the files created by the 4D environment itself.

Example

During the startup of a single-user database, you want to load (or create) your own settings in a file located in the ACI folder. To do so, in the On Startup database method, you can write code similar to this:

```
     MAP FILE TYPES("PREF";"PRF";"Preferences file")
        ` Map PREF MacOS file type to .PRF Windows file extension
⇒    $vsPrefDocName:=ACI folder+"MyPrefs"   ` Build pathname to the Preferences file
        ` Check if the file exists
     If (Test pathname($vsPrefDocName+(".PRF"*Num(On Windows)))#Is a document)
        $vtPrefDocRef:=Create document($vsPrefDocName;"PREF") ` If not, create it
     Else
        $vtPrefDocRef:=Open document($vsPrefDocName;"PREF") ` If so, open it
     End if
     If (OK=1)
        ` Process document contents
        CLOSE DOCUMENT($vtPrefDocRef)
     Else
        ` Handle error
     End if
```

See Also

System folder, Temporary folder, Test path name.

DATA SEGMENT LIST

4D Environment

version 6.0

DATA SEGMENT LIST (Segments)

Parameter	Type		Description
Segments	String array	←	Long names of data segments for the database

Description

DATA SEGMENT LIST populates the segments array with the long names of the data segments for the database with which you are currently working.

WARNING: This command does nothing if executed on 4D Client. If you need (for administrative purposes) to display a list of the data segments on a 4D Client station, use a Stored Procedure to build the data segment list and store it in a variable on the server machine, then get the contents of this variable using the GET PROCESS VARIABLE command.

Examples

1. In the Data Segments Information form for the [Dialogs] table, you want to display a drop-down list populated with the names of the data segments. To do so, write:

```
	` [Dialogs];"Data Segments Information" form method
      Case of
	: (Form event=On Load )
	      ` ...
	   ARRAY STRING(255;asDataSegName;0)
⇒	   DATA SEGMENT LIST(asDataSegName)
	      ` ...
      End case
```

2. The following method tells you if a database is segmented.

```
	` Is data file segmented -> Boolean
      C_BOOLEAN ($0)
⇒     DATA SEGMENT LIST($asDataSegName)
      $0:=(Size of array($asDataSegName)>1)
```

3. After a call to ADD DATA SEGMENT, you want to test whether the user added new segments.

```
⇒   DATA SEGMENT LIST($asBefore)
    ADD DATA SEGMENT
⇒   DATA SEGMENT LIST($asAfter)
    If(Size of array($asBefore)#Size of array($asAfter))
        ` Yes, there are more data segments
    Else
        ` Same number of data segments
    End if
```

See Also

Application file, Data file, Structure file.

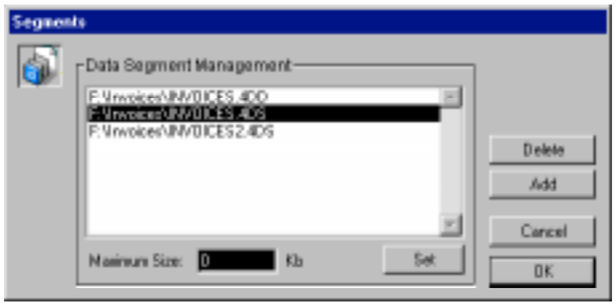
ADD DATA SEGMENT

Parameter	**Type**	**Description**

This command does not require any parameters

Description

The ADD DATA SEGMENT command displays the Data Segment Management dialog box shown here:



If the user clicks the OK button to validate the dialog box, the OK variable is set to 1. If the user clicks the Cancel button, OK is set to 0.

NOTE: This command does nothing when used with 4D Server.

When all data segments are full, 4th Dimension or 4D Server generates an error -9999. An error message is displayed, stating that the disk is full.

If you are using 4th Dimension, you can use the ON ERR CALL method to trap the error message so you can handle the error procedurally. You can then use ADD DATA SEGMENT to allow the user to add a new data segment on another volume that has available space.

If you are using 4D Server, you can display an alert stating that the Database Administrator must add a new data segment from the server machine.

See Also

ON ERR CALL.

System Variables and Sets

OK is set to 1 if the Data Segment Management dialog box is validated.

FLUSH BUFFERS

4D Environment

version 3

FLUSH BUFFERS

Parameter	Type	Description
This command does not require any parameters		

Description

The command FLUSH BUFFERS immediately saves the data buffers to disk. All changes that have been made to the database are stored on disk.

You usually do not need to call this command, as 4D saves data modification on a regular basis. The database property **Flush Data Buffers** (in the Design environment), which specifies how often to save, is typically used to control buffer flushing.

Note: 4D integrates a built-in data cache scheme for accelerating I/O operations. The fact that data modifications are, for some time, present in the data cache and not on the disk is transparent to your coding. For example, if you issue a QUERY call, the 4D database engine integrates the data cache in the query operation.

QUIT 4D

QUIT 4D

Parameter	Type	Description

This command does not require any parameters

Description

The QUIT 4D command exits 4th Dimension and returns to the Desktop. After you call QUIT 4D, the current process stops its execution, then 4D acts as follows:

• If there is an On Exit Database Method, 4D starts executing this method within a newly created local process. For example, you can use this database method to inform other processes, via interprocess communication, that they must close (data entry) or stop the execution of operations started by the On Startup Database Method (connection from 4D to another database server). Note that 4D will eventually quit; the On Exit Database Method can perform all the cleanup or closing operations you wish, but cannot refuse the quit and will at some point end.

• If there is no On Exit Database Method, 4D aborts each running process one by one, without distinction.

If the user is performing data entry, the records will be cancelled and not saved.

If you want to let the user save data entry modifications made in the current open windows, you can use interprocess communication to signal all the other user processes that the database is going to be exited. To do so, you can adopt two strategies:

• Perform these operations from within the current process before calling QUIT 4D
• Handle these operations from within the On Exit Database Method.

A third strategy is also possible. Before calling QUIT 4D, you check whether a window will need validation; if that is the case, you ask the user to validate or cancel these windows and then to choose Quit again. However, from a user interface standpoint, the first two strategies are preferable.

Example

The project method listed here is associated with the Quit or Exit menu item in the File menu.

```
		` M_FILE_QUIT Project Method

	CONFIRM("Are you sure that you want to quit?")
	If (OK=1)
⇒		QUIT 4D
	End if
```

See Also

On Exit Database Method.

SELECT LOG FILE

SELECT LOG FILE (logFile)

Parameter	Type		Description
logFile	String	→	Name of the Log file or "*" for closing the current Log file

Description

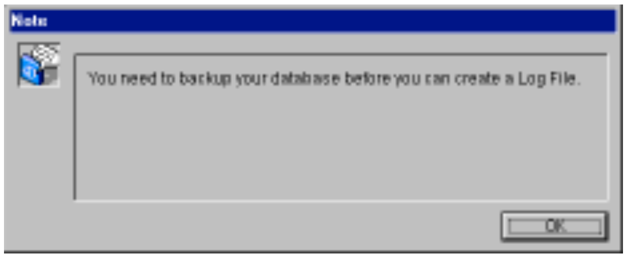
The SELECT LOG FILE command opens, creates, or closes the Log File according to the value you pass in logFile.

IMPORTANT: Calling SELECT LOG FILE is the same as choosing Log File from the File menu in the User environment. This should only be used when 4D Backup is installed in the database.

If you pass an empty string in logFile, SELECT LOG FILE presents an Open File dialog box, allowing the user to open a log file or to create a new one. If the user clicks the Open button and the file is opened correctly, the OK variable is set to 1. Otherwise, if the user clicks Cancel or if the Log File could not be opened or created, OK is set to 0.

If you pass "*" in logFile, SELECT LOG FILE closes the current Log File for the database. The OK variable is set to 1 when the log file is closed.

If you use SELECT LOG FILE to create or open a Log File when a full backup has not yet been performed and the data file already contains records, 4th Dimension displays the following alert:



You need to backup your database before you can create a Log File.

4D then generates an error -4447, which you can intercept with an ON ERR CALL method.

Note: The SELECT LOG FILE command does not do anything when used with 4D Server. For more information about this command, see the documentation for the 4D Backup plug-in.

See Also

ON ERR CALL.

System Variables and Sets

OK is set to 1 if the Log File is correctly opened, created, or closed.

Error Handling

An error -4447 is generated if the operation cannot be performed because the database needs to be backed up. You can intercept the error with an ON ERR CALL method.

4 Arrays

Arrays

version 6.0

An **array** is an ordered series of variables of the same type. Each variable is called an **element** of the array. The **size** of an array is the number of elements it holds. An array is given its size when it is created; you can then resize it as many times as needed by adding, inserting, or deleting elements, or by resizing the array using the same command used to create it.

You create an array with one of the array declaration commands. For details, see the section Creating Arrays.

Elements are numbered from **1 to N**, where N is the size of the array. An array always has an **element zero** that you can access just like any other element of the array, but this element is not shown when an array is present in a form. Although the element zero is not shown when an array supports a form object, there is no restriction in using it with the language. For more information about the element zero, see the section Using the element zero of an array.

Arrays are 4D variables. Like any variable, an array has a scope and follows the rules of the 4D language, though with some unique differences. For more information, see the sections Arrays and the 4D Language and Arrays and Pointers.

Arrays are language objects; you can create and use arrays that will never appear on the screen. Arrays are also user interface objects. For more information about the interaction between arrays and form objects, see the sections Arrays and Form Objects and Grouped Scrollable Areas.

Arrays are designed to hold reasonable amounts of data for a short period of time. However, because arrays are held in memory, they are easy to handle and quick to manipulate. For details, see the section Arrays and Memory.

Creating Arrays

You create an array with one of the array declaration commands described in this chapter. The following table lists the array declaration commands:

Command	Creates or resizes an array of:
ARRAY INTEGER	2-byte Integer values
ARRAY LONGINT	4-byte Integer values
ARRAY REAL	Real values
ARRAY TEXT	Text values (from 0 to 32,000 characters per element) (see Note)
ARRAY STRING	String values (from 0 to 255 characters per element) (see Note)
ARRAY DATE	Date values
ARRAY BOOLEAN	Boolean values
ARRAY PICTURE	Pictures values
ARRAY POINTER	Pointer values

Each array declaration command can create or resize one-dimensional or two-dimensional arrays. For more information about two-dimensional arrays, see the section Two-dimensional Arrays.

Note: The difference between Text arrays and String arrays lies in the nature of their elements. In both types of array, elements can hold text values (characters). However:

• In a Text array, each element is of variable length and stores its characters in a separate part of memory.
• In a String array, all elements have the same fixed length (the length passed when the array was created). All elements are stored one after the other in the same part of memory, no matter what the contents.

Due to this structural difference, string arrays act faster than text arrays. Note, however, that an element of a String array can only hold up to 255 characters.

The following line of code creates (declares) an Integer array of 10 elements:

```
ARRAY INTEGER(aiAnArray;10)
```

Then, the following code resizes that same array to 20 elements:

```
ARRAY INTEGER(aiAnArray;20)
```

Then, the following code resizes that same array to no elements:

```
ARRAY INTEGER(aiAnArray;0)
```

You reference the elements in an array by using curly braces ({…}). A number is used within the braces to address a particular element; this number is called the **element number**.

The following lines put five names into the array called atNames and then display them in alert windows:

```
ARRAY TEXT (atNames;5)
atNames{1} := "Richard"
atNames{2} := "Sarah"
atNames{3} := "Sam"
atNames{4} := "Jane"
atNames{5} := "John"
For ($vlElem;1;5)
    ALERT ("The element #"+String($vlElem)+" is equal to: "+atNames{$vlElem})
End for
```

Note the syntax atNames{$vlElem}. Rather than specifying a numeric literal such as atNames{3}, you can use a numeric variable to indicate which element of an array you are addressing.

Using the iteration provided by a loop structure (For...End for, Repeat... Until (...) or While (...) End while), compact pieces of code can address all or part of the elements in an array.

Arrays and other areas of the 4D language

There are other 4D commands that can create and work with arrays. For more information, refer to the descriptions of the following commands:

• To work with arrays and selection of records, use the commands SUBSELECTION TO ARRAY, SELECTION TO ARRAY, ARRAY TO SELECTION and DISTINCT VALUES.

• You can create graphs and charts on series of values stored in tables, subtables, and arrays. For more information, see the GRAPH command.

• Although version 6 brings a full set of new commands to work with hierarchical lists, the commands LIST TO ARRAY and ARRAY TO LIST (from the previous version) have been retained for compatibility.

• New commands in version 6 build arrays in one call. These commands are FONT LIST, WINDOW LIST, VOLUME LIST, FOLDER LIST, and DOCUMENT LIST.

See Also

ARRAY BOOLEAN, ARRAY DATE, ARRAY INTEGER, ARRAY LONGINT, ARRAY PICTURE, ARRAY POINTER, ARRAY REAL, ARRAY STRING, ARRAY TEXT, Arrays, Two-dimensional Arrays.

Arrays and Form Objects

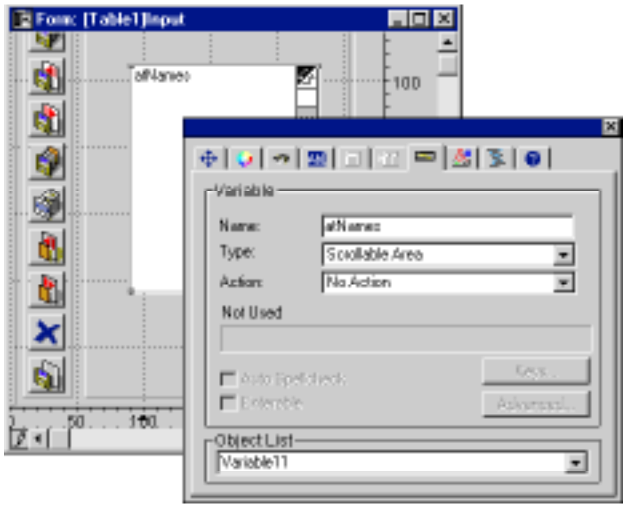
Arrays are language objects—you can create and use arrays that will never appear on the screen. However, arrays are also user interface objects. The following types of **Form Objects** are supported by arrays:

- Pop-up menu
- Drop-down List
- Combo Box
- Scrollable Area
- Tab Control

While you can predefine these objects in the Design Environment Form Editor (using the Default Values button of the Object Properties window), you can also define them programmatically using the arrays commands. In both cases, the form object is supported by an array created by you or 4D.

When using these objects, you can detect which item within the object has been selected (or clicked) by testing the array for its **selected element**. Conversely, you can select a particular item within the object by setting the selected element for the array.

When an array is used to support a form object, it has then a dual nature; it is both a language object and a user interface object. For example, when designing a form, you create a scrollable area; in the **Variable** page of the Object Properties window, you name the Variable Object:



Variable

Name: atNames

Type: Scrollable Area

Action: No Action

Object List: Variable11

The name, in this case atNames, is the name of the array you use for creating and handling the array.

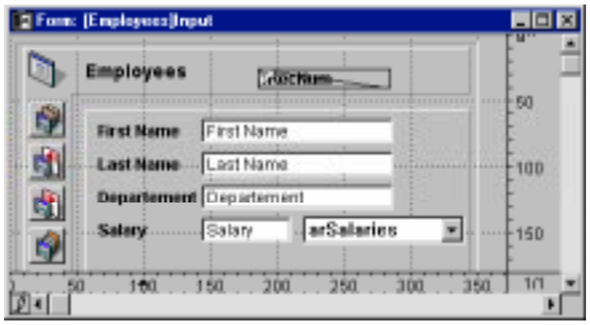
Note: You cannot display two-dimensional arays or pointer arrays.

Example: Creating a drop-down list

The following example shows how to fill an array and display it in a drop-down list. An array arSalaries is created using the ARRAY REAL command. It contains all the standard salaries paid to people in a company. When the user chooses an element from the drop-down list, the [Employees]Salary field is assigned the value chosen in the User or Custom Menus environment.

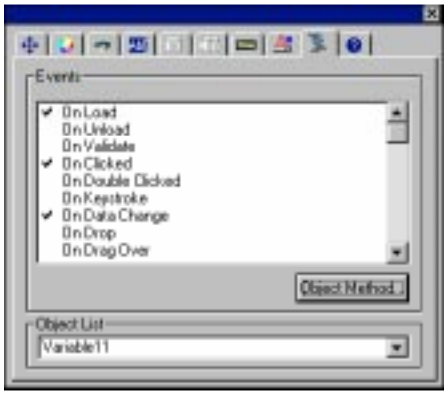
Create the arSalaries drop-down list on a form

Create a drop-down list and name it arSalaries. The name of the drop-down list should be the same as the name of the array.

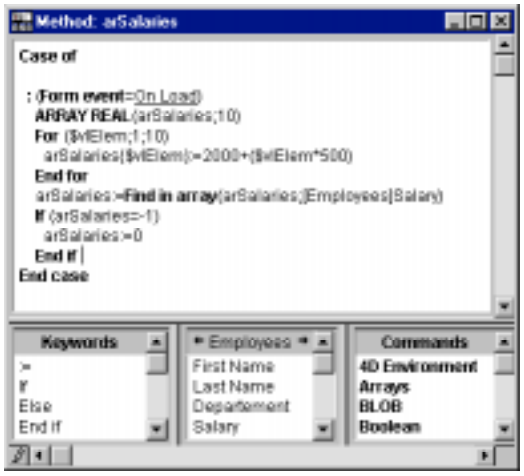


Initializing the array

Initialize the array arSalaries using the On Load event for the object. To do so, remember to enable that event in the **Object Properties** window, as shown:



Click the **Object Method** button and create the method, as follows:



```
Case of

 : (Form event=On Load)
  ARRAY REAL(arSalaries;10)
  For ($vlElem;1;10)
   arSalaries{$vlElem}:=2000+($vlElem*500)
  End for
  arSalaries:=Find in array(arSalaries;[Employees]Salary)
  If (arSalaries=-1)
   arSalaries:=0
  End if
End case
```

The lines:

```
ARRAY REAL(arSalaries;10)
For($vlElem;1;10)
    arSalaries{$vlElem}:=2000+($vlElem*500)
End for
```

create the numeric array 2500, 3000... 7000, corresponding to the annual salaries $30,000 up to $84,000, before tax.

The lines:

```
arSalaries:=Find in array(arSalaries;[Employees]Salary)
If (arSalaries=-1)
    arSalaries:=0
End if
```

handle both the creation of a new record or the modification of existing record.

• If you create a new record, the field [Employees]Salary is initially equal to zero. In this case, Find in array does not find the value in the array and returns -1. The test If (arSalaries=-1) resets arSalaries to zero, indicating that no element is selected in the drop-down list.
• If you modify an existing record, Find in array retrieves the value in the array and sets the selected element of the drop-down list to the current value of the field. If the value for a particular employee is not in the list, the test If (arSalaries=-1) deselects any element in the list.

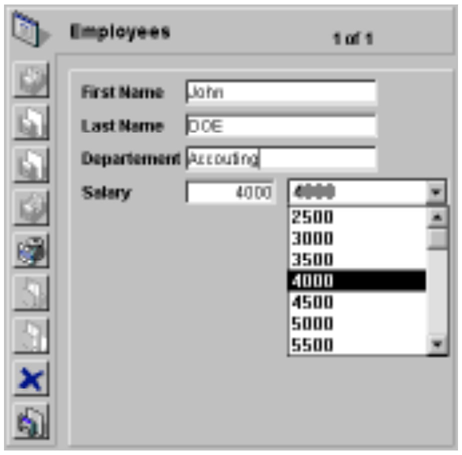
Note: For more information about the **array selected element**, read the next section.

Reporting the selected value to the [Employees]Salary field
To report the value selected from the drop-down list arSalaries, you just need to handle the On Clicked or On Data Change event to the object. The element number of the selected element is the value of the array arSalaries itself. Therefore, the expression arSalaries{arSalaries} returns the value chosen in the drop-down list.

Complete the method for the object arSalaries as follows:

```
Case of
    : (Form event=On Load)
        ARRAY REAL(arSalaries;10)
        For($vlElem;1;10)
            arSalaries{$vlElem}:=2000+($vlElem*500)
        End for
        arSalaries:=Find in array(arSalaries;[Employees]Salary)
        If (arSalaries=-1)
            arSalaries:=0
        End if
    : (Form event=On Data Change)
        [Employees]Salary:=arSalaries{arSalaries}
End case
```

In the User or Custom Menus environment, the drop-down list looks like this:



First Name	John
Last Name	DOE
Departement	Accouting
Salary	4000

The following section describes the common and basic operations you will perform on arrays while using them as form objects.

Getting the size of the array

You can obtain the current size of the array by using the Size of array command. Using the previous example, the following line of code would display 5:

```
ALERT ("The size of the array atNames is: "+String(Size of array(atNames)))
```

Reordering the elements of the array

You can reorder the elements of the array using the SORT ARRAY command. Using the previous example, and provided the array is shown as a scrollable area:

a. Initially, the area would look like the list on the left.

b. After the execution of the following line of code:

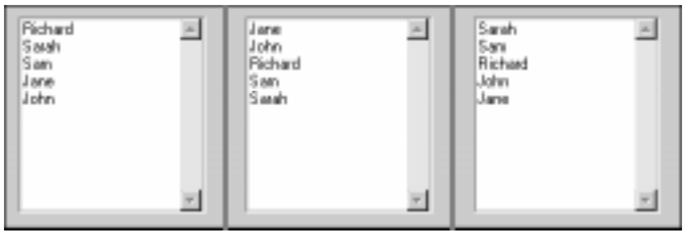
```
SORT ARRAY(atNames;>)
```

the area would look like the list in the middle.

c. After the execution of the following line of code:

```
SORT ARRAY(atNames;<)
```

the area would look like the list on the right.



Adding or deleting elements

You can add, insert, or delete elements using the commands INSERT ELEMENT and DELETE ELEMENT.

Handling clicks in the array: testing the selected element

Using the previous example, and provided the array is shown as a scrollable area, you can handle clicks in this area as follows:

```
    ` atNames scrollable area object method
Case of
    : (Form event=On Load)
          ` Initialize the array (as shown further above)
        ARRAY TEXT (atNames;5)
          ` ...
    : (Form event=On Unload)
          ` We no longer need the array
        CLEAR VARIABLE(atNames)

    : (Form event=On Clicked)
        If (atNames#0)
            vtInfo:="You clicked on: "+atNames{atNames}
        End if
    : (Form event=On Double Clicked)
        If (atNames#0)
            ALERT ("You double clicked on: "+atNames{atNames}
        End if
End case
```

Note: The events must be activated in the Object Properties window.

While the syntax atNames{$vlElem} allows you to work with a particular element of the array, the syntax atNames returns the element number of the selected element within the array. Thus, the syntax atNames{atNames} means "the value of the selected element in the array atNames." If no element is selected, atNames is equal to 0 (zero), so the test If (atNames#0) detects whether or not an element is actually selected.

Setting the selected element

In a similar fashion, you can programmatically change the selected element by assigning a value to the array.

Examples

```
    ` Selects the first element (if the array is not empty)
atNames:=1

    ` Selects the last element (if the array is not empty)
atNames:=Size of array(atNames)
```

```
  ` Deselects the selected element (if any) then no element is selected
atNames:=0

If ((0<atNames)&(atNames<Size of array(atNames))
      ` If possible, selects the next element to the selected element
   atNames:=atNames+1
End if

If (1<atNames)
       `  If possible, selects the previous element to the selected element
   atNames:=atNames-1
End if
```

Looking for a value in the array

The Find in array command searches for a particular value within an array. Using the previous example, the following code will select the element whose value is "Richard," if that is what is entered in the request dialog box:

```
$vsName:=Request("Enter the first name:")
If (OK=1)
   $vlElem:=Find in array (atNames;$vsName)
   If ($vlElem>0)
      atNames:=$vlElem
   Else
      ALERT ("This is no "+$vsName+" in that list of first names.")
   End if
End if
```

Pop-up menus, drop-down lists, scrollable areas, and tab controls can be usually handled in the same manner. Obviously, no additional code is required to redraw objects on the screen each time you change the value of an element, or add or delete elements.

Note: To create and use tab controls with icons and enabled and disabled tabs, you must use a hierarchical list as the supporting object for the tab control. For more information, see the example for the New list command.

Handling combo boxes

While you can handle pop-up menus, drop-down lists, scrollable areas, and tab controls with the algorithms described in the previous section, you must handle combo boxes differently.

A combo box is actually a text enterable area to which is attached a list of values (the elements from the array). The user can pick a value from this list, and then edit the text. So, in a combo box, the notion of selected element does not apply.

With combo boxes, there is never a selected element. Each time the user selects one of the values attached to the area, that value is put into the element zero of the array. Then, if the user edits the text, the value modified by the user is also put into that element zero.

Example

```
     ` asColors Combo Box object method
  Case of
     : (Form event=On Load)
        ARRAY STRING(31;asColors;3)
        asColors{1}:="Blue"
        asColors{2}:="White"
        asColors{3}:="Red"
     : (Form event=On Clicked)
        If (asColors{0}#"")
              ` The object automatically changes its value
              ` Using the On Clicked event with a Combo Box
              ` is required only when additional actions must be taken
        End if
     : (Form event=On Data Change)
           ` Find in array ignores element 0, so returns -1 or >0
        If (Find in array(asColors;asColors{0})<0)
              ` Entered value is not one the values attached to the object
              ` Add the value to the list for next time
           $vlElem:=Size of array(asColors)+1
           INSERT ELEMENT(asColors;$vlElem)
           asColors{$vlElem}:=asColors{0}
        Else
              ` Entered value is among the values attached to the object
        End if
  End case
```

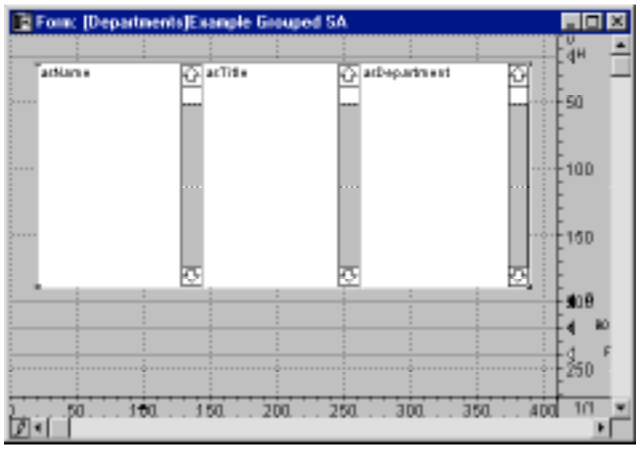
See Also

Arrays, Grouped Scrollable Areas.

Grouped Scrollable Areas

version 6.0

You can group scrollable areas for display in a form. When several scrollable areas are grouped, they act as one scrollable area. Each scrollable area can have its own font and style; however, we recommend that you use the same font height (which depends on the font and font size) for each column. When displayed during data entry, only the frontmost scrollable area displays a scroll bar. Following are three scrollable areas grouped together in the Design environment:

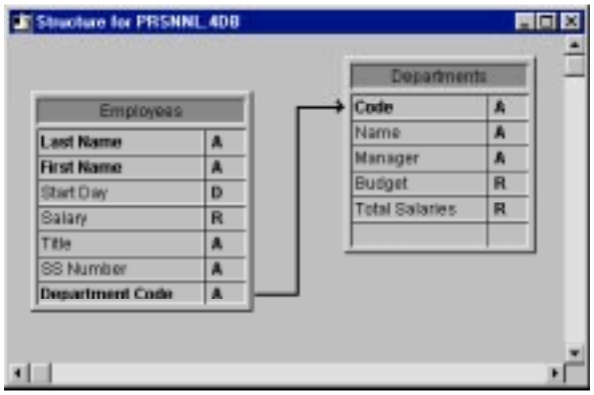


Here are some tips on creating grouped scrollable areas:
• Make sure that all the arrays have been given the same size (number of elements).
• Use the same font size for each area.
• Make each area the same height.
• Align the tops of all the areas.
• Make sure the areas do not overlap.
• Make sure that the area on the right is in front, because the scroll bar appears on the frontmost area.
• Group the areas (using the Group menu command) to make them work as one scrollable area.

The following project method fills the three arrays and displays them on the screen:

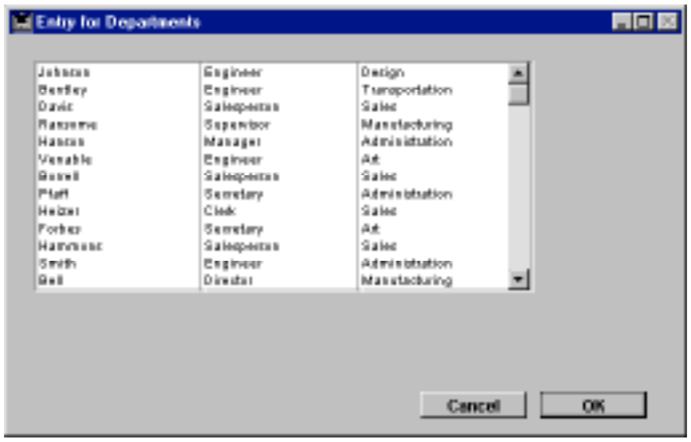
```
ALL RECORDS(Employees)
SELECTION TO ARRAY([Employees]Last Name;asName;[Employees]Title;asTitle;[Departments]Name;asDepartment)
DIALOG([Departments];"Example Grouped SA")
```

This method uses the data in the fields of the [People] table and the [Departments] table. These tables are shown here:



Note: The [Departments] table can be used, provided that there is an automatic relation from [People] to [Departments].

The resulting display:



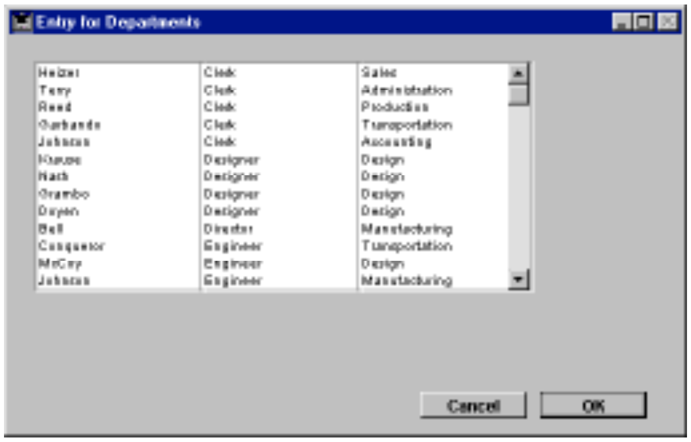
Johnson	Engineer	Design
Bentley	Engineer	Transportation
Davis	Salesperson	Sales
Ransome	Supervisor	Manufacturing
Hanson	Manager	Administration
Venable	Engineer	Art
Bowell	Salesperson	Sales
Pfaff	Secretary	Administration
Heizer	Clerk	Sales
Forbes	Secretary	Art
Hammond	Salesperson	Sales
Smith	Engineer	Administration
Bell	Director	Manufacturing

Note that only a single scroll bar is displayed; it is always on the frontmost scrollable area. This scroll bar controls the scrolling of all three arrays as if they were one. When the user clicks a line, all three areas are highlighted simultaneously. The variable associated with each scrollable area is set to the number of the line that the user clicks; only the object method for the area that is clicked executes. For example, if the user clicks the name "Bentley," asName, asTitle, and asDepartment are all set to two, but only the object method for asName executes. If you set the selected element of one of the arrays in the grouped scrollable areas, the other arrays are set to the same selected element for the next event, and the respective line in the scrollable area is highlighted.

The arrays can be sorted with the command SORT ARRAY. For example:

SORT ARRAY(asTitle;asName;asDepartment;>)

The following is the result of the sort:



Heizer	Clerk	Sales
Terry	Clerk	Administration
Reed	Clerk	Production
Garbando	Clerk	Transportation
Johnson	Clerk	Accounting
Krause	Designer	Design
Nash	Designer	Design
Grambo	Designer	Design
Dryen	Designer	Design
Bell	Director	Manufacturing
Conqueror	Engineer	Transportation
McCoy	Engineer	Design
Johnson	Engineer	Manufacturing

Note that the arrays were sorted based on the first argument to the SORT ARRAY command; the other two arrays were specified in order to keep the rows synchronized. The command SORT ARRAY always sorts the arrays (if several are specified) on the values of the first array and keeps the additional arrays synchronized.

Note: SORT ARRAY does not perform a multi-level sort on arrays. To show a table similar to the one above and also perform multi-level sorts (i.e., by department, then by title, then by name), use a subform in which you display the table, and then use ORDER BY.

See Also

Arrays, Arrays and Form Objects.

Arrays and the 4D Language

Arrays are 4D variables. Like any variable, an array has a scope and follows the rules of the 4D language, though with some unique differences.

Local, process and interprocess arrays

You can create and work with local, process, and interprocess arrays. Examples:

```
	ARRAY INTEGER ($aiCodes;100) ` This creates a local array of 100 2-byte Integer values
	ARRAY INTEGER (aiCodes;100) ` This creates a process array of 100 2-byte Integer values
	ARRAY INTEGER (<>aiCodes;100) ` This creates an interprocess array of 100 2-byte Integer values
```

The scope of these arrays is identical to the scope of other local, process, and interprocess variables:

Local arrays
A local array is declared when the name of the array starts with a dollar sign ($).

The scope of a local array is the method in which it is created. The array is cleared when the method ends. Local arrays with the same name in two different methods can have different types, because they are actually two different variables with different scopes.

When you create a local array within a form method, within an object method, within or a project method called as subroutine by the two previous type of method, the array is created and cleared each time the form or object method is invoked. In other words, the array is created and cleared for each form event. Consequently, you cannot use local arrays in forms, neither for display nor printing.

As with local variables, it is a good idea to use local arrays whenever possible. In doing so, you tend to minimize the amount of memory necessary for running your application.

Process arrays
A process array is declared when the name of the array starts with a letter.

The scope of a process array is the process in which it is created. The array is cleared when the process ends or is aborted. A process array automatically has one instance created per process. Therefore, the array is of the same type throughout the processes. However, its contents are particular to each process.

Interprocess arrays

An interprocess array is declared when the name of the array starts with <> (on Windows and Macintosh) or with the diamond sign, Option-Shift-V on a US keyboard (on Macintosh only).

The scope of an interprocess array consists of all processes during a working session. They should be used only to share data and transfer information between processes.

Tip: When you know in advance that an interprocess array will be accessed by several processes that could possible conflict, protect the access to that array with a semaphore. For more information, see the example for the Semaphore command.

Note: You can use process and interprocess arrays in forms to create form objects such as scrollable areas, drop-down lists, and so on.

Passing an Array as parameter

You can pass an array as parameter to a 4D command or to the routine of a 4D Plug-in. On the other hand, you cannot pass an array as parameter to a user method. The alternative is to pass a pointer to the array as parameter to the method. For details, see the section Arrays and Pointers.

Assigning and array to another array

Unlike text or string variables, you cannot assign one array to another. To copy (assign) an array to another one, use COPY ARRAY.

See Also

Arrays, Arrays and Pointers.

Arrays and Pointers

You can pass an array as parameter to a 4D command or to the routine of a 4D Plug-in. On the other hand, you cannot pass an array as parameter to a user method. The alternative is to pass a pointer to the array as parameter to the method.

Note: You can pass process and interprocess arrays as parameters, but not local arrays.

Here are some examples.

• Given this example:

```
If ((0<atNames)&(atNames<Size of array(atNames))
        ` If possible, selects the next element to the selected element
    atNames:=atNames+1
End if
```

If you need to do the same thing for 50 different arrays in various forms, you can avoid writing the same thing 50 times, by using the following project method:

```
    ` SELECT NEXT ELEMENT project method
    ` SELECT NEXT ELEMENT ( Pointer )
    ` SELECT NEXT ELEMENT ( -> Array )

C_POINTER ($1)

If ((0<$1->)&($1-><Size of array($1->))
    $1->:=$1->+1 ` If possible, selects the next element to the selected element
End if
```

Then, you can write:

```
SELECT NEXT ELEMENT (->atNames)
    ` ...
SELECT NEXT ELEMENT (->asZipCodes)
    ` ...
SELECT NEXT ELEMENT (->alRecordIDs)
    ` ... and so on
```

• The following project method returns the sum of all the elements of a numeric array (Integer, Long Integer, or real):

```
    ` Array sum
    ` Array sum ( Pointer )
    ` Array sum ( -> Array )

C_REAL ($0)

$0:=0
For ($vlElem;1;Size of array($1->))
    $0:=$0+$1->{$vlElem}
End for
```

Then, you can write:

```
$vlSum:=Array sum (->arSalaries)
    ` ...
$vlSum:=Array sum (->aiDefectCounts)
    ` ..
$vlSum:=Array sum (->alPopulations)
```

• The following project method capitalizes of all the elements of a string or text array:

```
    ` CAPITALIZE ARRAY
    ` CAPITALIZE ARRAY ( Pointer )
    ` CAPITALIZE ARRAY ( -> Array )

For ($vlElem;1;Size of array($1->))
    If ($1->{$vlElem}#"")
        $1->{$vlElem}:=Uppercase($1->{$vlElem}[[1]])+
                                                  Lowercase(Substring($1->{$vlElem};2))
    End if
End for
```

Then, you can write:

```
CAPITALIZE ARRAY (->atSubjects )
    ` ...
CAPITALIZE ARRAY (->asLastNames )
```

The combination of arrays, pointers, and looping structures, such as For... End for, allows you to write many useful small project methods for handling arrays.

See Also

Arrays, Arrays and the 4D Language.

Using the element zero of an array

An array always has an element zero. While element zero is not shown when an array supports a form object, there is no restriction in using it with the language.

One example of the use of element zero is the case of the combo box discussed in the section Arrays and Form Objects.

Here are two other examples.

1. If you want to execute an action only when you click on an element other than the previously selected element, you must keep track of each selected element. One way to do this is to use a process variable in which you maintain the element number of the selected element. Another way is to use the element zero of the array:

```
     ` atNames scrollable area object method
  Case of
     : (Form event=On Load)
          ` Initialize the array (as shown further above)
        ARRAY TEXT (atNames;5)
          ` ...
          ` Initialize the element zero with the number
          ` of the current selected element in its string form
          ` Here you start with no selected element
        atNames{0}:="0"

     : (Form event=On Unload)
          ` We no longer need the array
        CLEAR VARIABLE(atNames)

     : (Form event=On Clicked)
        If (atNames#0)
           If (atNames#Num(atNames{0}))
              vtInfo:="You clicked on: "+atNames{atNames}
                                             +" and it was not selected before."
              atNames{0}:=String(atNames)
           End if
        End if
     : (Form event=On Double Clicked)
        If (atNames#0)
           ALERT ("You double clicked on: "+atNames{atNames}
        End if
  End case
```

2. When sending or receiving a stream of characters to or from a document or a serial port, 4D provides a way to filter ASCII codes between platforms and systems that use different ASCII maps— the commands USE ASCII MAP, Mac to ISO, ISO to Mac, Mac to Win and Win to Mac.

In certain cases, you might want to fully control the way ASCII codes are translated. One way to do this is to use an Integer array of 255 elements, where the Nth element is set to the translated ASCII code for the character whose source ASCII code is N. For example, if the ASCII code #187 must be translated as #156, you would write <>aiCustomOutMap{187}:=156 and <>aiCustomInMap{156}:=187 in the method that initializes the interprocess arrays used everywhere in the database. You can then send a stream of characters with the following custom project method:

```
   ` X SEND PACKET ( Text { ; Time } )
For ($vlChar;1;Length($1))
   $1[[vlChar]]:=Char(<>aiCustomOutMap{Ascii($1[[vlChar]])})
End for
If (Count parameters>=2)
   SEND PACKET ($2;$1)
Else
   SEND PACKET ($1)
End if

   ` X Receive packet ( Text { ; Time } ) -> Text
If (Count parameters>=2)
   RECEIVE PACKET ($2;$1)
Else
   RECEIVE PACKET ($1)
End if
$0:=$1
For ($vlChar;1;Length($1))
   $0[[vlChar]]:=Char(<>aiCustomInMap{Ascii($0[[vlChar]])})
End for
```

In this advanced example, if a stream of characters containing NULL characters (ASCII code zero) is sent or received, the zero element of the arrays <>aiCustomOutMap and <>aiCustomInMap will play its role as any other element of the 255 element arrays.

See Also

Arrays.

Two-dimensional Arrays

version 6.0

Each of the array declaration commands can create or resize one-dimensional or two-dimensional arrays. Example:

```
    ` Creates a text array composed of 100 rows of 50 columns
ARRAY TEXT (atTopics;100;50)
```

Two-dimensional arrays are essentially language objects; you can neither display nor print them.

In the previous example:
• atTopics is a two-dimensional array
• atTopics{8}{5} is the 5th element (5th column...) of the 8th row
• atTopics{20} is the 20th row and is itself a one-dimensional array
• Size of array(atTopics) returns 100, which is the number of rows
• Size of array(atTopics{17}) returns 50, which the number of columns for the 17th row

In the following example, a pointer to each field of each table in the database is stored in a two-dimensional array:

```
    ` Create as many initially empty rows as tables
ARRAY POINTER (<>apFields;Count tables;0)
    ` For each table
For ($vlTable;1;Size of array(<>apFields))
        ` Resize the row with as many columns as fields in the table
    INSERT ELEMENT (<>apFields{$vlTable};1;Count fields($vlTable))
        ` Set the values of the elements
    For ($vlField;1;Size of array(<>apFields{$vlTable}))
        <>apFields{$vlTable}{$vlField}:=Field($vlTable;$vlField)
    End for
End for
```

Provided that this two-dimensional array has been initialized, you can obtain the pointers to the fields for a particular table in the following way:

```
    ` Get the pointers to the fields for the table currently displayed at the screen:
COPY ARRAY (<>apFields{Table(Current form table)};$apTheFieldsIamWorkingOn)
    ` Initialize Boolean and Date fields
For ($vlElem;1;Size of array($apTheFieldsIamWorkingOn))
    Case of
        : (Type($apTheFieldsIamWorkingOn{$vlElem}->)=Is Date)
            $apTheFieldsIamWorkingOn{$vlElem}->:=Current date
        : (Type($apTheFieldsIamWorkingOn{$vlElem}->)=Is Boolean)
            $apTheFieldsIamWorkingOn{$vlElem}->:=True
    End case
End for
```

Note: As this example suggests, rows of a two-dimensional arrays can be the same size or different sizes.

See Also

Arrays.

Arrays and Memory

Unlike the data you store on disk using tables and records, an array is always held in memory in its entirety.

For example, if all US zip codes were entered in the [Zip Codes] table, it would contain about 100,000 records. In addition, that table would include several fields: the zip code itself and the corresponding city, county, and state. If you select only the zip codes from California, the 4D database engine creates the corresponding selection of records within the [Zip Codes] table, and then loads the records only when they are needed (i.e., when they are displayed or printed). In order words, you work with an ordered series of values (of the same type for each field) that is partially loaded from the disk into the memory by the database engine of 4D.

Doing the same thing with arrays would be prohibitive for the following reasons:

• In order to maintain the four information types (zip code, city, county, state), you would have to maintain four large arrays in memory.
• Because an array is always held in memory in its entirety, you would have to keep all the zip codes information in memory throughout the whole working session, even though the data is not always in use.
• Again, because an array is always held in memory in its entirety, each time the database is started and then quit, the four arrays would have to be loaded and then saved on the disk, even though the data is not used or modified during the working session.

Conclusion: Arrays are intended to hold reasonable amounts of data for a short period of time. On the other hand, because arrays are held in memory, they are easy to handle and quick to manipulate.

However, in some circumstances, you may need to work with arrays holding hundreds or thousands of elements. The following table lists the formulas used to calculate the amount of memory used for each array type:

Array Type	**Formula for determining Memory Usage in Bytes**
Boolean	(31+number of elements)\8
Date	(1+number of elements) * 6
String	(1+number of elements) * Declared length (+1 of odd, +2 if even)
Integer	(1+number of elements) * 2
Long Integer	(1+number of elements) * 4
Picture	(1+number of elements) * 4 + Sum of the size of each picture
Pointer	(1+number of elements) * 16
Real	(1+number of elements) * 8 (Windows, PPC) or * 10 (68K)
Text	(1+number of elements) * 6 + Sum of the size of each text
Two-dimemsional	(1+number of elements) * 12 + Sum of the size of each array

Note: A few additional bytes are required to keep track of the selected element, the number of elements, and the array itself.

When working with very large arrays, the best way to handle full memory situations is to surround the creation of the arrays with an ON ERR CALL project method. Example:

```
    ` You are going to run a batch operation the whole night
    ` that requires the creation of large arrays. Instead of risking
    ` occurrences of errors in the middle of the night, put
    ` the creation of the arrays at the beginning of the operation
    ` and test the errors at this moment:
gError:=0 ` Assume no error
ON ERR CALL ("ERROR HANDLING") ` Install a method for catching errors
ARRAY STRING (63;asThisArray;50000) ` Roughly 3125K
ARRAY REAL (arThisAnotherArray;50000) ` 488K
ON ERR CALL ("") ` No longer need to catch errors
If (gError=0)
    ` The arrays could be created
    ` and let's pursue the operation
Else
    ALERT ("This operation requires more memory!")
End if
    ` Whatever the case, we no longer need the arrays
CLEAR VARIABLE (asThisArray)
CLEAR VARIABLE (arThisAnotherArray)
```

The ERROR HANDLING project method is listed here:

```
    ` ERROR HANDLING project method
gError:=Error ` Jusrt return the error code
```

See Also

Arrays, ON ERR CALL.

ARRAY INTEGER

version 3

ARRAY INTEGER (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY INTEGER creates and/or resizes an array of 2-byte Integer elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY INTEGER to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to 0.
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 2-byte Integer elements:

```
⇒      ARRAY INTEGER (aiValues;100)
```

2. This example creates a local array of 100 rows of 50 2-byte Integer elements:

```
⇒      ARRAY INTEGER ($aiValues;100;50)
```

3. This example creates an interprocess array of 50 2-byte Integer elements, and sets each element to its element number:

```
⇒      ARRAY INTEGER (<>aiValues;50)
       For ($vlElem;1;50)
          <>aiValues{$vlElem}:=$vlElem
       End for
```

ARRAY LONGINT

ARRAY LONGINT (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY LONGINT creates and/or resizes an array of 4-byte Long Integer elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

When applying ARRAY LONGINT to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to 0.
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 4-byte Long Integer elements:

```
⇒      ARRAY LONGINT (alValues;100)
```

2. This example creates a local array of 100 rows of 50 4-byte Long Integer elements:

```
⇒      ARRAY LONGINT ($alValues;100;50)
```

3. This example creates an interprocess array of 50 4-byte Long Integer elements and sets each element to its element number:

```
⇒      ARRAY LONGINT (<>alValues;50)
       For ($vlElem;1;50)
           <>alValues{$vlElem}:=$vlElem
       End for
```

ARRAY REAL

ARRAY REAL (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY REAL creates and/or resizes an array of Real elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY REAL to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to 0.
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 Real elements:

```
⇒       ARRAY REAL (arValues;100)
```

2. This example creates a local array of 100 rows of 50 Real elements:

```
⇒       ARRAY REAL ($arValues;100;50)
```

3. This example creates an interprocess array of 50 Real elements and sets each element to its element number:

```
⇒       ARRAY REAL (<>arValues;50)
        For ($vlElem;1;50)
            <>arValues{$vlElem}:=$vlElem
        End for
```

ARRAY STRING

ARRAY STRING (strLen; arrayName; size{; size2})

Parameter	Type		Description
strLen	Number	→	Length of string (1... 255)
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY STRING creates and/or resizes an array of String elements in memory.

• The strLen parameter specifies the maximum number of characters that can be contained in each array element in a string array. The length can be from 1 to 255 characters.
• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY STRING to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to "" (empty string).
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 31-character String elements:

```
⇒    ARRAY STRING (31;asValues;100)
```

2. This example creates a local array of 100 rows of 50 63-character String elements:

```
⇒    ARRAY STRING (63;$asValues;100;50)
```

3. This example creates an interprocess array of 50 255-character String elements and sets each element to the value “Element #” followed by its element number:

```
⇒   ARRAY STRING (255;<>asValues;50)
    For ($vlElem;1;50)
        <>asValues{$vlElem}:="Element #"+String($vlElem)
    End for
```

ARRAY TEXT

version 3

ARRAY TEXT (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number		
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY TEXT creates and/or resizes an array of Text elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY TEXT to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to "" (empty string).
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 Text elements:

```
⇒      ARRAY TEXT (atValues;100)
```

2. This example creates a local array of 100 rows of 50 Text elements:

```
⇒      ARRAY TEXT ($atValues;100;50)
```

3. This example creates an interprocess array of 50 Text elements and sets each element to the value “Element #” followed by its element number:

```
⇒      ARRAY TEXT (<>atValues;50)
       For ($vlElem;1;50)
           <>atValues{$vlElem}:="Element #"+String($vlElem)
       End for
```

ARRAY DATE

ARRAY DATE (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY DATE creates and/or resizes an array of Date elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY DATE to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to the null date (!00/00/00!).
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 Date elements:

```
⇒	ARRAY DATE (adValues;100)
```

2. This example creates a local array of 100 rows of 50 Date elements:

```
⇒	ARRAY DATE ($adValues;100;50)
```

3. This example creates an interprocess array of 50 Date elements, and sets each element to the current date plus a number of days equal to the element number:

```
⇒	ARRAY DATE (<>adValues;50)
	For ($vlElem;1;50)
		<>adValues{$vlElem}:=Current date+$vlElem
	End for
```

ARRAY BOOLEAN

version 3

ARRAY BOOLEAN (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY BOOLEAN creates and/or resizes an array of Boolean elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY BOOLEAN to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to False.
• If you reduce the array size, the last elements deleted from the array are lost.

Tip: In some contexts, an alternative to using Boolean arrays is using an Integer array where each element “means true” if different from zero and “means false” if equal to zero.

Examples

1. This example creates a process array of 100 Boolean elements:

```
⇒	ARRAY BOOLEAN (abValues;100)
```

2. This example creates a local array of 100 rows of 50 Boolean elements:

```
⇒	ARRAY BOOLEAN ($abValues;100;50)
```

3. This example creates an interprocess array of 50 Boolean elements and sets each even element to True:

```
⇒     ARRAY BOOLEAN (<>abValues;100)
      For ($vlElem;1;50)
          <>abValues{$vlElem}:=(($vlElem%2)=0)
      End for
```

ARRAY PICTURE

version 3

ARRAY PICTURE (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array, or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY PICTURE creates and/or resizes an array of Picture elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the first dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY PICTURE to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to empty pictures. This means that Picture size applied to one of these elements will return 0.
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 Picture elements:

```
⇒	ARRAY PICTURE (agValues;100)
```

2. This example creates a local array of 100 rows of 50 Picture elements:

```
⇒	ARRAY PICTURE ($agValues;100;50)
```

3. This example creates an interprocess array of Picture elements and loads each picture into one of the elements of the array. The array's size is equal to the number of 'PICT' resources available to the database. The array's resource name starts with "User Intf/":

```
   RESOURCE LIST("PICT";$aiResIDs;$asResNames)
⇒  ARRAY PICTURE (<>agValues;Size of array($aiResIDs))
   $vlPictElem:=0
   For ($vlElem;1;Size of array(<>agValues))
      If ($asResNames="User Intf/@")
         $vlPictElem:=vlPictElem+1
         GET PICTURE RESOURCE("PICT";$aiResIDs{$vlElem};$vgPicture)
         <>agValues{$vlPictElem}:=$vgPicture
      End if
   End for
   ARRAY PICTURE (<>agValues;$vlPictElem)
```

ARRAY POINTER

ARRAY POINTER (arrayName; size{; size2})

Parameter	Type		Description
arrayName	Array	→	Name of the array
size	Number	→	Number of elements in the array, or Number of rows if size2 is specified
size2	Number	→	Number of columns in a two-dimensional array

Description

The command ARRAY POINTER creates or resizes an array of Pointer elements in memory.

• The arrayName parameter is the name of the array.
• The size parameter is the number of elements in the array.
• The size2 parameter is optional; if size2 is specified, the command creates a two-dimensional array. In this case, size specifies the number of rows and size2 specifies the number of columns in each array. Each row in a two-dimensional array can be treated as both an element and an array. This means that while working with the firt dimension of the array, you can use other array commands to insert and delete entire arrays in a two-dimensional array.

While applying ARRAY POINTER to an existing array:

• If you enlarge the array size, the existing elements are left unchanged, and the new elements are initialized to null pointer. This means that Nil applied to one of these elements will return True.
• If you reduce the array size, the last elements deleted from the array are lost.

Examples

1. This example creates a process array of 100 Pointer elements:

```
⇒	ARRAY POINTER (apValues;100)
```

2. This example creates a local array of 100 rows of 50 Pointer elements:

```
⇒	ARRAY POINTER ($apValues;100;50)
```

3. This example creates an interprocess array of Pointer elements and sets each element pointing to the table whose number is the same as the element. The size of the array is equal to the number of tables in the database:

```
⇒    ARRAY POINTER (<>apValues;Count tables)
     For ($vlElem;1;Size of array(<>apValues))
        <>apValues{$vlElem}:=Table($vlElem)
     End for
```

Size of array

Size of array (array) → Number

Parameter	Type		Description
array	Array	→	Array whose size is returned
Function result	Number	←	Returns the number of elements in array

Description

The command Size of array returns the number of elements in array.

Example

1. The following example returns the size of the array anArray:

```
⇒	vlSize:=Size of array(anArray)	` vlSize gets the size of anArray
```

2. The following example returns the number of rows in a two-dimensional array:

```
⇒	vlRows:=Size of array(a2DArray)	` vlRows gets the size of a2DArray
```

3. The following example returns the number of columns for a row in a two-dimensional array:

```
⇒	vlColumns:=Size of array(a2DArray{10})	` vlColumns gets the size of a2DArray{10}
```

See Also

DELETE ELEMENT, INSERT ELEMENT.

SORT ARRAY

SORT ARRAY (array{; array2; ...; arrayN}{; > or <})

Parameter	Type		Description
array	Array	→	Arrays to sort
> or <		→	> to sort in Ascending order, or < to sort in Descending order, or Ascending order if omitted

Description

The command SORT ARRAY sorts one or more arrays into ascending or descending order.

Note: You cannot sort Pointer or Picture arrays. You can sort the elements of a two-dimensional array (i.e., a2DArray{$vlThisElem}) but you cannot sort the two-dimensional array itself (i.e., a2DArray).

The last parameter specifies whether to sort array in ascending or descending order. The "greater than" symbol (>) indicates an ascending sort; the "less than" symbol (<) indicates a descending sort. If you do not specify the sorting order, then the sort is ascending.

If more than one array is specified, the arrays are sorted following the sort order of the first array; no multi-level sorting is performed here. This feature is especially useful with grouped scrollable areas in a form; SORT ARRAY maintains the synchronicity of the arrays that sustain the scrollable areas.

Examples

1. The following example creates two arrays and then sorts them by company:

```
	ALL RECORDS ([People])
	SELECTION TO ARRAY ([People]Name;asNames;[People]Company;asCompanies)
⇒	SORT ARRAY (asCompanies; asNames;>)
```

However, because SORT ARRAY does not perform multi-level sorts, you will end up with people's names in random order within each company. To sort people by name within each company, you would write:

```
	ALL RECORDS ([People])
	ORDER BY ([People];[People]Company;>;[People]Name;>)
	SELECTION TO ARRAY ([People]Name;asNames;[People]Company;asCompanies)
```

2. You display the names from a [People] table in a floating window. When you click on buttons present in the window, you can sort this list of names from A to Z or from Z to A . As several people may have the same name, you also can use a [People]ID number field, which is indexed unique. When you click in the list of names, you will retrieve the record for the name you clicked. By maintaing a synchronized and hidden array of ID numbers, you are sure to access the record corresponding to the name you clicked:

```
          ` asNames array object method
      Case of
          : (Form event=On Load)
             ALL RECORDS([People])
             SELECTION TO ARRAY([People]Name;asNames;[People]ID number;alIDs)
⇒            SORT ARRAY(asNames;alIDs;>)
          : (Form event=On Unload)
             CLEAR VARIABLE(asNames)
             CLEAR VARIABLE(alIDs)
          : (Form event=On Clicked)
             If (asNames#0)
                   ` Use the array alIDs to get the right record
                QUERY([People];[People]ID Number=alIDs{asNames})
                   ` Do something with the record
             End if
      End case

          ` bA2Z button object method
          ` Sort the arrays in ascending order and keep them synchronized
⇒     SORT ARRAY(asNames;alIDs;>)

          ` bZ2A button object method
          ` Sort the arrays in descending order and keep them synchronized
⇒     SORT ARRAY(asNames;alIDs;<)
```

See Also

ORDER BY, SELECTION TO ARRAY.

Find in array

Find in array (array; value{; start}) → Number

Parameter	Type		Description
array	Array	→	Array to search
value	Expression	→	Value of same type to search in the array
start	Number	→	Element at which to start searching
Function result	Number	←	Number of the first element in array that matches value

Description

The command Find in array returns the number of the first element in array that matches value.

Find in array can be used with Text, String, Numeric, Date, Pointer, and Boolean arrays. The array and value parameters must be of the same type.

If no match is found, Find in array returns –1.

If start is specified, the command starts searching at the element number specified by start. If start is not specified, the command starts searching at element 1.

Examples

1. The following project method deletes all empty elements from the string or text array whose pointer is passed as parameter:

```
         ` CLEAN UP ARRAY project method
         ` CLEAN UP ARRAY ( Pointer )
         ` CLEAN UP ARRAY ( -> Text or String array )

      C_POINTER ($1)
      Repeat
⇒        $vlElem:=Find in array ($1->;"")
         If ($vlElem>0)
            DELETE ELEMENT ($1->;$vlElem)
         End if
      Until ($vlElem<0)
```

After this project method is implemented in a database, you can write:

```
ARRAY TEXT (atSomeValues;...)
   ` ...
   ` Do plenty of things with the array
   ` ...
   ` Eliminate empty string elements
CLEAN UP ARRAY (->atSomeValues)
```

2. The following project method selects the first element of an array whose pointer is passed as the first parameter that matches the value of the variable or field whose pointer is passed as parameter:

```
   ` SELECT ELEMENT project method
   ` SELECT ELEMENT ( Pointer ; Pointer)
   ` SELECT ELEMENT ( -> Text or String array ; -> Text or String variable or field )

⇒  $1->:=Find in array ($1->;$2->)
   If ($1->=-1)
      $1->:=0 ` If no element was found, set the array to no selected element
   End if
```

After this project method is implemented in a database, you can write:

```
   ` asGender pop-up menu object method
Case of
   : (Form Event=On Load)
      SELECT ELEMENT (->asGender;->[People]Gender)

End case
```

See Also

DELETE ELEMENT, INSERT ELEMENT, Size of array.

INSERT ELEMENT

version 3

INSERT ELEMENT (array; where{; howMany})

Parameter	Type		Description
array	Array	→	Name of the array
where	Number	→	Where to insert the elements
howMany	Number	→	Number of elements to be inserted, or 1 element if omitted

Description

The command INSERT ELEMENT inserts one or more elements into the array array. The new elements are inserted before the element specified by where, and are initialized to the empty value for the array type. All elements beyond where are consequently moved within the array by an offset of one or the value you pass in howMany.

If where is greater than the size of the array, the elements are added to the end of the array.

The howMany parameter is the number of elements to insert. If howMany is not specified, then one element is inserted. The size of the array grows by howMany.

Example

1. The following example inserts five new elements, starting at element 10:

```
⇒	INSERT ELEMENT (anArray;10;5)
```

2. The following example appends an element to an array:

```
	$vlElem:=Size of array(anArray)+1
⇒	INSERT ELEMENT (anArray;$vlElem)
	anArray{$vlElem}:=...
```

See Also

DELETE ELEMENT, Size of array.

DELETE ELEMENT

Arrays

version 3

DELETE ELEMENT (array; where{; howMany})

Parameter	Type		Description
array	Array	→	Array from which to delete elements
where	Number	→	Element at which to begin deletion
howMany	Number	→	Number of elements to delete, or 1 element if omitted

Description

The command DELETE ELEMENT deletes one or more elements from array. Elements are deleted starting at the element specified by where.

The howMany parameter is the number of elements to delete. If howMany is not specified, then one element is deleted. The size of the array shrinks by howMany.

Examples

1. The following example deletes three elements, starting at element 5:

```
⇒      DELETE ELEMENT (anArray; 5; 3)
```

2. The following example deletes the last element from an array, if it exists:

```
       $vlElem:=Size of array(anArray)
       If ($vlElem>0)
⇒          DELETE ELEMENT (anArray;$vlElem)
       End if
```

See Also

INSERT ELEMENT, Size of array.

COPY ARRAY

COPY ARRAY (source; destination)

Parameter	Type		Description
source	Array	→	Array from which to copy
destination	Array	←	Array to which to copy

Description

The command COPY ARRAY creates or overwrites the destination array destination with the exact contents, size, and type of the source array source.

The source and destination arrays can be local, process, or interprocess arrays. When copying arrays, the scope of the array does not matter.

Examples

The following example fills the array named C. It then creates a new array, named D, of the same size as C and with the same contents:

```
         ALL RECORDS ([People])  ` Select all records in People
         SELECTION TO ARRAY ([People]Company; C)  ` Move company field data into array C
⇒        COPY ARRAY (C; D)  ` Copy the array C to the array D
```

LIST TO ARRAY

version 3

Compatibility Note

Due to the new implementation of Choice Lists, compatibility for this command could not be fully maintained. Also, starting with version 6, we recommend that you start using the command Load list to work with the hierarchical lists defined in the Design environment List Editor.

LIST TO ARRAY (list; array{; itemRefs})

Parameter	Type		Description
list	String	→	List from which to copy the first level items
array	Array	←	Array to which to copy the list items
itemRefs	Array	←	List item reference numbers

Description

The command LIST TO ARRAY creates or overrides the array array with the first level items of the list list.

If you have not previously defined the array as a string or text array, LIST TO ARRAY creates a text array by default.

The optional itemRefs parameter (a numeric array) returns the list item reference numbers.

Compatibility Note: In the previous version of 4D, this array was filled with the names of any linked lists. If an element of the list had a linked list, the name of the linked list was put into the array element with the same number as the list element. If there was no linked list, then the element was the empty string. The second array was set to the same size as array. You could use the names in this array to access the linked lists.

You can continue to use LIST TO ARRAY to build an array based on the first level items of a hierarchical list. However, this command does not provide you with the child items, if any. To work with hierarchical lists, use the new Hierarchical Lists commands introduced in version 6.

Example

The following example copies the items of a list called Regions into an array called atRegions:

```
⇒	LIST TO ARRAY ("Regions"; atRegions )
```

See Also

ARRAY TO LIST, Load list, SAVE LIST.

ARRAY TO LIST

version 3

Compatibility Note
Due to the new implementation of Choice Lists, compatibility for this command could not be fully maintained. Also, starting with version 6, we recommend that you use the command SAVE LIST to work with the hierarchical lists defined in the Design environment List Editor.

ARRAY TO LIST (array; list{; itemRefs})

Parameter	Type		Description
array	Array	→	Array from which to copy array elements
list	String	→	List into which to copy array elements
itemRefs	Array	→	Numeric array of item reference numbers

Description

The command ARRAY TO LIST creates or replaces the list list (as defined in the Design environment List Editor) using the elements of the array array.

This command allows you to define only the first level items of the list.

The optional itemRefs parameter, if specified, must be a numeric array synchronized with the array array. Each element, then, indicates the list item reference number for the corresponding element in array. If you omit this parameter, 4D automatically sets the list item reference numbers to 1, 2... N.

Compatibility Note: In the previous version of 4D, this parameter was used to link other lists to each element in array. If an element of the links array was the name of an existing list, then that list was attached to the corresponding item.

You can continue to use ARRAY TO LIST to build a list based on the elements of an array. However, this command does not provide a means of working with the child items. To work with hierarchical lists, use the new Hierarchical Lists commands introduced in version 6.

Example

The following example copies the array atRegions to the list called “Regions:”

```
⇒	ARRAY TO LIST (atRegions;"Regions")
```

See Also

LIST TO ARRAY, Load list, ON ERR CALL, SAVE LIST.

Error Handling

An error -9957 is generated when ARRAY TO LIST is applied to a list that is currently being edited in the Design environment List Editor. You can catch this error using an ON ERR CALL project method.

SELECTION TO ARRAY

SELECTION TO ARRAY (field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})

Parameter	Type		Description
field | table	Field or Table	→	Field to use for retrieving data or Table to use for retrieving record numbers
array	Array	←	Array to receive field data or record numbers

Description

The command SELECTION TO ARRAY creates one or more arrays and copies data in the fields or record numbers from the current selection into the arrays.

The command SELECTION TO ARRAY applies to the selection for the table specified in the first parameter. SELECTION TO ARRAY, can perform the following:

• Load values from one or several fields.
• Load Record numbers using the syntax ...;[table];Array;...
• Load values from related fields, provided that there is a Many to One automatic relation between the tables or provided that you have previously called AUTOMATIC RELATIONS to make manual Many to One relations automatic. In both cases, values are loaded from tables through several levels of Many to One relations.

Each array is typed according to the field type. There are two exceptions:

• If a Text field is copied into a String array, the array will remain a String array.
• A Time field is copied into a Long Integer array.

Note: You cannot specify Subtable fields or subfields.

If you load record numbers, they are copied into a Long Integer array.

4D Server: The SUBSELECTION TO ARRAY command is optimized for 4D Server. Each array is created on the server and then sent, in its entirety, to the client machine.

WARNING: The SELECTION TO ARRAY command can create large arrays, depending on the range you specify in start and end, and on the type and size of the data you are loading. Arrays reside in memory, so it is a good idea to test the result after the command is completed. To do so, test the size of each resulting array or cover the call to the command, using an ON ERR CALL project method.

Note: After a call to SELECTION TO ARRAY, the current selection and current record remain the same, but the current record is no longer loaded. If you need to use the values of the fields in the current record, use the LOAD RECORD command after the SELECTION TO ARRAY command.

Examples

1. In the following example, the [People] table has an automatic relation to the [Company] table. The two arrays asLastName and asCompanyAddr are sized according to the number of records selected in the [People] table and will contain information from both tables:

⇒ **SELECTION TO ARRAY** ([People]Last Name;asLastName;[Company]Address;asCompanyAddr)

2. The following example returns the [Clients] record numbers in the array alRecordNumbers and the [Clients]Names field values in the array asNames:

⇒ **SELECTION TO ARRAY**([Clients];alRecordNumbers;[Clients]Names; asNames)

See Also

ARRAY TO SELECTION, AUTOMATIC RELATIONS, ON ERR CALL, SUBSELECTION TO ARRAY.

SUBSELECTION TO ARRAY

version 3.5.3

SUBSELECTION TO ARRAY (start; end; field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})

Parameter	Type		Description
start	Number	→	Selected record number where data retrieval starts
end	Number	→	Selected record number where data retrieval ends
field | table	Field or Table	→	Field to use for retrieving data or Table to use for retrieving record numbers
array	Array	←	Array to receive field data or record numbers

Description

SUBSELECTION TO ARRAY creates one or more arrays and copies data from the fields or record numbers from the current selection into the arrays.

Unlike SELECTION TO ARRAY, which applies to the current selection in its entirety, SUBSELECTION TO ARRAY only applies to the range of selected records specified by the parameters start and end.

The command expects you to pass in start and end the selected record numbers complying with the formula 1 <= start <= end <= Records in selection ([...]).

If you pass 1 <= start = end < Records in selection ([...]), you will load fields or get the record number from the record whose selected record is start = end.

If you pass incorrect selected record numbers, the command does the following:

• If end > Records in selection ([...]), it returns values from the selected record specified by start to the last selected record.
• If start > end, it returns values from the record whose selected record is start only.
• If both parameters are inconsistent with the size of the selection, it returns empty arrays.

Like SELECTION TO ARRAY, the command SUBSELECTION TO ARRAY applies to the selection for the table specified in the first parameter.

SUBSELECTION TO ARRAY, like SELECTION TO ARRAY, can perform the following:

• Load values from one or several fields.
• Load Record numbers using the syntax ...;[table];Array;...
• Load values from related fields, if there is a Many to One automatic relation between the tables or if you have previously called AUTOMATIC RELATIONS to change manual Many to One relations to automatic. In both cases, values can be loaded from tables through several levels of Many to One relations.

Each array is typed according to the field type. There are two exceptions:

• If a Text field is copied into a String array. In this case, the array will remain a String array.
• A Time field is copied into a Long Integer array.

Note: You cannot specify Subtable fields or subfields.

If you load record numbers, they are copied into a Long Integer array.

4D Server: The SUBSELECTION TO ARRAY command is optimized for 4D Server. Each array is created on the server and then sent, in its entirety, to the client machine.

WARNING: The SUBSELECTION TO ARRAY command can create large arrays, depending on the range you specify in start and end, and on the type and size of the data you are loading. Arrays reside in memory, so it is a good idea to test the result after the command is completed. To do so, test the size of each resulting array or cover the call to the command, using an ON ERR CALL project method.

If the command is successful, the size of each resulting array is equal to (end-start)+1, except if the end parameter exceeded the number of records in the selection. In such a case, each resulting array contains (Records in selection([...])-start)+1 elements.

Examples

1. The following line of code addresses the first 50 records from the current selection for the [Invoices] table. It loads the values from the [Invoices]Invoice ID field and the [Customers]Customer ID related field.

```
⇒       SUBSELECTION TO ARRAY(1;50;[Invoices]Invoice ID;
                                        alInvoID;[Customers]Customer ID;alCustID)
```

2. The following lines of code address the last 50 records from the current selection for [Invoices] table. It loads the record numbers of the [Invoices] records as well as those of the [Customers] related records:

```
        lSelSize := Records in selection ([Invoices])
⇒       SUBSELECTION TO ARRAY (lSelSize-49;lSelSize;[Invoices];
                                        alInvRecN;[Customers];alCustRecN)
```

3. The following lines of code process, in sequential “chunks”of 1000 records, a large selection that could not be downloaded in its entirety into arrays:

```
        lMaxPage := 1000
        lSelSize := Records in selection ([Phone Directory])
        For ($lPage ; 1; 1+((lSelSize-1)\lMaxPage) )
                ` Load the values and/or record numbers
⇒           SUBSELECTION TO ARRAY (1+(lMaxPage*($lPage-
1));lMaxPage*$lPage;...;...;...;...;...;...)
                ` Do something with the arrays
        End for
```

See Also

AUTOMATIC RELATIONS, ON ERR CALL, SELECTION TO ARRAY.

ARRAY TO SELECTION

ARRAY TO SELECTION (array; field{; array2; field2; ...; arrayN; fieldN})

Parameter	Type		Description
array	Array	→	Array to copy to the selection
field	Field	←	Field to receive the array data

Description

The command ARRAY TO SELECTION copies one or more arrays into a selection of records. All fields listed must belong to the same table.

If a selection exists at the time of the call, the elements of the array are put into the records, based on the order of the array and the order of the records. If there are more elements than records, new records are created. The records, whether new or existing, are automatically saved.

If the arrays are of different sizes, the first array is used to determine how many elements to copy. Any additional arrays are moved into the field that follows each array name.

This command does the reverse of SELECTION TO ARRAY. However, the ARRAY TO SELECTION command does not allow fields from different tables, including related tables, even when an automatic relation exists.

WARNING: Use ARRAY TO SELECTION with caution, because it overwrites information in existing records. If a record is locked by another process during the execution of ARRAY TO SELECTION, that record is not modified. Any locked records are put into the process set called LockedSet. After ARRAY TO SELECTION has executed, you can test the set LockedSet to see if any records were locked.

4D Server: The command is optimized for 4D Server. Arrays are sent by the client machine to the server, and the records are modified or created on the server machine. As such a request is handled synchronously, the client machine must wait for the operation to be completed successfully. In the multi-user or multi-process environment, any records that are locked will not be overwritten.

Example

In the following example, the two arrays asLastNames and asCompanies place data in the [People] table. The values from the array asLastNames area placed in the field [People]Last Name and the values from the array asCompanies are placed in the field [People]Company:

⇒ **ARRAY TO SELECTION** (asLastNames;[People]Last Name;asCompanies;[People]Company)

See Also

SELECTION TO ARRAY.

DISTINCT VALUES

version 6.0 (Modified)

DISTINCT VALUES (field; array)

Parameter	Type		Description
field	Field or Subfield	→	Field or subfield to use for data
array	Array	←	Array to receive indexed field data

Description

The command DISTINCT VALUES creates and populates the array array with non-repeated (unique) values coming from the field field for the current selection of the table to which the field or subfield belongs.

In the previous version of 4D, you could only pass alphanumeric fields to this command. Starting with version 6, you can pass any indexed field or subfield. Note however that the command does nothing when applied to an indexed Boolean field.

If you pass the field of a table, DISTINCT VALUES browses and retains the non-repeated values present only in the currently selected records. However, if you pass a subfield, DISTINCT VALUES browses all the subrecords present in each currently selected record.

If you create the array prior to the call, DISTINCT VALUES expects an array type compatible with the field or subfield you pass. Otherwise, in interpreted mode, DISTINCT VALUES will create an array of the proper type. However, if the field or subfield is of type Time, the command expects or creates a LongInt array.

After the call, the size of the array is equal to the number of distinct values found in the selection. The command does not change the current selection or the current record. The DISTINCT VALUES command uses the index of the field, so the elements in array are returned sorted in ascending order. If this is the order you need, you do not need to call SORT ARRAY after using DISTINCT VALUES.

WARNING: DISTINCT VALUES can create large arrays depending on the size of the selection and the number of different values in the records. Arrays reside in memory, therefore it is a good idea to test the result after the completion of the command. To do so, test the size of the resulting array or cover the call to the command, using an ON ERR CALL project method.

4D Server: The command is optimized for 4D Server. The array is created and the values are calculated on the server machine; the array is then sent, in its entirety, to the client.

Examples

1. The following example creates a list of cities from the current selection and tells the user the number of cities in which the firm has stores:

```
	ALL RECORDS([Retail Outlets]) ` Create a selection of records
⇒	DISTINCT VALUES([Retail Outlets]City;asCities)
	ALERT("The firm has stores in " +String(Size of array(asCities))+" cities.")
```

2. The following example returns in asKeywords all the keywords that are attached (using a subtable) to the 4D Write documents stored in the table [Documentation] and whose theme is “Economy”:

```
	QUERY ([Documentation];[Documentation]Theme="Economy")
⇒	DISTINCT VALUES([Documentation]Keywords'Keyword;asKeywords)
```

After this array has been built, you can reuse it to quickly locate all the documents associated with the selected keyword:

```
	QUERY ([Documentation];[Documentation]Keywords'Keyword=
							asKeywords{asKeywords})
	SELECTION TO ARRAY ([Documentation]Subject;asSubjects)
	   ` ...
```

See Also

ON ERR CALL, SELECTION TO ARRAY, SUBSELECTION TO ARRAY.

5 BLOB

BLOB Commands

BLOB

version 6.0

Definition

4th Dimension version 6 introduces the BLOB (Binary Large OBjects) data type.

You can define BLOB fields and BLOB variables:

• To create a BLOB field, select BLOB in the **Field type** drop-down-list within the **Field Properties** window.
• To create a BLOB variable, use the compiler declaration command **C_BLOB**. You can create local, process, and interprocess variables of type BLOB.

Note: There is no array for BLOBs.

Within 4th Dimension, a BLOB is a contiguous series of variable length bytes, which can be treated as one whole object or whose bytes can be addressed individually. A BLOB can be empty (null length) or can contain up to 2,147,483,647 bytes (2 GB).

BLOBs and Memory

A BLOB is loaded into memory in its entirety. A BLOB variable is held and exists in memory only. A BLOB field is loaded into memory from the disk, like the rest of the record to which it belongs.

Like the other field types that can retain a large amount of data (Picture and subtable field types), BLOB fields are not duplicated in memory when you modify a record. Consequently, the result returned by the commands **Old** and **Modified** is not significant when applied to a BLOB field.

Displaying BLOBs

A BLOB can retain any type of data, so it has no default representation on the screen. If you display a BLOB field or variable in a form, it will always appear blank, whatever its contents.

BLOB fields

You can use BLOB fields to store any kind of data, up to 2 GB. You cannot index a BLOB field, so you must use a formula in order to search records on values stored in a BLOB field. Do not use BLOB fields for storing data that you want to retrieve quickly with a search operation. For example, do not store keywords in a BLOB field; instead, use a subfile in which you can index the keyword subfield.

Parameter passing, Pointers and function results

4th Dimension BLOBs can be passed as parameters to 4D commands or 4D Extensions routines that expect a BLOB parameters. On the other hand, they cannot be passed as parameters to a user method. A BLOB cannot be returned as a function result.

To pass a BLOB to your own methods, define a pointer to the BLOB and pass the pointer as parameter.

Examples:

```
    ` Declare a variable of type BLOB
C_BLOB (anyBlobVar)
    ` The BLOB is passed as parameter to a 4D command
SET BLOB SIZE (anyBlobVar;1024*1024)
    ` The BLOB is passed as parameter to an external routine
$errCode:= Do Something With This BLOB  (anyBlobVar)
    ` A pointer to the BLOB is passed as parameter to a user method
COMPUTE BLOB (->anyBlobVar )
    ` Declare a variable of type Pointer
C_POINTER (aPointer)
    ` Define a pointer to the BLOB
aPointer := ->anyBlobVar
    ` A pointer to the BLOB is passed as parameter to a user method
COMPUTE BLOB (aPointer)
```

Note for 4D Extensions developers: A BLOB parameter is declared as “&O” (the letter “O”, not the digit “0”).

Assignment

You can assign BLOBs to each other.

Example:

```
    ` Declare two variables of type BLOB
C_BLOB (vBlobA;vBlobB)
    ` Set the size of the first BLOB to 10K
SET BLOB SIZE (vBlobA;10*1024)
    ` Assign the first BLOB to the second one
vBlobB:=vBlobA
```

However, no operator can be applied to BLOBs; there is no expression of type BLOB.

Addressing BLOB contents

You can address each byte of a BLOB individually using the curly brackets symbols {...}. Within a BLOB, bytes are numbered from 0 to N-1, where N is the size of the BLOB. Example:

```
    ` Declare a variable of type BLOB
C_BLOB (vBlob)
    ` Set the size of the BLOB to 256 bytes
SET BLOB SIZE (vBlob;256)
    ` The loop below initializes the 256 bytes of the BLOB to zero
For ( vByte ; 0 ; BLOB size (vBlob)-1)
    vBlob{vByte}:=0
End for
```

Because you can address all the bytes of a BLOB individually, you can actually store whatever you want in a BLOB field or variable.

BLOBs 4th Dimension commands

4th Dimension provides the following commands for working BLOBS:

• SET BLOB SIZE resizes a BLOB field or variable.
• BLOB size returns the size of a BLOB.
• DOCUMENT TO BLOB and BLOB TO DOCUMENT enable you to load and write a whole document to and from a BLOB (optionally, the data and resource forks on Macintosh).
• VARIABLE TO BLOB and BLOB TO VARIABLE as well as LIST TO BLOB and BLOB to list allow you to store and retrieve 4D variables in BLOBs.
• COMPRESS BLOB, EXPAND BLOB and BLOB PROPERTIES allow you to work with compressed BLOBs
• The commands BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB and TEXT TO BLOB enable you to manipulate any structured data coming from disk, resources, OS, and so on.
• DELETE FROM BLOB, INSERT IN BLOB and COPY BLOB allow quick handling of large chunks of data within BLOBs.

These commands are described in this chapter.

In addition:

• C_BLOB declares a variable of type BLOB. Refer to the Compiler chapter for more information.
• GET CLIPBOARD and APPEND CLIPBOARD enable you to deal with any data type stored in the Clipboard. Refer to the Clipboard chapter for more information.
• GET RESOURCE and SET RESOURCE enable you to work with any type stored of resource stored on disk. Refer to the Resources chapter for more information.

SET BLOB SIZE

BLOB

version 6.0

SET BLOB SIZE (blob; size{; filler})

Parameter	Type		Description
blob	BLOB	→	BLOB field or variable
size	Number	→	New size of the BLOB
filler	Number	→	ASCII code of filler character

Description

SET BLOB SIZE resizes the BLOB blob according to the value passed in size.

By default, new allocated bytes (if any) for the BLOB are initialized to 0x00. If you want to have those bytes initialized to another value, pass the value (0..255) into the optional filler parameter.

Examples

1. When you are through with a large process or interprocess BLOB, it is good idea to free the memory it occupies. To do so, write:

```
⇒     SET BLOB SIZE(aProcessBLOB;0)
⇒     SET BLOB SIZE(<>anInterprocessBLOB;0)
```

2. The following example creates a BLOB of 16K filled of 0xFF:

```
      C_BLOB(vxData)
⇒     SET BLOB SIZE(vxData;16*1024;0xFF)
```

See Also

BLOB size.

Error Handling

If you cannot resize a BLOB due to insufficient memory, the error -108 is generated. You can trap this error using an ON ERR CALL interruption method.

BLOB size

version 6.0

BLOB size (blob) → Number

Parameter	Type		Description
blob	BLOB	→	BLOB field or variable
Function result	Number	←	Size in bytes of the BLOB

Description

BLOB size returns the size of blob expressed in bytes.

Examples

The line of code adds 100 bytes to the BLOB myBlob:

```
⇒	SET BLOB SIZE (BLOB size(myBlob)+100)
```

See Also

SET BLOB SIZE.

COMPRESS BLOB

COMPRESS BLOB (blob{; compression})

Parameter	Type		Description
blob	BLOB	→	BLOB to compress
compression	Number	→	If not omitted: 1, compress as compact as possible 2, compress as fast as possible

Description

The COMPRESS BLOB command compresses the BLOB blob using the internal 4th Dimension compression algorithm.

The optional compression parameter allows to set the way the BLOB will be compressed:
• If you pass 1, the BLOB is compressed as much as possible, at the expanse of the speed of compression and decompression operations.
• If you pass 2, the BLOB is compressed as fast as possible (and will be decompressed as fast as possible), at the expense of the compression ratio (the compressed BLOB will be bigger).
• If you pass another value or if you omit the parameter, the BLOB is compressed as much as possible, using the compression mode 1.

4th Dimension provides the following predefined constants:

Constant	Type	Value
Compact compression mode	Long Integer	1
Fast compression mode	Long Integer	2

After the call, the OK variable is set to 1 if the BLOB has been successfully compressed. If the compression could not be performed, the OK variable is set to 0; for example, if there is not enough memory to compress a BLOB.

After a BLOB has been compressed, you can expand it using the EXPAND BLOB command.

To detect if a BLOB has been compressed, use the BLOB PROPERTIES command.

WARNING: A compressed BLOB is still a BLOB, so there is nothing to stop you from modifying its contents. However, if you do so, the EXPAND BLOB command will not be able to decompress the BLOB properly.

Examples

1. This example tests if the BLOB vxMyBlob is compressed, and, if it is not, compresses it:

```
     BLOB PROPERTIES (vxMyBlob;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
     If ($vlCompressed=Is not compressed)
⇒        COMPRESS BLOB (vxMyBlob)
     End if
```

Note however, that if you apply COMPRESS BLOB to an already compressed BLOB, the command detects it and does nothing.

2. The following example allows you to select a document and then compress it:

```
     $vhDocRef := Open document ("")
     If (OK=1)
         CLOSE DOCUMENT ($vhDocRef)
         DOCUMENT TO BLOB (Document;vxBlob)
         If (OK=1)
⇒            COMPRESS BLOB (vxBlob)
             If (OK=1)
                 BLOB TO DOCUMENT (vxBlob;Document)
             End if
         End if
     End if
```

See Also

BLOB PROPERTIES, EXPAND BLOB.

System Variables or Sets

The OK variable is set to 1 if the BLOB has been successfully compressed, otherwise it is set to 0.

EXPAND BLOB

version 6.0

EXPAND BLOB (blob)

Parameter	Type		Description
blob	BLOB	→	BLOB to expand

Description

The EXPAND BLOB command expands the BLOB blob that has been previously compressed using the COMPRESS BLOB command.

After the call, the OK variable is set to 1 if the BLOB has been compressed (or if the BLOB was not compressed originally). If the expansion could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

To detect if a BLOB has been compressed, use the BLOB PROPERTIES command.

Examples

1. This example tests if the BLOB vxMyBlob is compressed and, if so, expands it:

```
    BLOB PROPERTIES (vxMyBlob;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
    If ($vlCompressed#Is not compressed)
⇒       EXPAND BLOB (vxMyBlob)
    End if
```

Note however, that if you apply EXPAND BLOB to a BLOB that is not compressed, the command detects it and does nothing.

2. This example allows you to select a document and then expand it, if it is compressed:

```
    $vhDocRef := Open document ("")
    If (OK=1)
        CLOSE DOCUMENT ($vhDocRef)
        DOCUMENT TO BLOB (Document;vxBlob)
        If (OK=1)
            BLOB PROPERTIES (vxBlob;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
            If ($vlCompressed#Is not compressed)
⇒               EXPAND BLOB (vxBlob)
                If (OK=1)
                    BLOB TO DOCUMENT (vxBlob;Document)
                End if
            End if
        End if
    End if
```

See Also

BLOB PROPERTIES, COMPRESS BLOB.

System Variables or Sets

The OK variable is set to 1 if the BLOB has been successfully expanded, otherwise it is set to 0.

BLOB PROPERTIES

BLOB PROPERTIES (blob; compressed{; expandedSize{; currentSize}})

Parameter	Type		Description
blob	BLOB	→	BLOB for which to get information
compressed	Number	←	0 = BLOB is not compressed 1 = BLOB compressed compact 2 = BLOB compressed fast
expandedSize	Number	←	Size of BLOB (in bytes) when not compressed
currentSize	Number	←	Current size of BLOB (in bytes)

Description

The BLOB PROPERTIES command returns information about the BLOB blob.

• The compressed parameter tells whether or not the BLOB is compressed, and returns one of the following values. Note: 4th Dimension provides the predefined constants.

Constant	Type	Value
Is not compressed	Long Integer	0
Compact compression mode	Long Integer	1
Fast compression mode	Long Integer	2

• Whatever the compression status of the BLOB, the expandedSize parameter returns the size of the BLOB when it is not compressed.

• The parameter currentSize returns the current size of the BLOB. If the BLOB is compressed, you will usually obtain currentSize less than expandedSize. If the BLOB is not compressed, you will always obtain currentSize equal to expandedSize.

Examples

1. See examples for the commands COMPRESS BLOB and EXPAND BLOB.

2. After a BLOB has been compressed, the following project method obtains the percentage of space saved by the compression:

```
          ` Space saved by compression project method
          ` Space saved by compression (Pointer {; Pointer } ) -> Long
          ` Space saved by compression ( -> BLOB {; -> savedBytes } ) -> Percentage

      C_POINTER ($1;$2)
      C_LONGINT ($0;$vlCompressed;$vlExpandedSize;$vlCurrentSize)

⇒     BLOB PROPERTIES ($1->;$vlCompressed;$vlExpandedSize;$vlCurrentSize)
      If ($vlExpandedSize=0)
          $0:=0
          If (Count parameters>=2)
              $2->:=0
          End if
      Else
          $0:=100-(($vlCurrentSize/$vlExpandedSize)*100)
          If (Count parameters>=2)
              $2->:=$vlExpandedSize-$vlCurrentSize
          End if
      End if
```

After this method has been added to your application, you can use it this way:

```
          ` ...
      COMPRESS BLOB (vxBlob)
      $vlPercent:=Space saved by compression  (->vxBlob;->vlBlobSize)
      ALERT ("The compression saved "+String (vlBlobSize)+" bytes, so "+String
($vlPercent;"#0%")+
              " of space.")
```

See Also

COMPRESS BLOB, EXPAND BLOB.

DOCUMENT TO BLOB

version 6.0

DOCUMENT TO BLOB (document; blob{; *})

Parameter	Type		Description
document	String	→	Name of the document
blob	BLOB	→	BLOB field or variable to receive the document
		←	Document contents
*	*	→	On Macintosh only: Resource fork is loaded if * is passed otherwise Data fork is loaded

Description

DOCUMENT TO BLOB loads the whole contents of document into blob. You must pass the name of an existing document that is not already open, otherwise an error will be generated. To let the user choose the document to be loaded into the BLOB, use the command Open document and the process variable document (see Example).

Note regarding Macintosh: Macintosh documents can be composed of two forks: the Data fork and the Resource fork. By default, the command DOCUMENT TO BLOB loads the Data fork of the document. To load the Resource fork of the document instead, pass the optional * parameter. On Windows, the optional * parameter is ignored. Note that the 4D environment provides the equivalent of MacOS resource forks on Windows. For example, the data fork of a 4D database is stored in a file with the file extension .4DB; the resource fork is stored in a file with the same name and the file extension .RSR. On Windows, if you write a 4D application with the data fork and resource fork stored in BLOBs, you just need to access the file corresponding to the fork with which you want to work.

Example

You write an Information System that enables you to quickly store and retrieve documents. In a data entry form, you create a button that allows you to load a document into a BLOB field. The method for this button could be:

```
     $vhDocRef:=Open document("")  ` Select the document of your choice
     If (OK=1)  ` If a document has been chosen
        CLOSE DOCUMENT($vhDocRef)  ` We don't need to keep it open
⇒       DOCUMENT TO BLOB (Document;[YourTable]YourBLOBField)
        If (OK=0)
           ` Handle error
        End if
     End if
```

See Also

BLOB TO DOCUMENT, Open document.

System Variables

OK is set to 1 if the document is correctly loaded, otherwise OK is set to 0 and an error is generated.

Error Handling

• If you try to load (into a BLOB) a document that does not exist or that is already open by another process or application, the appropriate File Manager error is generated.

• An I/O error can occur if the document is locked, located on a locked volume, or if there is problem in reading the document.

• If there is not enough memory to load the document, an error -108 is generated.

In each case, you can trap the error using an ON ERR CALL interruption method.

BLOB TO DOCUMENT

version 6.0

BLOB TO DOCUMENT (document; blob{; *})

Parameter	Type		Description
document	String	→	Name of the document
blob	BLOB	→	New contents for the document
*	*	→	On Macintosh only: Resource fork is written if * is passed otherwise Data fork is written

Description

BLOB TO DOCUMENT rewrites the whole contents of document using the data stored in blob. You must pass the name of an existing document that is not already open, otherwise an error will be generated. If you want to let the user choose the document, use the commands Open document or Create document and use the process variable document (see example).

Note regarding Macintosh: Macintosh documents can be composed of two forks: the Data fork and the Resource fork. By default, the command BLOB TO DOCUMENT rewrites the Data fork of the document. To rewrite the Resource fork of the document instead, pass the optional * parameter. On Windows, the optional * parameter is ignored. Note that the 4D environment provides the equivalent of MacOS resource forks on Windows. For example, the data fork of a 4D database is stored in a file with the file extension .4DB; the resource fork is stored in a file with the same name and the file extension .RSR. On Windows, if you write a 4D application with the data fork and resource fork stored in BLOBs, you just need to access the file corresponding to the fork with which you want to work.

Example

You write an Information System that enables you to quickly store and retrieve documents. In a data entry form, you create a button which allows you to save a document that will contain the data previously loaded into a BLOB field. The method for this button could be:

```
      $vhDocRef:=Create document("") ` Save the document of your choice
      If (OK=1) ` If a document has been created
         CLOSE DOCUMENT($vhDocRef) ` We don't need to keep it open
⇒        BLOB TO DOCUMENT (Document;[YourTable]YourBLOBField)
         If (OK=0)
            ` Handle error
         End if
      End if
```

See Also

Create document, DOCUMENT TO BLOB, Open document.

System Variables

OK is set to 1 if the document is correctly written, otherwise OK is set to 0 and an error is generated.

Error Handling

• If you try to rewrite a document that does not exist or that is already open by another process or application, the appropriate File Manager error is generated.

• The disk space may be insufficient for writing the new contents of the document.

• I/O errors can occur while writing the document.

In all cases, you can trap the error using an ON ERR CALL interruption method.

VARIABLE TO BLOB

VARIABLE TO BLOB (variable; blob{; offset | *})

Parameter	Type		Description
variable	Variable	→	Variable to store in the BLOB
blob	BLOB	→	BLOB to receive the variable
offset | *	Variable | *	→	Offset within the BLOB (expressed in bytes) or * to append the value
		←	New offset after writing if not *

Description

The command VARIABLE TO BLOB stores the variable variable in the BLOB blob.

If you specify the * optional parameter, the variable is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of variables or lists (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the offset variable parameter, the variable is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the offset variable parameter, the variable is written at the offset (starting from zero) within the BLOB. No matter where you write the variable, the size of the BLOB is increased according to the location you passed (plus the size of the variable, if necessary). Newly allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the offset variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another variable or list.

VARIABLE TO BLOB accepts any type of variable (including other BLOBs), except the following:
• Pointer
• Array of pointers
• Two-dimensional arrays

However, if you store a Long Integer variable that is a reference to a hierarchical list (ListRef), VARIABLE TO BLOB will store the Long Integer variable, not the list. To store and retrieve hierarchical lists in and from a BLOB, use the commands LIST TO BLOB and BLOB to list.

WARNING: If you use a BLOB for storing variables, you must later use the command BLOB TO VARIABLE for reading back the contents of the BLOB, because variables are stored in BLOBs using a 4D internal format.

After the call, if the variable has been successfully stored, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, there was not enough memory.

Note regarding Platform Independence: VARIABLE TO BLOB and BLOB TO VARIABLE use a 4D internal format for handling variables stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms while using these two commands. In other words, a BLOB created on Windows using either of these commands can be reused on Macintosh, and vice-versa.

Examples

1. The two following project methods allow you to quickly store and retrieve arrays into and from documents on disk:

```
          ` SAVE ARRAY project method
          ` SAVE ARRAY ( String ; Pointer )
          ` SAVE ARRAY ( Document ; -> Array )
      C_STRING (255;$1)
      C_POINTER ($2)
      C_BLOB ($vxArrayData)
⇒     VARIABLE TO BLOB ($2->;$vxArrayData)  ` Store the array into the BLOB
      COMPRESS BLOB ($vxArrayData)  ` Compress the BLOB
      BLOB TO DOCUMENT ($1;$vxArrayData)  ` Save the BLOB on disk

          ` LOAD ARRAY project method
          ` LOAD ARRAY ( String ; Pointer )
          ` LOAD ARRAY ( Document ; -> Array )
      C_STRING (255;$1)
      C_POINTER ($2)
      C_BLOB ($vxArrayData)
      DOCUMENT TO BLOB ($1;$vxArrayData)  ` Load the BLOB from the disk
      EXPAND BLOB ($vxArrayData)  ` Expand the BLOB
⇒     BLOB TO VARIABLE ($vxArrayData;$2->)  ` Retrieve the array from the BLOB
```

After these methods have been added to your application, you can write:

```
      ARRAY STRING (...;asAnyArray;...)
          ` ...
      SAVE ARRAY ( $vsDocName;->asAnyArray)
          ` ...
      LOAD ARRAY ( $vsDocName;->asAnyArray)
```

2. The two following project methods allow you to quickly store and retrieve any set of variables into and from a BLOB:

```
         ` STORE VARIABLES INTO BLOB project method
         ` STORE VARIABLES INTO BLOB ( Pointer { ; Pointer ... { ;  Pointer } } )
         ` STORE VARIABLES INTO BLOB ( BLOB { ; Var1 ... { ; Var2 } } )
      C_POINTER (${1})
      C_LONGINT ($vlParam)

      SET BLOB SIZE ($1->;0)
      For ($vlParam;2;Count parameters)
⇒        VARIABLE TO BLOB (${$vlParam}->;$1->;*)
      End for


         ` RETRIEVE VARIABLES FROM BLOB project method
         ` RETRIEVE VARIABLES FROM BLOB ( Pointer { ; Pointer ... { ;  Pointer } } )
         ` RETRIEVE VARIABLES FROM BLOB ( BLOB { ; Var1 ... { ; Var2 } } )
      C_POINTER (${1})
      C_LONGINT ($vlParam;$vlOffset)

      $vlOffset:=0
      For ($vlParam;2;Count parameters)
⇒        BLOB TO VARIABLE ($1->;${$vlParam}->;$vlOffset)
      End for
```

After these methods have been added to your application, you can write:

```
      STORE VARIABLES INTO BLOB ( ->vxBLOB;->vgPicture;->asAnArray;->alAnotherArray)
         ` ...
      RETRIEVE VARIABLES FROM BLOB ( ->vxBLOB;->vgPicture;->asAnArray;->alAnotherArray)
```

See Also

BLOB to list, BLOB TO VARIABLE, LIST TO BLOB.

System Variables or Sets

The OK variable is set to 1 if the variable has been successfully stored, otherwise it is set to 0.

BLOB TO VARIABLE

version 6.0

BLOB TO VARIABLE (blob; variable{; offset})

Parameter	Type		Description
blob	BLOB	→	BLOB containing 4D variables
variable	Variable	←	Variable to write with BLOB contents
offset	Number	→	Position of variable within BLOB
		←	Position of following variable within BLOB

Description

The command BLOB TO VARIABLE rewrites the variable variable with the data stored within the BLOB blob at the byte offset (starting at zero) specified by offset.

The BLOB data must be consistent with the destination variable. Typically, you will use BLOBs that you previously filled out using the command VARIABLE TO BLOB.

If you do not specify the optional offset parameter, the variable data is read starting from the beginning of the BLOB. If you deal with a BLOB in which several variables have been stored, you must pass the offset parameter and, in addition, you must pass a numeric variable. Before the call, set this numeric variable to the appropriate offset. After the call, that same numeric variable returns the offset of the next variable stored within the BLOB.

After the call, if the variable has been successfully rewritten, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

Note regarding Platform Independence: BLOB TO VARIABLE and VARIABLE TO BLOB use a 4D internal format for handling variables stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms while using these two commands. In other words, a BLOB created on Windows using either of these commands can be reused on Macintosh, and vice-versa.

Example

See the examples for the command VARIABLE TO BLOB.

See Also

VARIABLE TO BLOB.

System Variables or Sets

The OK variable is set to 1 if the variable has been successfully rewritten, otherwise it is set to 0.

LIST TO BLOB

BLOB

version 6.0

LIST TO BLOB (list; blob{; offset | *})

Parameter	Type		Description
list	ListRef	→	Hierarchical list to store in the BLOB
blob	BLOB	→	BLOB to receive the Hierarchical list
offset | *	Variable | *	→	Offset within the BLOB (expressed in bytes) or * to append the value
		←	New offset after writing if not *

Description

The command LIST TO BLOB stores the hierarchical list list in the BLOB blob.

If you specify the * optional parameter, the hierarchical list is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of variables or lists (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the offset variable parameter, the hierarchical list is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the offset variable parameter, the hierarchical list is written at the offset (starting from zero) within the BLOB. No matter where you write the hierarchical list, the size of the BLOB is increased according to the location you passed (plus the size of the list, if necessary). Newly allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the offset variable parameter is returned, incremented by the number of bytes that have been written. You can therefore reuse that same variable with another BLOB writing command to write another variable or list.

WARNING: If you use a BLOB for storing lists, you must later use the command BLOB to list for reading back the contents of the BLOB, because lists are stored in BLOBs using a 4D internal format.

After the call, if the list has been successfully stored, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

Note regarding Platform Independence: LIST TO BLOB and BLOB to list use a 4D internal format for handling lists stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms when using these two commands. In other words, a BLOB created on Windows using those commands can be reused on Macintosh, and vice-versa.

Examples

See example for the command BLOB to list.

See Also

BLOB to list, BLOB TO VARIABLE, VARIABLE TO BLOB.

BLOB to list

BLOB to list (blob{; offset}) → ListRef

Parameter	Type		Description
blob	BLOB	→	BLOB containing a hierarchical list
offset	Number	→	Offset within the BLOB (expressed in bytes)
		←	New offset after reading
Function result	ListRef	←	Reference to newly created list

Description

The command BLOB to list creates a new hierarchical list with the data stored within the BLOB blob at the byte offset (starting at zero) specified by offset and returns a List Reference number for that new list.

The BLOB data must be consistent with the command. Typically, you will use BLOBs that you previously filled out using the command LIST TO BLOB.

If you do not specify the optional offset parameter, the list data is read starting from the beginning of the BLOB. If you deal with a BLOB in which several variables or lists have been stored, you must pass the offset parameter and, in addition, you must pass a numeric variable. Before the call, set this numeric variable to the appropriate offset. After the call, that same numeric variable returns the offset of the next variable stored within the BLOB.

After the call, if the hierarchical list has been successfully created, the OK variable is set to 1. If the operation could not be performed, the OK variable is set to 0; for example, if there was not enough memory.

Note regarding Platform Independence: BLOB to list and LIST TO BLOB use a 4D internal format for handling lists stored in BLOBs. As a benefit, you do not need to worry about byte swapping between platforms when using these two commands. In other words, a BLOB created on Windows using those two commands can be reused on Macintosh and vice-versa.

Example

In this example, the form method for a data entry form extracts a list from a BLOB field before the form appears on the screen, and stores it back to the BLOB field if the data entry is validated:

```
          ` [Things To Do];"Input" Form Method

      Case of

          : (Form event=On Load)
⇒            hList:=BLOB to list([Things To Do]Other Crazy Ideas)
             If (OK=0)
                hList:=New list
             End if

          : (Form event=On Unload)
             CLEAR LIST(hList;*)

          : (bValidate=1)
⇒            LIST TO BLOB(hList;[Things To Do]Other Crazy Ideas)

      End case
```

See Also

LIST TO BLOB.

System Variables and Sets

The OK variable is set to 1 if the list has been successfully created, otherwise it is set to 0.

INTEGER TO BLOB

version 6.0

INTEGER TO BLOB (integer; blob; byteOrder{; offset | *})

Parameter	Type		Description
integer	Number	→	Integer value to write into the BLOB
blob	BLOB	→	BLOB to receive the Integer value
byteOrder	Number	→	0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering
offset | *	Variable | *	←	New offset after writing if not *

Description

The command INTEGER TO BLOB writes the 2-byte Integer value integer into the BLOB blob.

The byteOrder parameter fixes the byte ordering of the 2-byte Integer value to be written. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	2

Note regarding Platform Independence: If you exchange BLOBs between the Macintosh and PC platforms, it is up to you to manage byte swapping issues when using this command.

If you specify the * optional parameter, the 2-byte Integer value is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the offset variable parameter, the 2-byte Integer value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the offset variable parameter, the 2-byte Integer value is written at the byte offset (starting from zero) within the BLOB. No matter where you write the 2-byte Integer value, the size of the BLOB is increased according to the location you passed (plus up to 2 bytes, if necessary). Newly allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the offset variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another value.

Examples

1. After executing this code:

```
⇒       INTEGER TO BLOB (0x0206;vxBlob;Native byte ordering)
```

• The size of vxBlob is 2 bytes
• On Macintosh vxBLOB{0} = $02 and vxBLOB{1} = $06
• On PC vxBLOB{0} = $06 and vxBLOB{1} = $02

2. After executing this code:

```
⇒       INTEGER TO BLOB (0x0206;vxBlob;Macintosh byte ordering)
```

• The size of vxBlob is 2 bytes
• On all platforms vxBLOB{0} = $02 and vxBLOB{1} = $06

3. After executing this code:

```
⇒       INTEGER TO BLOB (0x0206;vxBlob;PC byte ordering)
```

• The size of vxBlob is 2 bytes
• On all platforms vxBLOB{0} = $06 and vxBLOB{1} = $02

4. After executing this code:

```
        SET BLOB SIZE (vxBlob;100)
⇒       INTEGER TO BLOB (0x0206;vxBlob;PC byte ordering;*)
```

• The size of vxBlob is 102 bytes
• On all platforms vxBLOB{100} = $06 and vxBLOB{101} = $02
• The other bytes of the BLOB are left unchanged

5. After executing this code:

```
        SET BLOB SIZE (vxBlob;100)
        vlOffset:=50
⇒       INTEGER TO BLOB (518;vxBlob;Macintosh byte ordering;vlOffset)
```

• The size of vxBlob is 100 bytes
• On all platforms vxBLOB{50} = $02 and vxBLOB{51} = $06
• The other bytes of the BLOB are left unchanged
• The variable vlOffset has been incremented by 2 (and is now equal to 52)

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

LONGINT TO BLOB

LONGINT TO BLOB (longInt; blob; byteOrder{; offset | *})

Parameter	Type		Description
longInt	Number	→	Long Integer value to write into the BLOB
blob	BLOB	→	BLOB to receive the Long Integer value
byteOrder	Number	→	0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering
offset | *	Variable | *	→	Offset within the BLOB (expressed in bytes) or * to append the value
		←	New offset after writing if not *

Description

The command LONGINT TO BLOB writes the 4-byte Long Integer value integer into the BLOB blob.

The byteOrder parameter fixes the byte ordering of the 4-byte Long Integer value to be written. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	2

Note regarding Platform Independence: If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage byte swapping issues while using this command.

If you specify the * optional parameter, the 4-byte Long Integer value is appended to the BLOB and the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter nor the offset variable parameter, the 4-byte Long Integer value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the offset variable parameter, the 4-byte Long Integer value is written at the offset (starting from zero) within the BLOB. No matter where you write the 4-byte Long Integer value, the size of the BLOB is increased according to the location you passed (plus up to 4 bytes, if necessary). New allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the offset variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another value.

Examples

1. After executing this code:

```
⇒	LONGINT TO BLOB (0x01020304;vxBlob;Native byte ordering)
```

• The size of vxBlob is 4 bytes
• On Macintosh vxBLOB{0}=$01, vxBLOB{1}=$02, vxBLOB{2}=$03, vxBLOB{3}=$04
• On PC vxBLOB{0}=$04, vxBLOB{1}=$03, vxBLOB{2}=$02, vxBLOB{3}=$01

2. After executing this code:

```
⇒	LONGINT TO BLOB (0x01020304;vxBlob;Macintosh byte ordering)
```

• The size of vxBlob is 4 bytes
• On all platforms vxBLOB{0}=$01, vxBLOB{1}=$02, vxBLOB{2}=$03, vxBLOB{3}=$04

3. After executing this code:

```
⇒	LONGINT TO BLOB (0x01020304;vxBlob;PC byte ordering)
```

• The size of vxBlob is 4 bytes
• On all platforms vxBLOB{0}=$04, vxBLOB{1}=$03, vxBLOB{2}=$02, vxBLOB{3}=$01

4. After executing this code:

```
	SET BLOB SIZE (vxBlob;100)
⇒	LONGINT TO BLOB (0x01020304;vxBlob;PC byte ordering;*)
```

• The size of vxBlob is 104 bytes
• On all platforms vxBLOB{100}=$04, vxBLOB{101}=$03, vxBLOB{102}=$02, vxBLOB{103}=$01
• The other bytes of the BLOB are left unchanged

5. After executing this code:

```
	SET BLOB SIZE (vxBlob;100)
	vlOffset:=50
⇒	LONGINT TO BLOB (0x01020304;vxBlob;Macintosh byte ordering;vlOffset)
```

• The size of vxBlob is 100 bytes
• On all platforms vxBLOB{50}=$01, vxBLOB{51}=$02, vxBLOB{52}=$03, vxBLOB{53}=$04
• The other bytes of the BLOB are left unchanged
• The variable vlOffset has been incremented by 4 (and is now equal to 54)

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, REAL TO BLOB, TEXT TO BLOB.

REAL TO BLOB

version 6.0

REAL TO BLOB (real; blob; realFormat{; offset | *})

Parameter	Type		Description
real	Number	→	Real value to write into the BLOB
blob	BLOB	→	BLOB to receive the Real value
realFormat	Number	→	0 Native real format 1 Extended real format 2 Macintosh Double real format 3 Windows Double real format
offset | *	Variable | *	→	Offset within the BLOB (expressed in bytes) or * to append the value
		←	New offset after writing if not *

Description

The command REAL TO BLOB writes the Real value real into the BLOB blob.

The realFormat parameter fixes the internal format and byte ordering of the Real value to be written. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
Native real format	Long Integer	0
Extended real format	Long Integer	1
Macintosh double real format	Long Integer	2
PC double real format	Long Integer	3

Platform Independence Note: If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage real formats and byte swapping issues when using this command.

If you specify the * optional parameter, the Real value is appended to the BLOB; the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter or the offset variable parameter, the Real value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the offset variable parameter, the Real value is written at the offset (starting from zero) within the BLOB. No matter where you write the Real value, the size of the BLOB is increased according to the location you passed (plus up to 8 or 10 bytes, if necessary). New allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the offset variable parameter is returned, incremented by the number of bytes that have been written. Therefore, you can reuse that same variable with another BLOB writing command to write another value.

Examples

1. After executing this code:

```
        C_REAL (vrValue)
        vrValue := ...
⇒       REAL TO BLOB (vrValue;vxBlob;Native real format)
```

• On PC and Power Macintosh, the size of vxBlob is 8 bytes
• On Macintosh 68K, the size of vxBlob is 10 bytes

2. After executing this code:

```
        C_REAL (vrValue)
        vrValue := ...
⇒       REAL TO BLOB (vrValue;vxBlob;Extended real format)
```

• On all platforms, the size of vxBlob is 10 bytes

3. After executing this code:

```
        C_REAL (vrValue)
        vrValue := ...
⇒       REAL TO BLOB (vrValue;vxBlob;Macintosh Double real format) ` or Windows double
real format
```

• On all platforms, the size of vxBlob is 8 bytes

4. After executing this code:

```
        SET BLOB SIZE (vxBlob;100)
        C_REAL (vrValue)
        vrValue := ...
⇒       INTEGER TO BLOB (vrValue;vxBlob;Windows Double real format) ` or Macintosh
double real format
```

• On all platforms, the size of vxBlob is 8 bytes

5. After executing this code:

```
        SET BLOB SIZE (vxBlob;100)
⇒      REAL TO BLOB (vrValue;vxBlob;Extended real format;*)
```

• On all platforms, the size of vxBlob is 110 bytes
• On all platforms, the real value is stored at the bytes #100 to #109
• The other bytes of the BLOB are left unchanged

6. After executing this code:

```
        SET BLOB SIZE (vxBlob;100)
        C_REAL (vrValue)
        vrValue := ...
        vlOffset:=50
⇒      REAL TO BLOB (vrValue;vxBlob;Windows Double real format;vlOffset) ` or Macintosh
double real format
```

• On all platforms, the size of vxBlob is 100 bytes
• On all platforms, the real value is stored in the bytes #50 to #57
• The other bytes of the BLOB are left unchanged
• The variable vlOffset has been incremented by 8 (and is now equal to 58)

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, TEXT TO BLOB.

TEXT TO BLOB

version 6.0

TEXT TO BLOB (text; blob; textFormat{; offset | *})

Parameter	Type		Description
text	String	→	Text value to write into the BLOB
blob	BLOB	→	BLOB to receive the text value
textFormat	Number	→	0 C String 1 Pascal String 2 Text with length 3 Text without length
offset | *	Variable | *	→	Offset within the BLOB (expressed in bytes) or * to append the value
		←	New offset after writing if not *

Description

The command TEXT TO BLOB writes the Text value text into the BLOB blob.

The textFormat parameter fixes the internal format of the text value to be written. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
C string	Long Integer	0
Pascal string	Long Integer	1
Text with length	Long Integer	2
Text without length	Long Integer	3

The following table describes each of these formats:

Text format	Description and Examples
C string	The text is ended by a NULL character (ASCII code $00) "" → $00 "Hello World!" → $48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00
Pascal string	The text is preceded by a 1-byte length "" → $00 "Hello World!" → $0C 48 65 6C 6C 6F 20 57 6F 72 6C 64 21
Text with length	The text is preceded by a 2-byte length "" → $00 00 "Hello World!" → $00 0C 48 65 6C 6C 6F 20 57 6F 72 6C 64 21

Text without length	The text is composed only of its characters. "" → No data "Hello World!" → $48 65 6C 6C 6F 20 57 6F 72 6C 64 21

Note: The command accepts both Text (declared with C_TEXT) and String (declared with C_STRING) expressions. Remember that a Text variable can contain up to 32,000 characters and a String variable can contain up to the number of characters in its declaration, with a maximum of 255 characters.

If you specify the * optional parameter, the Text value is appended to the BLOB; the size of the BLOB is extended accordingly. Using the * optional parameter, you can sequentially store any number of Integer, Long Integer, Real or Text values (see other BLOB commands) in a BLOB, as long as the BLOB fits into memory.

If you do not specify the * optional parameter nor the offset variable parameter, the Text value is stored at the beginning of the BLOB, overriding its previous contents; the size of the BLOB is adjusted accordingly.

If you pass the offset variable parameter, the Text value is written at the offset (starting from zero) within the BLOB. No matter where you write the Text value, the size of the BLOB is, increased according to the location you passed (plus up to the size of the text, if necessary). New allocated bytes, other than the ones you are writing, are initialized to zero.

After the call, the offset variable parameter is returned, incremented by the number of bytes that have been written. Therfore, you can reuse that same variable with another BLOB writing command to write another value.

Example

After executing this code:

```
	SET BLOB SIZE (vxBlob;0)
	C_TEXT (vtValue)
	vtValue := "Hello World!"  ` Length of vtValue is 12 bytes
⇒	TEXT TO BLOB (vtValue;vxBlob;C string)  ` Size of BLOB becomes 13 bytes
⇒	TEXT TO BLOB (vtValue;vxBlob;Pascal string)  ` Size of BLOB becomes 13 bytes
⇒	TEXT TO BLOB (vtValue;vxBlob;Text with length)  ` Size of BLOB becomes 14 bytes
⇒	TEXT TO BLOB (vtValue;vxBlob;Text without length)  ` Size of BLOB becomes 12 bytes
```

See Also

BLOB to integer, BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB.

BLOB to integer

BLOB to integer (blob; byteOrder{; offset}) → Number

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the integer value
byteOrder	Number	→	0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering
offset	Variable	→	Offset within the BLOB (expressed in bytes)
		←	New offset after reading
Function result	Number	←	2-byte Integer value

Description

The command BLOB to integer returns a 2-byte Integer value read from the BLOB blob.

The byteOrder parameter fixes the byte ordering of the 2-byte Integer value to be read. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	2

Note regarding Platform Independence: If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage byte swapping issues when using this command.

If you specify the optional offset variable parameter, the 2-byte Integer value is read at the offset (starting from zero) within the BLOB. If you do not specify the optional offset variable parameter, the first two bytes of the BLOB are read.

Note: You should pass an offset (in bytes) value between 0 (zero) and the size of the BLOB minus 2. If you do not do so, an error -111 is generated.

After the call, the variable is incremented by the number of bytes read, Therefore, you can reuse that same variable with another BLOB reading command to read another value.

Example

The following example reads 20 Integer values from a BLOB, starting at the offset 0x200:

```
        $vlOffset:=0x200
        For ($viLoop;0;19)
⇒           $viValue:=BLOB to integer(vxSomeBlob;PC byte ordering;$vlOffset)
                ` Do something with $viValue
        End for
```

See Also

BLOB to longint, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB to longint

version 6.0

BLOB to longint (blob; byteOrder{; offset}) → Number

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the Long Integer value
byteOrder	Number	→	0 Native byte ordering 1 Macintosh byte ordering 2 PC byte ordering
offset	Variable	→	Offset within the BLOB (expressed in bytes)
		←	New offset after reading
Function result	Number	←	4-byte Long Integer value

Description

The command BLOB to longint returns a 4-byte Long Integer value read from the BLOB blob.

The byteOrder parameter fixes the byte ordering of the 4-byte Long Integer value to be read. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
Native byte ordering	Long Integer	0
Macintosh byte ordering	Long Integer	1
PC byte ordering	Long Integer	2

Note regarding Platform Independence: If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage byte swapping issues while using this command.

If you specify the optional offset variable parameter, the 4-byte Long Integer is read at the offset (starting from zero) within the BLOB. If you do not specify the optional offset variable parameter, the first four bytes of the BLOB are read.

Note: You should pass an offset value between 0 (zero) and the size of the BLOB minus 4. If you do not do so, an error -111 is generated.

After the call, the variable is incremented by the number of bytes read. Therefore, you can reuse that same variable with another BLOB reading command to read another value.

Example

The following example reads 20 Long Integer values from a BLOB, starting at the offset 0x200:

```
        $vlOffset:=0x200
        **For** ($viLoop;0;19)
⇒           $vlValue:=**BLOB to longint**(vxSomeBlob;PC byte ordering;$vlOffset)
                ` Do something with $vlValue
        **End for**
```

See Also

BLOB to integer, BLOB to real, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB to real

version 6.0

BLOB to real (blob; realFormat{; offset})

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the Real value
realFormat	Number	→	0 Native real format 1 Extended real format 2 Macintosh Double real format 3 Windows Double real format
offset	Variable	→	Offset within the BLOB (expressed in bytes)
		←	New offset after reading

Description

The command BLOB to real returns a Real value read from the BLOB blob.

The realFormat parameter fixes the internal format and byte ordering of the Real value to be read. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
Native real format	Long Integer	0
Extended real format	Long Integer	1
Macintosh double real format	Long Integer	2
PC double real format	Long Integer	3

Note regarding Platform Independence: If you exchange BLOBs between Macintosh and PC platforms, it is up to you to manage real formats and byte swapping issues while using this command.

If you specify the optional offset variable parameter, the Read value is read at the offset (starting from zero) within the BLOB. If you do not specify the optional offset variable parameter, the first 8 or 10 bytes of the BLOB are read.

Note: You should pass an offset value between 0 (zero) and the size of the BLOB minus 8 or 10. If you do not do so, an error -111 is generated.

After the call, the variable is incremented by the number of bytes read. Therefore, you can reuse that same variable with another BLOB reading command to read another value.

Example

The following example reads 20 Real values from a BLOB, starting at the offset 0x200:

```
        $vlOffset:=0x200
        For ($viLoop;0;19)
⇒           $vrValue:=BLOB to real(vxSomeBlob;PC byte ordering;$vlOffset)
                ` Do something with $vrValue
        End for
```

See Also

BLOB to integer, BLOB to longint, BLOB to text, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

BLOB to text

version 6.0

BLOB to text (blob; textFormat{; offset{; textLength}})

Parameter	Type		Description
blob	BLOB	→	BLOB from which to get the Text value
textFormat	Number	→	0 C String 1 Pascal String 2 Text with length 3 Text without length
offset	Variable	→	Offset within the BLOB (expressed in bytes)
		←	New offset after reading
textLength	Number	→	Number of characters to be read

Description

The command BLOB to text returns a Text value read from the BLOB blob. The textFormat parameter fixes the internal format of the text value to be read. You pass one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
C string	Long Integer	0
Pascal string	Long Integer	1
Text with length	Long Integer	2
Text without length	Long Integer	3

The following table describes each of these formats:

Text format	Description & Examples
C string	The text is ended by a NULL character (ASCII code $00) "" → $00 "Hello World!" → $48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00
Pascal string	The text is preceded a 1-byte length "" → $00 "Hello World!" → $0C 48 65 6C 6C 6F 20 57 6F 72 6C 64 21
Text with length	The text is preceded by a 2-byte length "" → $00 00 "Hello World!" → $00 0C 48 65 6C 6C 6F 20 57 6F 72 6C 64 21
Text without length	The text is only composed of its characters. "" → No data "Hello World!" → $48 65 6C 6C 6F 20 57 6F 72 6C 64 21

WARNING: The number of characters to be read is determined by the textFormat parameter, EXCEPT for the format Text without length, for which you MUST specify the number of characters to be read in the parameter textLength. For the other formats, textLength is ignored and you can omit it.

Remember that a Text variable can contain up to 32,000 characters and a String variable can contain up to the number of characters in its declaration, with a maximum of 255 characters. If you try to read more data than a variable can hold, 4D will truncate the result of the command when placing it into the variable.

If you specify the optional offset variable parameter, the Text value is read at the offset (starting from zero) within the BLOB. If you do not specify the optional offset variable parameter, the beginning of the BLOB is read according to the value you pass in textFormat. Note that you must pass the offset variable parameter when you are reading text without length.

Note: You should pass an offset value between 0 (zero) and the size of the BLOB minus the size of the text to be read. If you do not do so, the function result is unpredictable.

After the call, the variable is incremented by the number of bytes read. Therefore, you can reuse that same variable with another BLOB reading command to read another value.

Example

The following example reads an hypothetical MacOS-based resource whose internal format is identical to that of the 'STR#' resources:

```
    GET RESOURCE ("ABCD";viResID;vxResData;viMyResFile)
    vlSize:=BLOB Size(vxResData)
    If (vlSize>0)
          ` The resource starts with a 2-byte integer specifying the number of strings
       vlOffset:=0
       viNbEntries:=BLOB to integer(vxResData;Macintosh Byte Ordering;vlOffset)
          ` Then the resource contains concatenated, not padded, Pascal strings
       For (viEntry;1;viNbEntries)
          If (vlOffset<vlSize)
⇒             vsEntry:=BLOB to text(vxResData;Pascal string;vlOffset)
                ` Do something with vsEntry
          Else
                ` Resource data is invalid, get out of the loop
             viEntry:=viNbEntries+1
          End if
       End for
    End if
```

See Also

BLOB to integer, BLOB to longint, BLOB to real, INTEGER TO BLOB, LONGINT TO BLOB, REAL TO BLOB, TEXT TO BLOB.

INSERT IN BLOB

INSERT IN BLOB (blob; offset; len{; filler})

Parameter	Type		Description
blob	BLOB	→	BLOB into which bytes will be inserted
offset	Variable	→	Starting position where bytes will be inserted
		←	Ending position after insertion
len	Number	→	Number of bytes to be inserted
filler	Number	→	Default byte value (0x00..0xFF) 0x00 if omitted

Description

The command INSERT IN BLOB inserts the number of bytes specified by len into the BLOB blob at the position specified by offset. The BLOB then becomes len bytes larger.

If you do not specify the optional filler parameter, the bytes inserted into the BLOB are set to 0x00. Otherwise, the bytes are set to the value you pass in filler (modulo 256 — 0..255).

Before the call, you pass, in the offset variable parameter, the position of the insertion relative to the beginning of the BLOB. After the call, the offset variable parameter returns the position just after the insertion.

See Also

DELETE FROM BLOB.

DELETE FROM BLOB

version 6.0

DELETE FROM BLOB (blob; offset; len)

Parameter	Type		Description
blob	BLOB	→	BLOB from which to delete bytes
offset	Number	→	Starting offset where bytes will be deleted
len	Number	→	Number of bytes to be deleted

Description

The command DELETE FROM BLOB deletes the number of bytes specified by len from the BLOB blob at the position specified by offset (expressed relative to the beginning of the BLOB). The BLOB then becomes len bytes smaller.

See Also

INSERT IN BLOB.

COPY BLOB

COPY BLOB (srcBLOB; dstBLOB; srcOffset; dstOffset; len)

Parameter	Type		Description
srcBLOB	BLOB	→	Source BLOB
dstBLOB	BLOB	→	Destination BLOB
srcOffset	Variable	→	Source position for the copy
		←	Ending position after the copy
dstOffset	Variable	→	Destination position for the copy
		←	Ending position after the copy
len	Number	→	Number of bytes to be copied

Description

The command COPY BLOB copies the number of bytes specified by len from the BLOB srcBLOB to the BLOB dstBLOB.

The copy starts at the position (expressed relative to the beginning of the source BLOB) specified by srcOffset and takes place at the position (expressed relative to the beginning of the destination BLOB) specified by dstOffset.

Note: The destination BLOB can be resized if necessary.

After the call, the variables srcOffset and dstOffset return, respectively, the positions within the source and destination BLOBs just after the copy.

See Also

DELETE FROM BLOB, INSERT IN BLOB.

6 Boolean

Boolean Commands

Boolean

version 6.0

4D includes Boolean functions, are used for Boolean calculations:

True
False
Not

Examples

This example sets a Boolean variable based on the value of a button. It returns True in myBoolean if the myButton button was clicked and False if the button was not clicked. When a button is clicked, the button variable is set to 1.

```
If (myButton=1)   ` If the button was clicked
    myBoolean:=True   ` myBoolean is set to True
Else   ` If the button was not clicked,
    myBoolean:=False   ` myBoolean is set to False
End if
```

The previous example can be simplified into one line.

```
myBoolean:=(myButton=1)
```

See Also

False, Logical Operators, Not, True.

In addition, the following 4D commands return a Boolean result: Activated, After, Before, Before selection, Before subselection, Caps lock down, Compiled application, Deactivated, During, End selection, End subselection, In break, In footer, In header, In transaction, Is a list, Is a variable, Is in set, Is user deleted, Locked, Macintosh command down, Macintosh control down, Macintosh option down, Modified, Modified record, Nil, Outside call, Read only state, Semaphore, Shift down, True, Undefined, User in group, Windows Alt down, Windows Ctrl down.

True

Boolean

version 3

True → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

True returns the Boolean value True.

Example

The following example sets the variable vbOptions to True:

```
⇒	vbOptions:=True
```

See Also

False, Not.

False

Boolean

version 3

False → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

False returns the Boolean value False.

Example

The following example sets the variable vbOptions to False:

```
⇒	vbOptions:=False
```

See Also

Not, True.

Not

Boolean

version 3

Not (boolean) → Boolean

Parameter	Type		Description
boolean	Boolean	→	Boolean value to negate

Description

The Not function returns the negation of boolean, changing True to False or False to True.

Example

This example first assigns True to a variable, then changes the variable value to False, and then back to True.

```
   vResult:=True  ` vResult is set to True
⇒  vResult:=Not(vResult)  ` vResult is set to False
⇒  vResult:=Not(vResult)  ` vResult is set to True
```

7 Clipboard

APPEND TO CLIPBOARD

version 6.0

APPEND TO CLIPBOARD (dataType; data)

Parameter	Type		Description
dataType	String	→	4-character data type string
data	BLOB	→	Data to append to the Clipboard

Description

The APPEND TO CLIPBOARD command appends to the Clipboard the data contained in the BLOB data under the data type specified in dataType.

WARNING: The value you pass in dataType is case sensitive, i.e., “abcd” is not equal to “ABCD.”

If the BLOB data is correctly appended to the Clipboard, the OK variable is set to 1. Otherwise the OK variable is set to 0 and an error may be generated.

Usually, you will use the APPEND TO CLIPBOARD command to append multiple instances of the same data to the Clipboard or to append data that is not of type TEXT or PICT. To append new data to the Clipboard, you must first clear the Clipboard using the CLEAR CLIPBOARD command.

If you want to clear and append:
• text to the Clipboard, use the SET TEXT TO CLIPBOARD command,
• a picture to the Clipboard, use the SET PICTURE TO CLIPBOARD command.

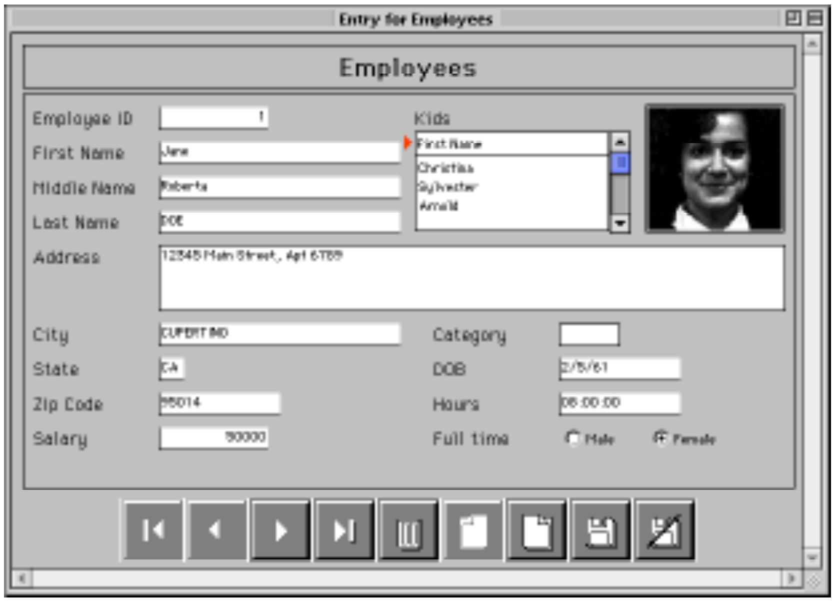
However, note that if a BLOB actually contains some text or a picture, you can use the APPEND TO CLIPBOARD command to append a text or a picture to the Clipboard.

Example

Using Clipboard commands and BLOBs, you can build sophisticated Cut/Copy/Paste schemes that deal with structured data rather than a unique piece of data. In the following example, the two project methods SET RECORD TO CLIPBOARD and GET RECORD FROM CLIPBOARD enable you to treat a whole record as one piece of data to be copied to or from the Clipboard.

```
        ` SET RECORD TO CLIPBOARD project method
        ` SET RECORD TO CLIPBOARD ( Number )
        ` SET RECORD TO CLIPBOARD ( Table number )

    C_LONGINT($1;$vlField;$vlFieldType)
    C_POINTER($vpTable;$vpField)
    C_STRING(255;$vsDocName)
    C_TEXT($vtRecordData;$vtFieldData)
    C_BLOB($vxRecordData)

        ` Clear the Clipboard (it will stay empty if there is no current record)
⇒   CLEAR CLIPBOARD
        ` Get a pointer to the table whose number is passed as parameter
    $vpTable:=Table($1)
        ` If there is a current record for that table
    If ((Record number($vpTable->)>=0) | (Record number($vpTable->)=-3))
            ` Initialize the text variable that will hold the text image of the record
        $vtRecordData:=""
            ` For each field of the record:
        For ($vlField;1;Count fields($1))
                ` Get the type of the field
            GET FIELD PROPERTIES($1;$vlField;$vlFieldType)
                ` Get a pointer to the field
            $vpField:=Field($1;$vlField)
                ` Depending on the type of the field, copy (or not) its data
                ` in the appropriate manner
            Case of
                : (($vlFieldType=Is Alpha field ) | ($vlFieldType=Is Text ))
                        $vtFieldData:=$vpField->
                : (($vlFieldType=Is Real ) | ($vlFieldType=Is Integer ) |
                   ($vlFieldType=Is LongInt ) | ($vlFieldType=Is Date ) | ($vlFieldType=Is Time ))
                    $vtFieldData:=String($vpField->)
                : ($vlFieldType=Is Boolean )
                    $vtFieldData:=String(Num($vpField->);"Yes;;No")
            Else
                    ` Skip and ignore other field data types
                $vtFieldData:=""
            End case
                ` Accumulate the field data into the text variable holding
                ` the text image of the record
            $vtRecordData:=$vtRecordData+Field name($1;$vlField)+":"+Char(9)
                                                                    +$vtFieldData+CR
                ` Note: The method CR returns Char(13) on Macintosh
                ` and Char(13)+Char(10) on Windows
        End for
            ` Put the text image of the record into the clipboard
        SET TEXT TO CLIPBOARD($vtRecordData)
        ` Name for scrap file in Temporary folder
     $vsDocName:=Temporary folder+"Scrap"+String(1+(Random%99))
        ` Delete the scrap file if it exists (error should be tested here)
     DELETE DOCUMENT($vsDocName)
        ` Create scrap file
     SET CHANNEL(10;$vsDocName)
        ` Send the whole record into the scrap file
     SEND RECORD($vpTable->)
        ` Close the scrap file
     SET CHANNEL(11)
        ` Load the scrap file into a BLOB
     DOCUMENT TO BLOB($vsDocName;$vxRecordData)
        ` We longer need the scrap file
     DELETE DOCUMENT($vsDocName)
        ` Append the full image of the record into the Clipboard
        ` Note: We use arbitrarily "4Drc" as data type
⇒    APPEND TO CLIPBOARD("4Drc";$vxRecordData)
        ` At this point, the clipboard contains:
        ` (1) A text image of the record (as shown in the screen shots below)
        ` (2) A whole image of the record (Picture, Subfile and BLOB fields included)
  End if
```

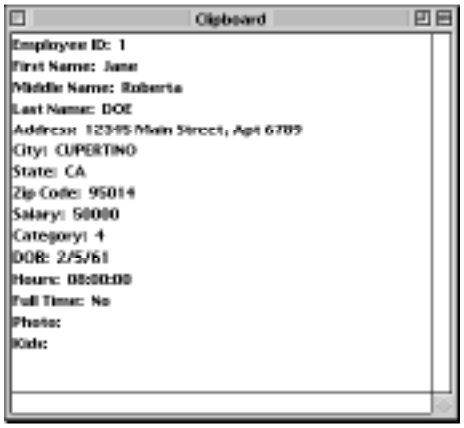
While entering the following record:


Entry for Employees

Employees

Employee ID	1	Kids	First Name: Christina, Sylvester, Arnold
First Name	Jane		
Middle Name	Roberta		
Last Name	DOE		
Address	12345 Main Street, Apt 6789		
City	CUPERTINO	Category	
State	CA	DOB	2/5/61
Zip Code	95014	Hours	08:00:00
Salary	50000	Full time	Male [✓ Female]

If you apply the method SET RECORD TO CLIPBOARD to the [Employees] table, the Clipboard will contain the text image of the record, as shown, and also the whole image of the record.



```
Employee ID: 1
First Name: Jane
Middle Name: Roberta
Last Name: DOE
Address: 12345 Main Street, Apt 6789
City: CUPERTINO
State: CA
Zip Code: 95014
Salary: 50000
Category: 4
DOB: 2/5/61
Hours: 08:00:00
Full Time: No
Photo:
Kids:
```

You can paste this image of the record to another record, using the method GET RECORD FROM CLIPBOARD, as follows:

```
      ` GET RECORD FROM CLIPBOARD method
      ` GET RECORD FROM CLIPBOARD ( Number )
      ` GET RECORD FROM CLIPBOARD ( Table number )
   C_LONGINT($1;$vlField;$vlFieldType;$vlPosCR;$vlPosColon)
   C_POINTER($vpTable;$vpField)
   C_STRING(255;$vsDocName)
   C_BLOB($vxClipboardData)
   C_TEXT($vtClipboardData;$vtFieldData)

      ` Get a pointer to the table whose number is passed as parameter
   $vpTable:=Table($1)
      ` If there is a current record
   If ((Record number($vpTable->)>=0) | (Record number($vpTable->)=-3))
      Case of
            ` Does the clipboard contain a full image record?
         : (Test clipboard("4Drc")>0)
               ` If so, extract the clipboard contents
            GET CLIPBOARD("4Drc";$vxClipboardData)
               ` Name for scrap file in Temporary folder
            $vsDocName:=Temporary folder+"Scrap"+String(1+(Random%99))
               ` Delete the scrap file if it exists (error should be tested here)
            DELETE DOCUMENT($vsDocName)
               ` Save the BLOB into the scrap file
            BLOB TO DOCUMENT($vsDocName;$vxClipboardData)
```

```
		` Open the scrap file
	SET CHANNEL(10;$vsDocName)
		` Receive the whole record from the scrap file
	RECEIVE RECORD($vpTable->)
		` Close the scrap file
	SET CHANNEL(11)
		` We longer need the scrap file
	DELETE DOCUMENT($vsDocName)
		` Does the clipboard contain TEXT?
: (Test clipboard("TEXT")>0)
		` Extract the text from the clipboard
	$vtClipboardData:=Get text from clipboard
		` Initialize field number to be increment
	$vlField:=0
	Repeat
			` Look for the next field line in the text
		$vlPosCR:=Position(CR ;$vtClipboardData)
		If ($vlPosCR>0)
				` Extract the field line
			$vtFieldData:=Substring($vtClipboardData;1;$vlPosCR-1)
				` If there is a colon ":"
			$vlPosColon:=Position(":";$vtFieldData)
			If ($vlPosColon>0)
				` Take only the field data (eliminate field name)
				$vtFieldData:=Substring($vtFieldData;$vlPosColon+2)
			End if
				` Increment field number
			$vlField:=$vlField+1
				` Clipboard may contain more data than we need...
			If ($vlField<=Count fields($vpTable))
					` Get the type of the field
				GET FIELD PROPERTIES($1;$vlField;$vlFieldType)
					` Get a pointer to the field
				$vpField:=Field($1;$vlField)
					` Depending on the type of the field,
					` copy (or not) the text in the appropriate manner
				Case of
					: (($vlFieldType=Is Alpha field ) | ($vlFieldType=Is Text ))
						$vpField->:=$vtFieldData
					: (($vlFieldType=Is Real ) |
								($vlFieldType=Is Integer ) | ($vlFieldType=Is LongInt ))
						$vpField->:=Num($vtFieldData)
					: ($vlFieldType=Is Date )
						$vpField->:=Date($vtFieldData)
					: ($vlFieldType=Is Time )
						$vpField->:=Time($vtFieldData)
```

```
                              : ($vlFieldType=Is Boolean )
                                 $vpField->:=($vtFieldData="Yes")
                           Else
                                 ` Skip and ignore other field data types
                           End case
                        Else
                              ` All fields have been assigned, get out of the loop
                           $vtClipboardData:=""
                        End if
                           ` Eliminate text that has just been extracted
                        $vtClipboardData:=Substring($vtClipboardData;$vlPosCR+Length(CR ))
                     Else
                           ` No delimiter found, get out of the loop
                        $vtClipboardData:=""
                     End if
                        ` Repeat as long as we have data
                  Until (Length($vtClipboardData)=0)
               Else
                  ALERT("The Clipboard does not any data that can be pasted as a record.")
            End case
         End if
```

See Also

CLEAR CLIPBOARD, SET PICTURE TO CLIPBOARD, SET TEXT TO CLIPBOARD.

System Variables

If the BLOB data is correctly appended to the clipboard, OK is set to 1; otherwise OK is set to 0 and an error may be generated.

Error Handling

If there is not enough memory to append the BLOB data to the clipboard, an error -108 is generated.

CLEAR CLIPBOARD

CLEAR CLIPBOARD

Parameter	Type	Description
This command does not require any parameters		

Description

The CLEAR CLIPBOARD command clears the Clipboard of its contents. If the Clipboard contains multiple instances of the same data, all instances are cleared. After a call to CLEAR CLIPBOARD, the Clipboard becomes empty.

You must call CLEAR CLIPBOARD once before appending new data to the Clipboard using the command APPEND TO CLIPBOARD, because this latter command does not clear the Clipboard before appending the new data.

Calling CLEAR CLIPBOARD once and then calling APPEND TO CLIPBOARD several times enables you to Cut or Copy the same data under different formats.

On the other hand, the commands SET TEXT TO CLIPBOARD and SET PICTURE TO CLIPBOARD automatically clear the Clipboard before appending the TEXT or PICT data to it.

Example

(1) The following code clears and then appends data to the clipboard:

```
⇒	CLEAR CLIPBOARD  ` Make sure the clipboard becomes empty
	APPEND TO CLIPBOARD('XWKZ';$vxSomeData)  ` Append some data of type 'XWKZ'
	APPEND TO CLIPBOARD('SYLK';$vxSylkData)  ` Append same data but as Sylk data
```

(2) See example for the APPEND TO CLIPBOARD command.

See Also

APPEND TO CLIPBOARD.

GET CLIPBOARD

version 6.0

GET CLIPBOARD (dataType; data)

Parameter	Type		Description
dataType	String	→	4-character string data type
data	BLOB	←	Requested data extracted from the clipboard

Description

The GET CLIPBOARD command returns into the BLOB field or into the variable data the data present in the Clipboard and whose type you pass in dataType.

WARNING: The value you pass in dataType is case sensitive, i.e., “abcd” is not equal to “ABCD.”

If the data is correctly extracted from the clipboard, the command sets the OK variable to 1. If the Clipboard is empty or does not contains any data of the specified type, the command returns an empty BLOB, sets the OK variable to 0 and generates an error -102. If there is not enough memory to extract the data from the clipboard,the command sets the OK variable to 0 and generates an error -108.

Example

The following object methods for two buttons copy from and paste data to the array asOptions (pop-up menu, drop-downlist,...) located in a form:

```
      ` bCopyasOptions object method
   If (Size of array(asOptions)>0)  ` Is there something to copy?
         ` Accumulate the array elements in a BLOB
      VARIABLE TO BLOB (asOptions;$vxClipData)
      CLEAR CLIPBOARD  ` Empty the clipboard
      APPEND TO CLIPBOARD ("artx";asOptions)  ` Note the data type arbitrarily chosen
   End if

      ` bPasteasOptions object method
   If (Test clipboard ("artx")>0)  ` Is there some "artx" data in the clipboard?
⇒     GET CLIPBOARD ("artx";$vxClipData)  ` Extract the data from the clipboard
         ` Populate the array with the BLOB data
      BLOB TO VARIABLE ($vxClipData;asOptions)
      asOptions:=0  ` Reset the selected element for the array
   End if
```

See Also
APPEND TO CLIPBOARD, GET PICTURE FROM CLIPBOARD, Get text from clipboard.

System Variables

If the data is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

Error Handling

- If there is not enough memory to extract the data, an error -108 is generated.
- If there is no data of the requested type in the clipboard, an error -102 is generated.

GET PICTURE FROM CLIPBOARD

version 6.0

GET PICTURE FROM CLIPBOARD (picture)

Parameter	Type		Description
picture	Picture	←	Picture extracted from the Clipboard

Description

GET PICTURE FROM CLIPBOARD returns the picture present in the Clipboard into the picture field or variable picture.

If the picture is correctly extracted from the Clipboard, the command sets the OK variable to 1. If the Clipboard is empty or does not contain a picture, the command returns an empty picture, sets the OK variable to 0, and generates an error -102. If there is not enough memory to extract the picture from the Clipboard, the command sets the OK variable to 0 and generates an error -108.

Examples

The following button's object method assigns the picture present in the Clipboard (if any) to the field [Employees]Photo:

```
    If (Test clipboard ("PICT")>0)
⇒       GET PICTURE FROM CLIPBOARD ([Employees]Photo)
    Else
        ALERT ("The clipboard does not contain any picture.")
    End if
```

See Also

GET CLIPBOARD, Get text from clipboard, Test clipboard.

System Variables

If the picture is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

Error Handling

- If there is not enough memory to extract the picture, an error -108 is generated.
- If there is no picture in the Clipboard, an error -102 is generated.

Get text from clipboard

version 6.0

Get text from clipboard → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Returns the text (if any) present in the Clipboard

Description

Get text from clipboard returns the text present in the clipboard.

If the text is correctly extracted from the Clipboard, the command sets the OK variable to 1. If the Clipboard is empty or does not contain any text, the command returns an empty string, sets the OK variable to 0, and generates an error -102. If there is not enough memory to extract the text from the Clipboard, the command sets the OK variable to 0 and generates an error -108.

4th Dimension text fields and variables can contain up to 32,000 characters. If there are more than 32,000 characters in the Clipboard, the result returned by Get text from clipboard will be truncated when placed into the field or variable receiving the value. To handle very large Clipboard text contents, first test the size of the data using the command Test clipboard. Then, if the text exceeds 32,000 characters, use the command GET CLIPBOARD instead of Get text from clipboard.

Examples

The following example tests the for the presence of text in the Clipboard, then, depending on the size of the data, extracts the text from the Clipboard as text or as a BLOB:

```
     $vlSize:=Test clipboard ("TEXT")
     Case of
        : ($vlSize<=0)
           ALERT ("There is no text in the clipboard.")
        : ($vlSize<=32000)
⇒          $vtClipData:=Get text from clipboard
           If (OK=1)
              ` Do something with the text
           End if
        : ($vlSize>32000)
            GET CLIPBOARD ("TEXT";$vxClipData)
            If (OK=1)
                ` Do something with the BLOB
            End if
    End case
```

See Also

GET CLIPBOARD, GET PICTURE FROM CLIPBOARD, Test clipboard.

System Variables

If the text is correctly extracted, OK is set to 1; otherwise OK is set to 0 and an error is generated.

Error Handling

• If there is not enough memory to extract the text, an error -108 is generated.
• If there is no text in the Clipboard, an error -102 is generated.

SET PICTURE TO CLIPBOARD

version 6.0

SET PICTURE TO CLIPBOARD (picture)

Parameter	Type		Description
picture	Picture	→	Picture whose copy is to be put into the Clipboard

Description

SET PICTURE TO CLIPBOARD clears the Clipboard and puts a copy of the picture you passed in picture into the Clipboard.

After you have put a picture into the Clipboard, you can retrieve it using the command GET PICTURE FROM CLIPBOARD or by calling GET CLIPBOARD ("PICT";...).

If the picture is correctly put in the Clipboard, the OK variable is set to 1. If there is not enough memory to put a copy of the picture into the Clipboard, the OK variable is set to 0, but no error is generated.

Example

Using a floating window, you display a form that contains the array asEmployeeName, which lists the names of the employees from an [Employees] table. Each time you click on a name, you want to copy the employee's picture to the Clipboard. In the object method for the array, you write:

```
   If (asEmployeeName#0)
      QUERY ([Employees];[Employees]Last name=asEmployeeName{asEmployeeName})
      If (Picture size ([Employees]Photo)>0)
⇒        SET PICTURE TO CLIPBOARD ([Employees]Photo)  ` Copy the employee's photo
      Else
         CLEAR CLIPBOARD  ` No photo or no record found
      End if
   End if
```

See Also

APPEND TO CLIPBOARD, GET PICTURE FROM CLIPBOARD.

System Variables or Sets

If a copy of the picture is correctly put into the Clipboard, the OK variable is set to 1.

SET TEXT TO CLIPBOARD

version 6.0

SET TEXT TO CLIPBOARD (text)

Parameter	Type		Description
text	String	→	Text whose copy is to be put into the Clipboard

Description

SET TEXT TO CLIPBOARD clears the clipboard and then puts a copy of the text you passed in text into the Clipboard.

After you have put some text into the Clipboard, you can retrieve it using the Get text from clipboard command or by calling GET CLIPBOARD ("TEXT";...).

If the text is correctly put in the Clipboard, the OK variable is set to 1. If there is not enough memory to put a copy of the text into the Clipboard, the OK variable is set to 0, but no error is generated.

4th Dimension text expressions can contain up to 32,000 characters. To copy larger text values, accumulate the text into a BLOB, call CLEAR CLIPBOARD, then call APPEND TO CLIPBOARD ("TEXT";...).

Example

See the example for the APPEND TO CLIPBOARD command.

See Also

APPEND TO CLIPBOARD, Get text from clipboard.

System Variables or Sets

If a copy of the text is correctly put into the Clipboard, the OK variable is set to 1.

Test clipboard

Test clipboard (dataType) → Number

Parameter	Type		Description
dataType	String	→	4-character data type string
Function result	Number	←	Size (in bytes) of data stored in Clipboard or error code result

Description

The Test clipboard command allows you to test if there is data of the type you passed in dataType present in the Clipboard.

WARNING: The value you pass in dataType is case sensitive, i.e., “abcd” is not equal to “ABCD.”

If the Clipboard is empty or does not contain any data of the specified type, the command returns an error -102 (see the table of predefined constants). If the Clipboard contains data of the specified type, the command returns the size of this data, expressed in bytes.

After you have detected that the Clipboard contains data of the type in which you are interested, you can extract that data from the Clipboard using one the following commands:

• If the Clipboard contains type TEXT data, you can obtain that data using the Get text from clipboard command, which returns a text value, or the GET CLIPBOARD command, which returns the text into a BLOB.

• If the Clipboard contains type PICT data, you can obtain that data using the GET PICTURE FROM CLIPBOARD command, which returns the picture into a picture field or variable, or the GET CLIPBOARD command, which returns the picture into a BLOB.

• For any other data type, use the GET CLIPBOARD command, which returns the data into a BLOB.

4th Dimension provides the following predefined constants:

Constant	Type	Value
No such data in clipboard	Long Integer	-102
Text data	String	TEXT
Picture data	String	PICT

Examples

(1) The following code tests whether the Clipboard contains a picture and, if so, copies that picture into a 4D variable:

```
⇒      If (Test clipboard (Picture data) > 0)  ` Is there a picture in the clipboard?
          GET PICTURE FROM CLIPBOARD ($vPicVariable)  ` If so, extract the picture from
the clipboard
       Else
          ALERT("There is no picture in the clipboard.")
       End if
```

(2) Usually, applications cut and copy data of type TEXT or PICT into the Clipboard, because most applications recognize two standard data types. However, an application can append to the Clipboard several instances of the same data in different formats. For example, each time you cut or copy a part of a spreadsheet, the spreadsheet application could append the data under the hypothetical 'SPSH' format, as well as in SYLK and TEXT formats. The 'SPSH' instance would contain the data formatted using the application's data structure. The SYLK form would contain the same data, but using the SYLK format recognized by most of the other spreadsheet programs. Finally, the TEXT format would contain the same data, without the extra information included in the SYLK or the hypothetical 'SPSH' format. At this point, while writing Cut/Copy/Paste routines between 4th Dimension and that hypothetical spreadsheet application, assuming you know the description of the 'SPSH' format and that you are ready to parse SYLK data, you could write something like:

```
       Case of
             ` First, check whether the clipboard contains data
             ` from the hypothetical spreadsheet application
⇒         : (Test clipboard ('SPSH') > 0)
             ` ...
             ` Second, check whether the clipboard contains Sylk data
⇒         : (Test clipboard ('SYLK') > 0)
             ` ...
             ` Finally check whether the clipboard contains Text data
⇒         : (Test clipboard ('TEXT') > 0)
             ` ...
       End case
```

In other words, you try to extract from the Clipboard the instance of the data that carries most of the original information.

(3) See the example for the APPEND TO CLIPBOARD command.

See Also

GET CLIPBOARD, GET PICTURE FROM CLIPBOARD, Get text from clipboard.

8 Communications

SET CHANNEL (port | operation{; settings | document})

Parameter	**Type**		**Description**
port | operation	Number	→	Serial port number, or Document operation to perform
settings | document	Number | String	→	Serial port settings, or Document name

Description

The SET CHANNEL command opens a serial port or a document. You can open only one serial port or one document at a time with this command.

Historical Note: This command was originally the first 4D command used for working with serial ports and documents on disks. Since that time, new commands have been added. Today, you will typically work with documents on disk using the commands Open document, Create document and Append document. With these commands, you can read and write characters to and from documents using SEND PACKET or RECEIVE PACKET (these commands work with SET CHANNEL, too). However, if you want to use the commands SEND VARIABLE, RECEIVE VARIABLE, SEND RECORD and RECEIVE RECORD, you must use SET CHANNEL to access the document on disk.

The description of SET CHANNEL is composed of two sections:
• Working with Serial Ports
• Working with Documents

Working with Serial Ports - SET CHANNEL (port;settings)

The first form of the SET CHANNEL command opens a serial port, setting the protocol and other port information. Data can be sent with SEND PACKET, SEND RECORD or SEND VARIABLE, and received with RECEIVE BUFFER, RECEIVE PACKET, RECEIVE RECORD or RECEIVE VARIABLE.

The port Parameter

The first parameter, port, selects the port and the protocol.

On Windows:

You can address up to 99 serial ports (one at a time). The following table lists the values for port:

Range	Description
101 to 199	Serial communication with no protocol
201 to 299	Serial communication with software protocol such as XON/XOFF
301 to 399	Serial communication with hardware protocol such as RTS/CTS

Important: The value you pass in port must refer to an existing serial COM port recognized by the Windows session. For example, in order to be able to use the values 101, 103 and 125, the serial ports COM1, COM3 and COM25 must have been set up correctly.

On Macintosh:

You determine the value for port by adding the serial port and protocol values as listed in the following table.

	Value to accumulate in port parameter	Description
Serial Port	0	Macintosh Printer Port (or Windows COM2)
	1	Macintosh Modem Port (or Windows COM1)
Protocol	0	None
	20	XON/XOFF
	30	DTR

For example, to use XON/XOFF with the modem port, you would add 1 + 20 = 21. You would then use 21 as the value of the port parameter. For code compatibility across platforms, the port values as used on Macintosh are redirected as follows on Windows:

port value	Description
0	COM2
1	COM1
20	COM2 (with software protocol such as XON/XOFF)
21	COM1 (with software protocol such as XON/XOFF)
30	COM2 (with hardware protocol such as RTS/CTS)
31	COM1 (with hardware protocol such as RTS/CTS)

The settings Parameter

The settings parameter sets the speed, number of data bits, number of stop bits, and parity. You determine the value for settings by adding the speed, data bits, stop bits, and parity values as listed in the following table. For example, to set 1200 baud, 8 data bits, 1 stop bit, and no parity, you would add 94 + 3072 + 16384 + 0 = 27742. You would then use 27742 as the value of the setup parameter.

	Value to accumulate in settings parameter	**Description**
Speed (in baud)	380	300
	189	600
	94	1200
	62	1800
	46	2400
	30	3600
	22	4800
	14	7200
	10	9600
	4	19200
	0	57600
	1022	115200
	1021	230400
Data bits	0	5
	2048	6
	1024	7
	3072	8
Stop bits	16384	1
	–32768	1.5
	–16384	2
Parity	0	None
	4096	Odd
	12288	Even

Tip: The various numeric values to be accumulated and passed in port and settings (but not including the values for COM1...COM99) are available as predefined constants in the theme Communications within the Design environment Explorer windows. For COM1...COM99, use numeric literals.

Working with Documents on Disk - SET CHANNEL(operation;document)

The second form of the SET CHANNEL command allows you to create, open, and close a document. Unlike the System documents commands, it can open only one document at a time. The document can be read from or written to.

The operation parameter specifies the operation to be performed on the document specified by document. The following table lists the values of operation and the resulting operation with different values for document. The first column lists the allowed values for operation. The second column lists the allowed values for document. The third column lists the resulting operation.
For example, to display an Open File dialog box to open a text file, you would use the following line:

⇒ **SET CHANNEL** (13; "")

Operation	Document	Result
10	String	Opens the document specified by String. If the document doesn't exist, the document is opened and created.
10	"" (empty string)	Displays the Open File dialog box to open a file. All file types are displayed.
11	none	Closes an open file.
12	"" (empty string)	Displays the Save File dialog box to create a new file.
13	"" (empty string)	Displays the Open File dialog box to open a file. Only text file types are displayed.

All of the operations in this table set the Document system variable if appropriate. They also set the OK system variable to 1 if the operation was successful. Otherwise, the OK system variable is set to 0.

Examples

See examples for the commands RECEIVE BUFFER, SET TIMEOUT and RECEIVE RECORD.

See Also

Append document, Create document, Open document, RECEIVE BUFFER, RECEIVE PACKET, RECEIVE RECORD, RECEIVE VARIABLE, SEND PACKET, SEND RECORD, SEND VARIABLE, SET TIMEOUT.

SET TIMEOUT

SET TIMEOUT (seconds)

Parameter	Type		Description
seconds	Number	→	Seconds until the timeout

Description

SET TIMEOUT specifies how much time a serial port command has to complete. If the serial port command does not complete within the specified time, seconds, the serial port command is canceled, an error -9990 is generated, and the OK system variable is set to 0. You can catch the error with an error-handling method installed using ON ERR CALL.

Note that the time is the total time allowed for the command to execute, not the time between characters received. To cancel a previous setting and stop monitoring serial port communication, use a setting of 0 for seconds.

The commands that are affected by the timeout setting are:
• RECEIVE PACKET
• RECEIVE RECORD
• RECEIVE VARIABLE

Example

The following example sets the serial port to receive data. It then sets a time-out. The data is read with RECEIVE PACKET. If the data is not received in time, an error occurs:

```
        SET CHANNEL (MacOS Serial Port; Speed 9600 +
                                Data Bits 8 + Stop Bits One + Parity None) ` Open Serial Port
⇒       SET TIMEOUT (10)  ` Set the timeout for 10 seconds
        ON ERR CALL ("CATCH COM ERRORS")   ` Do not let the method being interrupted
        RECEIVE PACKET (vtBuffer; Char (13))  ` Read until a carriage return is met
        If (OK=0)
           ALERT ("Error receiving data.")
        Else
           [People]Name:=vtBuffer  ` Save received data in a field
        End if
        ON ERR CALL("")
```

See Also

ON ERR CALL, RECEIVE BUFFER, RECEIVE PACKET, RECEIVE RECORD, RECEIVE VARIABLE.

USE ASCII MAP (map | *; mapInOut)

Parameter	Type		Description
map | *	String | *	→	Document name of the map to use, or * to reset to default ASCII map
mapInOut	Number	→	0 = Output map 1 = Input map

Description

USE ASCII MAP has two forms. The first form loads the ASCII map named map from disk and uses that ASCII map. If mapInOut is 0, the map is loaded as the output map. If mapInOut is 1, the map is loaded as the input map.

The ASCII map must have been previously created with the ASCII map dialog box in the User environment. After an ASCII map is loaded, 4th Dimension uses the map during transfer of data between the database and a document or a serial port. Transfer operations include the import and export of text (ASCII), DIF, and SYLK files. An ASCII map also works on data transferred with SEND PACKET, RECEIVE PACKET, and RECEIVE BUFFER. It has no effect on transfers of data done with SEND RECORD, SEND VARIABLE, RECEIVE RECORD, and RECEIVE VARIABLE.

If you give an empty string for map, USE ASCII MAP displays a standard Open File dialog box so that the user can specify an ASCII map document. Whenever you execute USE ASCII MAP, the OK system variable is set to 1 if the map is successfully loaded, and to 0 if it is not.

The second form of USE ASCII MAP, with the asterisk (*) parameter instead of map, restores the default ASCII map. If mapInOut is 0, the map is reset for output. If mapInOut is 1, the map is reset for input. The default ASCII map has no translation between characters.

Example

The following example loads a special ASCII map from disk. It then exports data. Finally, the default ASCII map is restored:

```
⇒	USE ASCII MAP ("MactoPC"; 0)   ` Load an alternative ASCII map
	EXPORT TEXT ([MyTable]; "MyText")   ` Export data through the map
⇒	USE ASCII MAP (*; 0)   ` Restore the default map
```

See Also

EXPORT DIF, EXPORT SYLK, EXPORT TEXT, IMPORT DIF, IMPORT SYLK, IMPORT TEXT, Mac to Win, RECEIVE BUFFER, RECEIVE PACKET, SEND PACKET, Win to Mac.

SEND PACKET

SEND PACKET ({docRef; }packet)

Parameter	Type		Description
docRef	DocRef	→	Document reference number, or Current channel (serial port or document)
packet	String	→	String or Text to be sent

Description

SEND PACKET sends a packet to a serial port or to a document. If docRef is specified, the packet is written to the document referenced by docRef. If docRef is not specified, the packet is written to the serial port or document previously opened by the SET CHANNEL command. A packet is just a piece of data, generally a string of characters.

Before you use SEND PACKET, you must open a serial port or a document with SET CHANNEL, or open a document with one of the document commands.

When writing to a document, the first SEND PACKET begins writing at the beginning of the document unless the document was opened with Append document. Until the document is closed, each subsequent packet is appended to any previously sent packets.

Version 6 Note: This command is still useful for a document opened with SET CHANNEL. On the other hand, for a document opened with Open document, Create document and Append document, you can now use the new commands Get document position and SET DOCUMENT POSITION to get and change the location in the document where the next writing (SEND PACKET) or reading (RECEIVE PACKET) will occur.

Important: SEND PACKET writes Windows ASCII data on Windows and Macintosh ASCII data on Macintosh. Each of these uses eight bits. Standard ASCII uses only the lower seven bits. Many devices do not use the eighth bit in the same way as does Windows/Macintosh. If the string to be sent contains data that uses the eighth bit, be sure to create an ASCII map to translate the ASCII characters, and execute USE ASCII MAP before using SEND PACKET. Protocols like XON/XOFF use some low ASCII codes to establish communication between machines. Be careful to not send such ASCII codes, as this may interfere with the protocol or even break communication.

Example

The following example writes data from fields to a document. It writes the fields as fixed-length fields. Fixed-length fields are always of a specific length. If a field is shorter than the specified length, the field is padded with spaces. (That is, spaces are added to make up the specified length.) Although the use of fixed-length fields is an inefficient method of storing data, some computer systems and applications still use them:

```
      $vhDocRef := Create document ("")  ` Create a document
      If (OK=1)  ` Was the document created?
         For ($vlRecord; 1; Records in selection ([People]))  ` Loop once for each record
                ` Send a packet. Create the packet from a string of 15 spaces
                ` containing the first name field
⇒           SEND PACKET ($vhDocRef; Change string(15 * Char(Space); [People]First;1))
                ` Send a second packet. Create the packet from a string of 15 spaces
                ` containing the last name field
                ` This could be in the first SEND PACKET, but is separated for clarity
⇒           SEND PACKET ($vhDocRef; Change string (15 * Char(Space); [People]Last; 1))
            NEXT RECORD([People])
         End for
            ` Send a Char(26), which is used as an end-of-file marker for some computers
⇒        SEND PACKET ($vhDocRef; Char(SUB ASCII Code))
         CLOSE DOCUMENT ($vhDocRef)  ` Close the document
      End if
```

See Also

Get document position, RECEIVE PACKET, SET DOCUMENT POSITION.

RECEIVE PACKET

version 3

RECEIVE PACKET ({docRef; }receiveVar; stopChar | numChars)

Parameter	Type		Description
docRef	DocRef	→	Document reference number, or Current channel (serial port or document)
receiveVar	Variable	→	Variable to receive data
stopChar | numChars	String | Number	→	Character at which to stop receiving, or Number of characters to receive

Description

RECEIVE PACKET reads characters from a serial port or from a document.

If docRef is specified, this command reads characters from a document opened using Open document, Create document or Append document. If docRef is omitted, this command reads characters from the serial port or the document opened using SET CHANNEL.

Whatever the source, the characters read are returned in receiveVar, which must be a Text or String variable. To read a particular number of characters, pass this number in numChars. To read characters until a particular character is encountered, pass this character in stopChar (the stop character is not returned in receiveVar).

When reading a document, if stopChar | numChars is not specified, RECEIVE PACKET will stop reading at the end of the document. However, remember that while a string variable has a fixed length, a text variable accepts up to 32000 characters. When reading from a serial port, RECEIVE PACKET will attempt to wait indefinitely until the timeout (if any) has elapsed (see SET TIMEOUT) or until the user interrupts the reception (see below).

During execution of RECEIVE PACKET, the user can interrupt the reception by pressing Ctrl-Alt-Shift (Windows) or Command-Option-Shift (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using ON ERR CALL. Usually, you will only have to handle interruption of a reception when communicating over a serial port.

When reading a document, the first RECEIVE PACKET begins reading at the beginning of the document. The reading of each subsequent packet begins at the character following the last character read.

Version 6 Note: This command is still useful for document opened with SET CHANNEL. On the other hand, for a document opened with Open document, Create document and Append document, you can now use the new commands Get document position and SET DOCUMENT POSITION to get and change the location in the document where the next writing (SEND PACKET) or reading (RECEIVE PACKET) will occur.

When attempting to read past the end of a file, RECEIVE PACKET will return with the data read up to that point and the variable OK will be set to 1. Then, the next RECEIVE PACKET will return an empty string and set the OK variable to zero.

Examples

1. The following example reads 20 characters from a serial port into the variable getTwenty:

```
⇒      RECEIVE PACKET (getTwenty; 20)
```

2. The following example reads data from the document referenced by the variable myDoc into the variable vData. It reads until it encounters a carriage return:

```
⇒      RECEIVE PACKET (myDoc;vData;Char (Carriage Return))
```

3. The following example reads data from a document into fields. The data is stored as fixed-length fields. The method calls a subroutine to strip any trailing spaces (spaces at the end of the string). The subroutine follows the method:

```
       $vhDocRef := Open document ("";"TEXT")  ` Open a TEXT document
       If (OK=1)  ` If the document was opened
          Repeat  ` Loop until no more data
⇒            RECEIVE PACKET ($vhDocRef; $Var1; 15)  ` Read 15 characters
⇒            RECEIVE PACKET ($vhDocRef; $Var2; 15)  ` Do same as above for second field
             If (OK = 1)  ` If we are not beyond the end of the document
                CREATE RECORD([People])  ` Create a new record
                [People]First := Strip ($Var1)  ` Save the first name
                [People]Last := Strip ($Var2)  ` Save the last name
                SAVE RECORD([People])  ` Save the record
             End if
          Until (OK =0)
          CLOSE DOCUMENT ($vhDocRef)  ` Close the document
       End if
```

The spaces at the end of the data are stripped by the following method, called Strip:

```
For ($i; Length ($1); 1; –1)  ` Loop from end of string to start
    If ($1[[$i]] # " ")  ` If it is not a space…
        $i := -$i  ` Force the loop to end
    End if
End for
$0 := Delete string ($1; –$i; Length ($1))  ` Delete the spaces
```

See Also

Get document position, RECEIVE PACKET, SEND PACKET, SET DOCUMENT POSITION, SET TIMEOUT.

System Variables or Sets

After a call to RECEIVE PACKET, the OK system variable is set to 1 if the packet is received without error. Otherwise, the OK system variable is set to 0.

RECEIVE BUFFER

RECEIVE BUFFER (receiveVar)

Parameter	Type		Description
receiveVar	Variable	→	Variable to receive data

Description

RECEIVE BUFFER reads the serial port that was previously opened with SET CHANNEL. The serial port has a buffer that fills with characters until a command reads from the buffer. RECEIVE BUFFER gets the characters from the serial buffer, put them into receiveVar then clears the buffer. If there are no characters in the buffer, then receiveVar will contain nothing.

On Windows:
The Windows serial port buffer is limited in size. This means that the buffer can overflow. When it is full and new characters are received, the new characters replace the oldest characters. The old characters are lost; therefore, it is essential that the buffer is read quickly when new characters are received.

On Macintosh
The Macintosh serial port buffer is 64 characters in size. This means that the buffer can hold 64 characters before it overflows. When it is full and new characters are received, the new characters replace the oldest characters. The old characters are lost; therefore, it is essential that the buffer is read quickly when new characters are received.

Note: There are 4D plug-ins that enable you to increase the size of the serial buffer.

RECEIVE BUFFER is different from RECEIVE PACKET in that it takes whatever is in the buffer and then immediately returns. RECEIVE PACKET waits until it finds a specific character or until a given number of characters are in the buffer.

During the execution of RECEIVE BUFFER, the user can interrupt the reception by pressing Ctrl-Alt-Shift (Windows) or Command-Option-Shift (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using ON ERR CALL.

Example

The project method LISTEN TO SERIAL PORT uses RECEIVE BUFFER to get text from the serial port and accumulate it into a an interprocess variable:

```
    ` LISTEN TO SERIAL PORT
While (<>IP_Listen_Serial_Port)
    RECEIVE BUFFER($vtBuffer)
    If ((Length($vtBuffer)+Length(<>vtBuffer))>MAXTEXTLEN)
        <>vtBuffer:=""
    End if
    <>vtBuffer:=<>vtBuffer+$Buffer
End while
```

This method can be executed as a process method for a local process:

```
    ` Start listening to the serial port
SET CHANNEL (201; Speed 9600 + Data Bits 8 + Stop Bits One
                                            + Parity None) ` Open Serial Port
<>IP_Listen_Serial_Port:=True
$vlSerialPID:=New process("LISTEN TO SERIAL PORT";16*1024;"$Serial Port Listener")
```

At this point, any other process can read the interprocess <>vtBuffer to work with the data coming from the serial port.

To stop listening to the serial port, just execute:

```
    ` Stop listening to the serial port
<>IP_Listen_Serial_Port:=False
```

Note that access to the interprocess <>vtBuffer variable should be protected by a semaphore, so that processes will not conflict. See the command Semaphore for more information.

See Also

ON ERR CALL, RECEIVE PACKET, Semaphore, SET CHANNEL, Variables.

SEND VARIABLE

SEND VARIABLE (variable)

Parameter	Type		Description
variable	Variable	→	Variable to send

Description

SEND VARIABLE sends variable to the document or serial port previously opened by SET CHANNEL. The variable is sent with a special internal format that can be read only by RECEIVE VARIABLE. SEND VARIABLE sends the complete variable (including its type and value).

Notes

1. If you send a variable to a document using this command, the document must have been opened using the SET CHANNEL command. You cannot use SEND VARIABLE with a document opened with Open document, Append document or Create document.
2. This command does not support array variables. If you want to send and receive arrays from a document or over a serial port, use the new BLOB commands introduced in version 6.

Example

See example for the command RECEIVE RECORD.

See Also

RECEIVE RECORD, RECEIVE VARIABLE, SEND RECORD, SET CHANNEL.

RECEIVE VARIABLE

version 3

RECEIVE VARIABLE (variable)

Parameter	Type		Description
variable	Variable	→	Variable in which to receive

Description

RECEIVE VARIABLE receives variable, which was previously sent by SEND VARIABLE from the document or serial port previously opened by SET CHANNEL.

In interpreted mode, if the variable does not exist prior to the call to RECEIVE VARIABLE, the variable is created, typed and assigned according to what has been received. In compiled mode, the variable must be of the same type as what is received. In both cases, the contents of the variable are replaced with what is received.

Notes

1. If you receive a variable from a document using this command, the document must have been opened using the SET CHANNEL command. You cannot use RECEIVE VARIABLE with a document opened with Open document, Append document or Create document.
2. This command does not support array variables. If you want to send and receive arrays from a document or over a serial port, use the new BLOB commands introduced in version 6.
3. During the execution of RECEIVE VARIABLE, the user can interrupt the reception by pressing Ctrl-Alt-Shift (Windows) or Command-Option-Shift (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using ON ERR CALL. Usually, you only need to handle the interruption of a reception while communicating over a serial port.

Example

See example for the command RECEIVE RECORD.

See Also

ON ERR CALL, RECEIVE RECORD, SEND RECORD, SEND VARIABLE.

System Variables or Sets

The OK system variable is set to 1 if the variable is received. Otherwise, the OK system variable is set to 0.

SEND RECORD

SEND RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table from which to send the current record, or
			Default table, if omitted

Description

SEND RECORD sends the current record of table to the serial port or document opened by the SET CHANNEL command. The record is sent with a special internal format that can be read only by RECEIVE RECORD. If no current record exists, SEND RECORD has no effect.

The complete record is sent. This means that all subrecords, pictures and BLOBs stored in the record are also sent.

Important: When records are being sent and received using SEND RECORD and RECEIVE RECORD, the source table structure and the destination table structure must be compatible. If they are not, 4D will convert values according to the table definitions when RECEIVE RECORD is executed.

Note: If you send a record to a document using this command, the document must have been opened using the SET CHANNEL command. You cannot use SEND RECORD with a document opened with Open document, Append document or Create document.

Example

See example for the command RECEIVE RECORD.

See Also

RECEIVE RECORD, RECEIVE VARIABLE, SEND VARIABLE.

RECEIVE RECORD

version 3

RECEIVE RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table into which to receive the record, or Default table, if omitted

Description

RECEIVE RECORD receives a record into table from the serial port or document opened by the SET CHANNEL command. The record must have been sent with SEND RECORD. When you execute RECEIVE RECORD, a new record is automatically created for table. If the record is received correctly, you must then use SAVE RECORD to save the new record.

The complete record is received. This means that all subrecords, pictures and BLOBs stored in the record are also received.

Important: When records are being sent and received using SEND RECORD and RECEIVE RECORD, the source table structure and the destination table structure must be compatible. If they are not, 4D will convert values according to the table definitions when RECEIVE RECORD is executed.

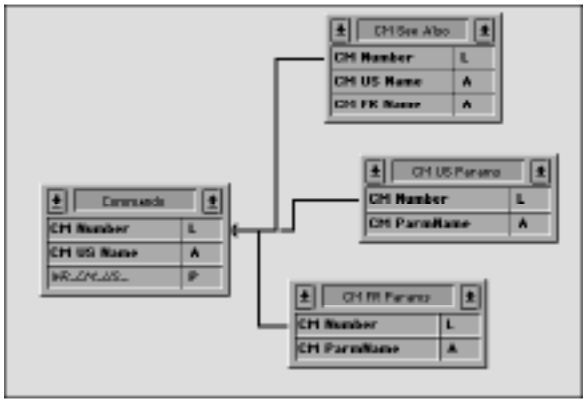
Notes

1. If you receive a record from a document using this command, the document must have been opened using the SET CHANNEL command. You cannot use RECEIVE RECORD with a document opened with Open document, Append document or Create document.
2. During the execution of RECEIVE RECORD, the user can interrupt the reception by pressing Ctrl-Alt-Shift (Windows) or Command-Option-Shift (Macintosh). This interruption generates an error -9994 that you can catch with an error-handling method installed using ON ERR CALL. Usually, you only need to handle the interruption of a reception while communicating over a serial port.

Example

A combined use of SEND VARIABLE, SEND RECORD, RECEIVE VARIABLE and RECEIVE RECORD is ideal for archiving data or for exchanging data between identical single-user databases used in different places. You can exchange data between 4D databases using the import/export commands such as EXPORT TEXT and IMPORT TEXT. However, if your data contains graphics, subtables and/or related tables, using SEND RECORD and RECEIVE RECORD is far more convenient.

For instance, the documentation you are currently reading has been created using 4D and 4D Write. Because several writers in different locations wordwide were working on it, we needed a simple way to exchange data between the different databases. Here is a simplified view of the database structure:



The table [Commands] contains the description of each command or topic. The tables [CM US Params] and [CM FR Params] respectivily contain the parameter list for each command in English and in French. The table [CM See Also] contains the commands listed as reference (See Also section) for each command. Exchanging documentation between databases therefore consists in sending the [Commands] records and their related records. To do so, we use SEND RECORD and RECEIVE RECORD. In addition, we use SEND VARIABLE and RECEIVE VARIABLE in order to mark the import/export document with tags.

Here is the (simplified) project method for exporting the documentation:

```
          ` CM_EXPORT_SEL project method
          ` This method works with the current selection of the [Commands] table

⇒     SET CHANNEL(12;"") ` Let's the user create an open a channel document
      If (OK=1)
              ` Tag the document with a variable that indicates its contents
              ` Note: the BUILD_LANG process variable indicates
              ` if US (English) or FR (French) data is sent
          $vsTag:="4DV6COMMAND"+BUILD_LANG
⇒         SEND VARIABLE($vsTag)
              ` Send a variable indicationg how many [Commands] are sent
          $vlNbCmd:=Records in selection([Commands])
⇒         SEND VARIABLE($vlNbCmd)
          FIRST RECORD([Commands])
          ` For each command
        For ($vlCmd;1;$vlNbCmd)
              ` Send the [Commands] record
⇒           SEND RECORD([Commands])
              ` Select all the related records
            RELATE MANY([Commands])
              ` Depending on the language, send a variable indicating
              ` the number of parameters that will follow
            Case of
              : (BUILD_LANG="US")
                  $vlNbParm:=Records in selection([CM US Params])
              : (BUILD_LANG="FR")
                  $vlNbParm:=Records in selection([CM FR Params])
            End case
⇒           SEND VARIABLE($vlNbParm)
              ` Send the parameter records (if any)
            For ($vlParm;1;$vlNbParm)
                Case of
                  : (BUILD_LANG="US")
⇒                       SEND RECORD([CM US Params])
                        NEXT RECORD([CM US Params])
                  : (BUILD_LANG="FR")
⇒                       SEND RECORD([CM FR Params])
                        NEXT RECORD([CM FR Params])
                 End case
            End for
              ` Send a variable indicating how many "See Also" will follow
            $vlNbSee:=Records in selection([CM See Also])
⇒           SEND VARIABLE($vlNbSee)
              ` Send the [See Also] records (if any)
            For ($vlSee;1;$vlNbSee)
⇒               SEND RECORD([CM See Also])
                NEXT RECORD([CM See Also])
            End for
              ` Go to the next [Commands] record and continue the export
            NEXT RECORD([Commands])
        End for
⇒       SET CHANNEL(11) ` Close the document
    End if
```

Here is the (simplified) project method for importing the documentation:

```
          ` CM_IMPORT_SEL project method

⇒    SET CHANNEL(10;"")   ` Let's user open an existing document
     If (OK=1) ` If a document was open
⇒       RECEIVE VARIABLE($vsTag)   ` Try receiving the expected tag variable
        If ($vsTag="4DV6COMMAND@")   ` Did we get the right tag?
              ` Extract language from the tag
           $CurLang:=Substring($vsTag;Length($vsTag)-1)
           If (($CurLang="US") | ($CurLang="FR"))   ` Did we get a valid language
                 ` How many commands are there in this document?
⇒             RECEIVE VARIABLE($vlNbCmd)
              If ($vlNbCmd>0)   ` If at least one
                 For ($vlCmd;1;$vlNbCmd)   ` For each archived [Commands] record
                       ` Receive the record
⇒                   RECEIVE RECORD([Commands])
                       ` Call a subroutine that saves the new record or copies its values
                       ` into an already existing record
                    CM_IMP_CMD ($CurLang)
                       ` Receive the number of parameters (if any)
⇒                   RECEIVE VARIABLE($vlNbParm)
                    If ($vlNbParm>=0)
                          ` Call a subroutine that calls RECEIVE RECORD then saves
                          ` the new records or copies them into already existing records
                       CM_IMP_PARM ($vlNbParm;$CurLang)
                    End if
                       ` Receive the number of “See Also” (if any)
⇒                   RECEIVE VARIABLE($vlNbSee)
                    If ($vlNbSee>0)
                          ` Call a subroutine that calls RECEIVE RECORD then saves
                          ` the new records or copies them into already existing records
                       CM_IMP_SEEA ($vlNbSee;$CurLang)
                    End if
                 End for
              Else
                 ALERT("The number of commands in this export document is invalid.")
              End if
           Else
              ALERT("The language of this export document is unkown.")
           End if
        Else
           ALERT("This document is NOT a Commands export document.")
        End if
⇒       SET CHANNEL(11)   ` Close document
     End if
```

Note that we do not test the OK variable while receiving the data nor try to catch the errors. However, because we stored variables in the document that describes the document itself, if these variables, once received, made sense, the probability for an error is very low. If for instance a user opens a wrong document, the first test stops the operation right away.

See Also

RECEIVE VARIABLE, SEND RECORD, SEND VARIABLE.

System Variables or Sets

The OK system variable is set to 1 if the record is received. Otherwise, the OK system variable is set to 0.

9 Compiler

Compiler Commands

Compiler

version 6.0

4D Compiler translates your database applications into assembly level instructions. The advantages of 4D Compiler are:

• **Speed:** Your database can run from 3 to 1,000 times faster.

• **Code checking:** Your database application is scanned for the consistency of code. Both logical and syntactical conflicts are detected.

• **Protection:** A compiled database is functionally identical to the original, except that the structure and procedures cannot be viewed or modified, deliberately or inadvertently. Compiling a database ensures security.

• **Stand-alone double-clickable applications:** 4D Compiler creates stand-alone applications (.EXE files) with their own custom icons.

The commands in this theme relate to the use of the compiler. They enable you to normalize data types throughout your database. The IDLE command is specifically used in compiled databases.

C_BLOB	C_INTEGER	C_REAL	IDLE
C_BOOLEAN	C_LONGINT	C_STRING	
C_DATE	C_PICTURE	C_TEXT	
C_GRAPH	C_POINTER	C_TIME	

These commands, except IDLE, declare variables and cast them as a specified data type. Declaring variables resolves ambiguities concerning a variable's data type. If a variable is not declared with one of these commands, the compiler attempts to determine a variable's data type. The data type of a variable used in a form is often difficult for the compiler to determine. Therefore, it is especially important that you use these commands to declare a variable used in a form.

Numeric operations on long integer and integer variables are usually much faster than operations on the default numeric type (real).

General rules about writing code that will be compiled

• Variable indirection as used in 4th Dimension version 1 is not allowed. You cannot use alpha indirection, with the section symbol (§), to indirectly reference variables. Nor can you use numeric indirection, with the curly braces ({...}), for this purpose. Curly braces can only be used when accessing specific elements of an array that has been declared. However, you can use parameter indirection, as described in the documentation for 4D Compiler.

• You can't change the data type of any variable or array.

• You can't change a one-dimensional array to a two-dimensional array, or change a two-dimensional array to a one-dimensional array.

• You can't change the length of string variables or of elements in string arrays.

• Although 4D Compiler will type the variable for you, you should specify the data type of a variable by using compiler directives where the data type is ambiguous, such as in a form.

• Another reason to explicitly type your variables is to optimize your code. This rule applies especially to any variable used as a counter. Use variables of a long integer data type for maximum performance.

• To clear a variable (initialize it to null), use CLEAR VARIABLE with the name of the variable. Do not use a string to represent the name of the variable in the CLEAR VARIABLE command.

• The Undefined function will always return False. Variables are always defined.

Examples

(1) The following are some basic variable declarations for 4D Compiler:

```
          ` The process variable vxMyBlob is declared as a variable of type BLOB
⇒      C_BLOB(vxMyBlob)
          ` The interprocess variable <>OnWindows is declared as a variable of type Boolean
⇒      C_BOOLEAN(<>OnWindows)
          ` The local variable $vdCurDate is declared as a variable of type Date
⇒      C_DATE($vdCurDate)
          ` The 3 process variables vg1, vg2 and vg3 are declared as variables of type Graph
⇒      C_GRAPH(vg1;vg2;vg3)
```

(2) In the following example, the project method OneMethodAmongOthers declares 3 parameters:

```
          ` OneMethodAmongOthers Project Method
          ` OneMethodAmongOthers ( Real ; Integer { ; Long } )
          ` OneMethodAmongOthers ( Amount ; Percentage { ; Ratio } )

⇒      C_REAL($1)          ` 1st parameter is of type Real
⇒      C_INTEGER($2)        ` 2nd parameter is of type Integer
⇒      C_LONGINT($3)        ` 3rd parameter is of type Long Integer

       ` ...
```

(3) In the following example, the project method Capitalize accepts a string parameter and returns a string result:

```
      ` Capitalize Project Method
      ` Capitalize ( String ) -> String
      ` Capitalize ( Source string ) -> Capitalized string

⇒   C_STRING(255;$0;$1)
    $0:=Uppercase(Substring($1;1;1))+Lowercase(Substring($1;2))
```

(4) In the following example, the project method SEND PACKETS accepts a time parameter followed by a variable number of text parameters:

```
      ` SEND PACKETS Project Method
      ` SEND PACKETS ( Time ; Text { ; Text2... ; TextN } )
      ` SEND PACKETS ( docRef ; Data { ; Data2... ; DataN } )

⇒   C_TIME ($1)
⇒   C_TEXT (${2})
⇒   C_LONGINT ($vlPacket)

    For ($vlPacket;2;Count parameters)
      SEND PACKET ($1;${$vlPacket})
    End for
```

(5) In the following example, the project method COMPILER_Param_Predeclare28 predeclares the syntax of other project methods for 4D Compiler

```
      ` COMPILER_Param_Predeclare28 Project Method

      ` OneMethodAmongOthers ( Real ; Integer { ; Long } )
⇒   C_REAL(OneMethodAmongOthers;$1)
⇒   C_INTEGER(OneMethodAmongOthers;$2)      ` ...
⇒   C_LONGINT(OneMethodAmongOthers;$3)      ` ...

      ` Capitalize ( String ) -> String
⇒   C_STRING(Capitalize;255;$0;$1)

      ` SEND PACKETS ( Time ; Text { ; Text2... ; TextN } )
⇒   C_TIME(SEND PACKETS;$1)
⇒   C_TEXT(SEND PACKETS;${2})    ` ...
```

See Also

C_BLOB, C_BOOLEAN, C_DATE, C_GRAPH, C_INTEGER, C_LONGINT, C_PICTURE, C_POINTER, C_REAL, C_STRING, C_TEXT, C_TIME, IDLE.

C_BLOB

version 6.0

C_BLOB ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

C_BLOB casts each specified variable as a BLOB variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with "COMPILER."

Advanced Tip: The syntax C_BLOB(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_BLOB(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands.

C_BOOLEAN

Compiler

version 3

C_BOOLEAN ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_BOOLEAN casts each specified variable as a Boolean variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_BOOLEAN(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_BOOLEAN(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands, Count parameters.

C_DATE

C_DATE ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_DATE casts each specified variable as a Date variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with "COMPILER."

Advanced Tip: The syntax C_DATE(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_DATE(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands, Count parameters.

C_GRAPH

C_GRAPH ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	String	→	Name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_GRAPH casts each specified variable as a Graph variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_GRAPH(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_GRAPH(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands.

C_INTEGER

C_INTEGER ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The C_INTEGER command casts each specified variable as an Integer variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_INTEGER(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_INTEGER(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler commands, Count parameters, C_LONGINT, C_REAL.

C_LONGINT

C_LONGINT ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The C_LONGINT command casts each specified variable as a Long Integer variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_LONGINT(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_LONGINT(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands, Count parameters, C_INTEGER, C_REAL.

C_PICTURE

C_PICTURE ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_PICTURE casts each specified variable as a Picture variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_PICTURE(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_PICTURE(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler commands, Count parameters.

C_POINTER

C_POINTER ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_POINTER casts each specified variable as a Pointer variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with "COMPILER."

Advanced Tip: The syntax C_POINTER(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_POINTER(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands, Count parameters.

C_REAL

C_REAL ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_REAL casts each specified variable as a Real variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_REAL(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_REAL(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands, Count parameters, C_INTEGER, C_LONGINT.

C_STRING

C_STRING ({method; }size; variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
size	Number	→	Size of the string
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_STRING casts each specified variable as a String variable.

The size parameter specifies the maximum length of the strings that the variable can contain. Strings are limited to 255 characters. If speed is a concern, use string variables rather than text variables wherever possible.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_STRING(...;${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_STRING(...;${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler commands, Count parameters, C_TEXT.

C_TEXT

C_TEXT ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_TEXT casts each specified variable as a Text variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_TEXT(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_TEXT(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler Commands, Count parameters, C_STRING.

C_TIME

C_TIME ({method; }variable{; variable2; ...; variableN})

Parameter	Type		Description
method	Method	→	Optional name of method
variable	Variable or ${...}	→	Name of variable(s) to declare

Description

The command C_TIME casts each specified variable as a Time variable.

The first form of the command, in which the optional method parameter is NOT passed, is used to declare and type any process, interprocess, or local variable.

Note: This form can be used in interpreted databases.

The second form of the command, in which the optional method parameter IS passed, is used to predeclare for 4D Compiler the result and/or the parameters ($0, $1, $2 etc) for a method. Use this form of the command in order to skip the Typing variables phase while compiling a database, saving compilation time.

WARNING: The second form cannot be executed in interpreted mode. For this reason, if you are using this syntax, keep it in a method that is not executed in interpreted mode. The name of this method must start with “COMPILER.”

Advanced Tip: The syntax C_TIME(${...}) allows you to declare a variable number of parameters of the same type, under the condition that these are the last parameters for the method. For example, the declaration C_TIME(${5}) tells 4D and 4D Compiler that starting with the fifth parameter, the method can receive a variable number of parameters of that type. For more information, see the Count parameters command.

Examples

See examples in the section Compiler Commands.

See Also

Compiler commands, Count parameters.

IDLE

Compiler

version 3

IDLE

Parameter	**Type**	**Description**
This command does not require any parameters		

Description

The IDLE command is designed only for 4D Compiler. This command is only used in compiled databases in which user-defined methods are written so that no calls are made back to the 4th Dimension engine. For example, if a procedure has a For loop in which no 4th Dimension commands are executed, the loop could not be interrupted by a process installed with ON SERIAL PORT CALL or ON EVENT CALL, nor could a user switch to another application. In this case, you should insert IDLE to allow 4th Dimension to trap events. If you do not want any interruptions, omit IDLE.

Examples

In the following example, the loop would never terminate in a compiled database without the call to IDLE:

```
          ` Do Something Project Method
       ON EVENT CALL ("EVENT METHOD")
       <>vbWeStop:=False
       MESSAGE ("Processing..."+Char(13)+"Type any key to interrupt...")
       Repeat
              ` Do some processing that doesn't involve a 4D command
⇒         IDLE
       Until (<>vbWeStop)
       ON EVENT CALL ("")
```

with:

```
          ` EVENT METHOD Project Method
       If (Undefined(KeyCode))
          KeyCode:=0
       End if
       If (KeyCode#0)
          CONFIRM ("Do you really want to stop this operation?")
          If (OK=1)
              <>vbWeStop:=True
          End if
       End if
```

See Also

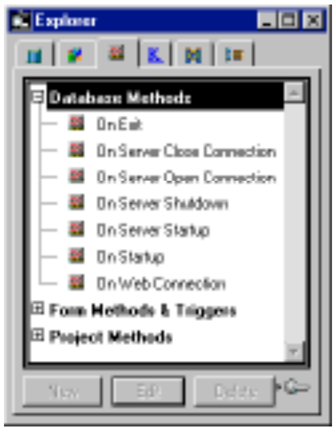
Compiler commands, ON EVENT CALL, ON SERIAL PORT CALL.

10 Database Methods

Database Methods

version 6.0

Database methods are methods that are automatically executed by 4th Dimension when a general session event occurs.



To create or open and edit a database method:

1. Open the **Explorer** window.
2. Select the **Methods** tab.
3. Expand the **Database Methods** theme.
4. Double click on the method.

or:

1. Select the method.
2. Click **Edit** or press Enter or Return.

You edit a database method in the same way as any other method.

You cannot call a database method from another method. Database methods are automatically invoked by 4th Dimension at certain points in a working session. The following table summarizes execution of database methods:

Database Method	**4th Dimension**	**4D Server**	**4D Client**
On Startup	Yes, Once	No	Yes, Once
On Exit	Yes, Once	No	Yes, Once
On Web Connection	Yes, Multiple	Yes, Multiple	No
On Server Startup	No	Yes, Once	No
On Server Shutdown	No	Yes, Once	No
On Server Open Connection	No	Yes, Multiple	No
On Server Close Connection	No	Yes, Multiple	No

For detailed information about each of the database methods, see the following sections:

- On Startup Database Method
- On Exit Database Method
- On Web Connection Database Method
- On Server Startup Database Method (*4D Server Reference* manual)
- On Server Shutdown Database Method (*4D Server Reference* manual)
- On Server Open Connection Database Method (*4D Server Reference* manual)
- On Server Close Connection Database Method (*4D Server Reference* manual)

See Also

Methods.

On Startup Database Method

version 6.0

The On Startup Database Method is called once when you open a database.

This occurs in the following 4D environments:
• 4th Dimension
• 4D Client (on the client side, after the connection has been accepted by 4D Server)
• 4D Runtime
• 4D application compiled and merged with 4D Compiler and 4D Engine

Note: The On Startup Database Method is NOT invoked by 4D Server.

The On Startup Database Method is automatically invoked by 4D; unlike project methods, you cannot call this database method yourself. To call and perform tasks from within the On Startup Database Method, as well as from project methods later on, use subroutines.

The On Startup Database Method is the perfect place to:
• Initialize interprocess variables that you will use during the whole working session.
• Start processes automatically when a database is opened.
• Load Preferences or Settings saved for this purpose during the previous working session.
• Prevent the opening of the database if a condition is not met (i.e., missing system resources) by explicitly calling QUIT 4D.
• Perform any other actions that you want to be performed automatically each time a database is opened.

Compatibility with previous versions of 4D

Database methods are a new type of method introduced in version 6. In previous versions of 4th Dimension, there was only one method (procedure) that 4D automatically executed when you opened a database. This procedure had to be called STARTUP (US English INTL version) or DEBUT (French version) in order to be invoked. If you have converted a version 3 database to version 6, and if you want to take advantage of the new On Startup Database Method capability, you must deselect the **Use Old Startup Method** property in the **Database Properties** dialog box (shown in this section). This property only affects the STARTUP/On Startup Database Method alternative. If you do not deselect this property and add, for instance, an On Exit Database Method, this latter will be invoked by 4D.

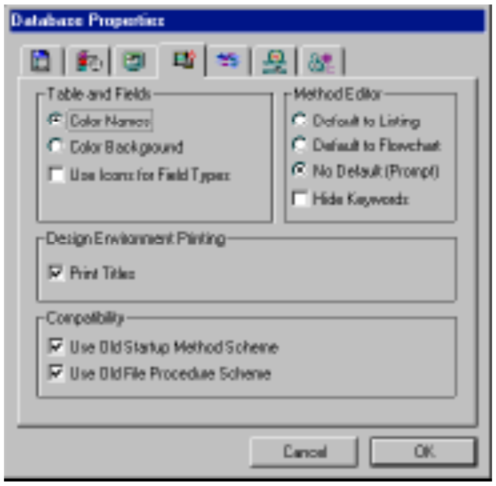

Database Properties

Table and Fields
- [✓ Color Names]
- Color Background
- ☐ Use Icons for Field Types

Method Editor
- Default to Listing
- Default to Flowchart
- [✓ No Default (Prompt)]
- ☐ Hide Keywords

Design Environment Printing
- ☑ Print Titles

Compatibility
- ☑ Use Old Startup Method Scheme
- ☑ Use Old File Procedure Scheme

Example

See the example in the section On Exit Database Method.

See Also

Database Methods, Methods, On Exit Database Method, QUIT 4D.

On Exit Database Method

The On Exit Database Method is called once when you quit a database.

This method is used in the following 4D environments:
• 4th Dimension
• 4D Client (on the client side)
• 4D Runtime
• 4D application compiled and merged with 4D Compiler and 4D Engine

Note: The On Exit Database Method is NOT invoked by 4D Server.

The On Exit Database Method is automatically invoked by 4D; unlike project methods, you cannot call this database method yourself. To call and perform tasks from within the On Startup Database Method, as well as from project methods, use subroutines.

A database can be exited if any of the following occur:
• The user selects the menu command **Quit** from the User or Design Environment **File** menu
• A call to the QUIT 4D command is issued
• A 4D Plug-in issues a call to the QUIT 4D entry point

No matter how the exit from the database was initiated, 4D performs the following actions:

• If there is no On Exit Database Method, 4D aborts each running process one by one, without distinction. If the user is performing data entry, the records will be cancelled and not saved.

• If there is an On Exit Database Method, 4D starts executing this method within a newly created local process. You can therefore use this database method to inform other processes, via interprocess communication, that they must close (data entry) or stop executing. Note that 4D will eventually quit—the On Exit Database Method can perform all the cleanup or closing operations you want, but it cannot refuse the quit, and will at some point end.

The On Exit Database Method is the perfect place to:
• Stop processes automatically started when the database was opened
• Save (locally, on disk) Preferences or Settings to be reused at the beginning of the next session in the On Startup Database Method
• Perform any other actions that you want to be done automatically each time a database is exited

Example

The following example covers all the methods used in a database that tracks the significant events that occur during a working session and writes a description in a text document called “Journal.”

• The On Startup Database Method initializes the interprocess variable <>vbQuit4D, which tells all the use processes whether or not the database is being exited. It also creates the journal file, if it does not already exist.

```
    ` On Startup Database Method
C_TEXT(<>vtIPMessage)
C_BOOLEAN(<>vbQuit4D)
<>vbQuit4D:=False

If (Test path name("Journal") # Is a document)
    $vhDocRef:=Create document("Journal")
    If (OK=1)
        CLOSE DOCUMENT($vhDocRef)
    End if
End if
WRITE JOURNAL ("Opening Session")
```

• The project method WRITE JOURNAL, used as subroutine by the other methods, writes the information it receives, in the journal file:

```
    ` WRITE JOURNAL Project Method
    ` WRITE JOURNAL ( Text )
    ` WRITE JOURNAL ( Event description )
C_TEXT($1)
C_TIME($vhDocRef)

While (Semaphore("$Journal"))
    DELAY PROCESS(Current process;1)
End while
$vhDocRef:=Append document("Journal")
If (OK=1)
    PROCESS PROPERTIES(Current process;$vsProcessName;$vlState;
                                                        $vlElapsedTime;$vbVisible)
    SEND PACKET($vhDocRef;String(Current date)+Char(9)
                     +String(Current time)+Char(9)+String(Current process)+Char(9)
                                        +$vsProcessName+Char(9)+$1+Char(13))
    CLOSE DOCUMENT($vhDocRef)
End if
CLEAR SEMAPHORE("$Journal")
```

Note that the document is open and closed each time. Also note the use of a semaphore as “access protection” to the document—we do not want two processes trying to access the journal file at the same time.

• The M_ADD_RECORDS project method is executed when a menu item **Add Record** is chosen in Custom menus:

```
    ` M_ADD_RECORDS Project Method

MENU BAR(1)
Repeat
    ADD RECORD([Table1];*)
    If (OK=1)
        WRITE JOURNAL ("Adding record #"+String(Record number([Table1]))
                                                        +" in Table1")
    End if
Until ((OK=0) | <>vbQuit4D)
```

This method loops until the user cancels the last data entry or exits the database.

• The input form for [Table 1] includes the treatment of the On Outside Call events. So, even if a process is in data entry, it can be exited smoothly, with the user either saving (or not saving) the current data entry:

```
    ` [Table1];"Input" Form Method
Case of
    : (Form event=On Outside Call)
        If (<>vtIPMessage="QUIT")
            CONFIRM("Do you want to save the changes made to this record?")
            If (OK=1)
                ACCEPT
            Else
                CANCEL
            End if
        End if
End case
```

• The M_QUIT project method is executed when **Quit** is chosen from the **File** menu in the Custom Menus environment:

```
    ` M_QUIT Project Method
$vlProcessID:=New process("DO_QUIT";32*1024;"$DO_QUIT")
```

The method uses a trick. When QUIT 4D is called, the command has an immediate effect. Therefore, the process from which the call is issued is in “stop mode” until the database is actually exited. Since this process can be one of the processes in which data entry occurs, the call to QUIT 4D is made in a local process that is started only for this purpose.

Here is the DO_QUIT method:

```
    ` DO_QUIT Project Method
CONFIRM("Are you sure you want to quit?")
If (OK=1)
    WRITE JOURNAL ("Quitting Database")
    QUIT 4D
        ` QUIT 4D has an immediate effect, any line of code below will never be executed
        ` ...
End if
```

• Finally, here is the On Exit Database Method which tells all open user processes “It's time to get out of here!” It sets <>vbQuit4D to True and sends interprocess messages to the user processes that are performing data entry:

```
    ` On Exit Database Method
<>vbQuit4D:=True
Repeat
    $vbDone:=True
    For ($vlProcess;1;Count tasks)
        PROCESS PROPERTIES($vlProcess;$vsProcessName;$vlState;
                                                        $vlElapsedTime;$vbVisible)
        If ((($vsProcessName="ML_@") | ($vsProcessName="M_@"))) & ($vlState>=0))
            $vbDone:=False
            <>vtIPMessage:="QUIT"
            BRING TO FRONT($vlProcess)
            CALL PROCESS($vlProcess)
            $vhStart:=Current time
            Repeat
                DELAY PROCESS(Current process;60)
            Until ((Process state($vlProcess)<0) |
                                            ((Current time-$vhStart)>=?00:01:00?))
        End if
    End for
Until ($vbDone)
WRITE JOURNAL ("Closing session")
```

Note: Processes that have names beginning with "ML_..." or "M_..." are started by menu commands for which the **Start a New Process** property has been selected. In this example, these are the processes started when the menu command **Add record** was chosen.

The test (Current time-$vhStart)>=?00:01:00? allows the database method to get out of the “waiting the other process” Repeat loop if the other process does not act immediately.

• The following is a typical example of the Journal file produced by the database:

```
2/6/97   15:47:25   1   User/Custom Menus process   Opening Session
2/6/97   15:55:43   5   ML_1                        Adding record #23 in Table1
2/6/97   15:55:46   5   ML_1                        Adding record #24 in Table1
2/6/97   15:55:54   6   $DO_QUIT                    Quitting Database
2/6/97   15:55:58   7   $xx                         Closing session
```

Note: The name $xx is the name of the local process started by 4D in order to execute the On Exit Database Method.

See Also

On Startup Database Method, QUIT 4D.

11 Data Entry

ADD RECORD

ADD RECORD ({table}{; }{*})

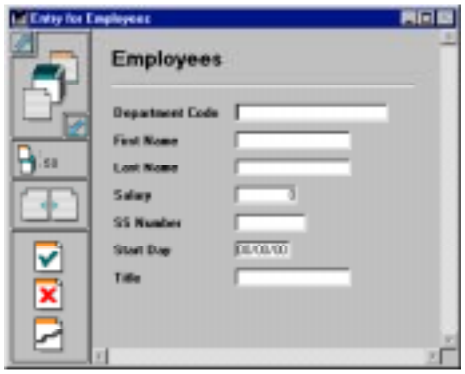
Parameter	Type		Description
table	Table	→	Table to use for data entry, or Default table, if omitted
*		→	Hide scroll bars

Description

The command ADD RECORD lets the user add a new record to the database for the table table or for the default table, if you omit the table parameter.

ADD RECORD creates a new record, makes the new record the current record for the current process, and displays the current input form. In the Custom Menus environment, after the user has accepted the new record, the new record is the only record in the current selection.

The following figure shows a typical data entry form.



The form is displayed in the frontmost window of the process. The window has scroll bars and a size box. Specifying the optional * parameter causes the window to be drawn without scroll bars or a size box.

ADD RECORD displays the form until the user accepts or cancels the record. If the user is adding several records, the command must be executed once for each new record.

The record is saved (accepted) if the user clicks an Accept button or presses the Enter key (numeric keypad), or if the ACCEPT command is executed.

The record is not saved (canceled) if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the CANCEL command is executed.

After a call to ADD RECORD, OK is set to 1 if the record is accepted, to 0 if canceled.

Note: Even when canceled, the record remains in memory and can be saved if SAVE RECORD is executed before the current record pointer is changed.

Examples

1. The following example is a loop commonly used to add new records to a database:

```
        INPUT FORM ([Customers];"Std Input")   ` Set input form for [Customers] table
        Repeat  ` Loop until the user cancels
⇒          ADD RECORD ([Customers];*)  ` Add a record to the [Customers] table
        Until (OK=0)   ` Until the user cancels
```

2. The following example queries the database for a customer. Depending on the results of the search, one of two things may happen. If no customer is found, then the user is allowed to add a new customer with ADD RECORD. If at least one customer is found, the user is presented with the first record found, which can be modified with MODIFY RECORD:

```
        READ WRITE([Customers])
        INPUT FORM([Customers];"Input")   ` Set the input form
        vlCustNum:=Num(Request ("Enter Customer Number:"))   ` Get the customer number
        If (OK=1)
           QUERY ([Customers];[Customers]CustNo=vlCustNum)   ` Look for the customer
           If (Records in selection([Customers])=0)   ` If no customer is found...
⇒             ADD RECORD([Customers])  ` Add a new customer
           Else
              If(Not(Locked([Customers])))
                 MODIFY RECORD([Customers])  ` Modify the record
                 UNLOAD RECORD([Customers])
              Else
                 ALERT("The record is currently being used.")
              End if
           End if
        End if
```

See Also

ACCEPT, CANCEL, CREATE RECORD, MODIFY RECORD, SAVE RECORD.

System Variables or Sets

Accepting the record sets the OK system variable to 1; canceling it sets the OK system variable to 0. The OK system variable is set only after the record is accepted or canceled.

MODIFY RECORD

Data Entry

version 3

MODIFY RECORD ({table}{; }{*})

Parameter	Type		Description
table	Table	→	Table to use for data entry, or Default table, if omitted
*		→	Hide scroll bars

Description

The command MODIFY RECORD lets the user modifies the current record for the table table or for the default table if you omit the table parameter. MODIFY RECORD loads the record, if it is not already loaded for the current process, and displays the current input form. If there is no current record, then MODIFY RECORD does nothing. MODIFY RECORD does not affect the current selection.

The form is displayed in the frontmost window of the process. The window has scroll bars and a size box. Specifying the optional * parameter causes the window to be drawn without scroll bars or a size box.

To use MODIFY RECORD, the current record must have read-write access and should not be locked.
If the form contains buttons for moving within the selection of records, MODIFY RECORD lets the user click the buttons to modify records and move to other records.

The record is saved (accepted) if the user clicks an Accept button or presses the Enter key (numeric key pad), or if the ACCEPT command is executed.

The record is not saved (canceled) if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the CANCEL command is executed. Even when canceled, the record remains in memory and can be saved if SAVE RECORD is executed before the current record pointer is changed.

After a call to MODIFY RECORD, OK is set to 1 if the record is accepted, to 0 if canceled.

Note: Even when canceled, the record remains in memory and can be saved if SAVE RECORD is executed before the current record pointer is changed.

If you are using MODIFY RECORD and the user does not change any of the data in the record, the record is not considered to be modified, and accepting the record does not cause it to be saved again. Actions such as changing variables, checking check boxes, and selecting radio buttons do not qualify as modifications. Only changing data in a field, either through data entry or through a method, causes the record to be saved.

Example

See example for the command ADD RECORD.

See Also

ADD RECORD, Locked, Modified record, READ WRITE, UNLOAD RECORD.

System Variables or Sets

Accepting the record sets the OK system variable to 1; canceling it sets the OK system variable to 0. The OK system variable is set only after the record is accepted or canceled.

ADD SUBRECORD

Data Entry

version 3

ADD SUBRECORD (subtable; form{; *})

Parameter	Type		Description
subtable	Subtable	→	Subtable to use for data entry
form	String	→	Form to use for data entry
*		→	Hide scroll bars

Description

The command ADD SUBRECORD lets the user add a new subrecord to subtable, using the form form. ADD SUBRECORD creates a new subrecord in memory, makes it the current subrecord, and displays form. A current record for the parent table must exist. If a current parent record does not exist for the process, ADD SUBRECORD has no effect. The form must belong to subtable.

The subrecord is kept in memory (accepted) if the user clicks an Accept button or presses the Enter key (numeric pad), or if the ACCEPT command is executed. After the subrecord has been added, the parent record must be explicitly saved in order for the subrecord to be saved.

The subrecord is not saved if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the CANCEL command is executed.

After a call to ADD SUBRECORD, OK is set to 1 if the subrecord is accepted, to 0 if canceled.

The form is displayed in the frontmost window of the process. The window has scroll bars and a size box. Specifying the optional * parameter causes the window to be drawn without scroll bars or a size box.

Example

The following example is part of a method. It adds a subrecord for a new child to an employee's record. The data for the children is stored in a subtable named [Employees]Children. Note that the [Employees] record must be saved in order for the new subrecord to be saved:

```
⇒    ADD SUBRECORD([Employees]Children;"Add Child")
     If (OK=1)   ` If the user accepted the subrecord
        SAVE RECORD ([Employees])   ` save the employee's record
     End if
```

See Also

ACCEPT, CANCEL, MODIFY SUBRECORD, SAVE RECORD.

System Variables or Sets

Accepting the subrecord sets the OK system variable to 1; canceling it sets the OK system variable to 0.

MODIFY SUBRECORD

MODIFY SUBRECORD (subtable; form{; *})

Parameter	Type		Description
subtable	Subtable	→	Subtable to use for data entry
form		→	Form to use for data entry
*		→	Hide scroll bars

Description

The command MODIFY SUBRECORD displays the current subrecord of subtable for modification using the form form. The form must belong to subtable.

A current record for the parent table must exist. If a current parent record does not exist for the process, MODIFY SUBRECORD has no effect. In addition, if there is no current subrecord, then MODIFY SUBRECORD does nothing.

The subrecord is kept in memory (accepted) if the user clicks an Accept button or presses the Enter key (numeric pad), or if the ACCEPT command is executed. After the subrecord has been modified, the parent record must be explicitly saved in order for the subrecord to be saved.

The subrecord is not modified if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the CANCEL command is executed.

After a call to MODIFY SUBRECORD, OK is set to 1 if the subrecord modifications are accepted, to 0 if canceled.

The form is displayed in the frontmost window of the process. The window has scroll bars and a size box. Specifying the optional * parameter causes the window to be drawn without scroll bars or a size box.

See Also

ACCEPT, ADD SUBRECORD, CANCEL, SAVE RECORD.

System Variables or Sets

Accepting the subrecord modifications sets the OK system variable to 1; canceling it sets the OK system variable to 0.

DIALOG

version 3

DIALOG ({table; }form)

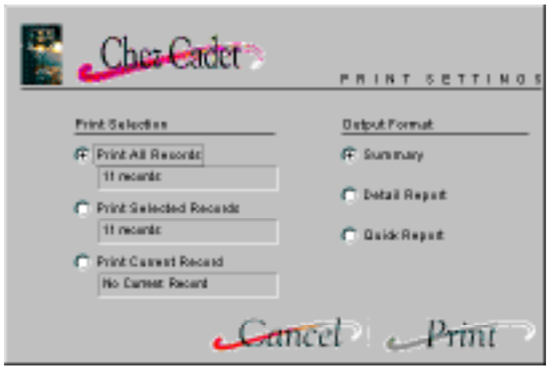
Parameter	Type		Description
table	Table	→	Table owning the form or Default table if omitted
form	Form	→	Form to display as dialog

Description

The command DIALOG presents the form form to the user. This command is often used to get information from the user through the use of variables, or to present information to the user, such as options for performing an operation.

It is common to display the form inside a modal window created with the Open window command.

Here is a typical example of a dialog:



Print Selection

[✓ Print All Records]
11 records

Print Selected Records
11 records

Print Current Record
No Current Record

Output Format

[✓ Summary]

Detail Report

Quick Report

In a dialog, data entry can be performed only by using variables. Fields can be displayed with the current values, but are not enterable.

Tip: Sometimes dialogs can be simulated by ADD RECORD, if you need the capabilities provided by field data entry. In this case, if the form is accepted, a record is added to the table.

Tip: Conversely, data entry can be performed using the DIALOG command. In this case, you must create and save the record. DIALOG does not manipulate records.

Use DIALOG instead of ALERT, CONFIRM, or Request when the information that must be presented or gathered is more complex than those commands can manage.

Unlike ADD RECORD or MODIFY RECORD, DIALOG does not use the current input form. You must specify the form to be used in the form parameter. Also, the default button panel is not used if buttons are omitted. Instead the OK and Cancel buttons are automatically created. Adding any custom button removes the default OK and Cancel buttons.

The dialog is accepted if the user clicks an Accept button or presses the Enter key (numeric key pad), or if the ACCEPT command is executed.

The dialog is canceled if the user clicks a Cancel button or presses the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh), or if the CANCEL command is executed.

After a call to DIALOG, if the dialog is accepted, OK is set to 1; if it is canceled, OK is set to 0.

Example

The following example shows the use of DIALOG to specify search criteria. A custom form containing the variables vName and vState is displayed so the user can enter the search criteria.

```
      Open window (10;40;370;220)  ` Open a modal window
⇒     DIALOG([Company];"Search Dialog")  ` Display a custom search dialog
      CLOSE WINDOW  ` No longer need the modal window
      If (OK=1)  ` If the dialog is accepted
         QUERY ([Company];[Company]Name=vName;*)
         QUERY ([Company];&;[Company]State=vState)
      End if
```

See Also

ACCEPT, ADD RECORD, CANCEL, Open window.

System Variables or Sets

After a call to DIALOG, if the dialog is accepted, OK is set to 1; if it is canceled, OK is set to 0.

Modified

Data Entry

version 3

Modified (field) → Boolean

Parameter	Type		Description
field	Field	→	Field to test
Function result	Boolean	←	True if the field has been assigned a new value, otherwise False

Description

Modified returns True if field has been programmatically assigned a value or has been edited during data entry.

During data entry, a field is considered modified if the user has edited the field (whether or not the original value is changed) and then left it by going to another field or by clicking on a control. Note that just tabbing out of a field does not set Modified to True. The field must have been edited in order for Modified to be True.

When executing a method, a field is considered to be modified if it has been assigned a value (different or not).

In both cases, use the Old command to detect if the field value has been actually changed.

Note: Although modified can be applied to any type of field, if you use it in combination with the old command, be aware of the restrictions that apply to the old command. For details, see the description of the Old command.

During data entry, it is usually easier to perform operations in object methods than to use Modified in form methods. Since an object method is sent an On Data Change event whenever a field is modified, the use of an object method is equivalent to using Modified in a form method.

Examples

1. The following example tests if either the [Orders]Quantity field or the [Orders]Price field has changed. If either has been changed, then the [Orders]Total field is recalculated.

```
⇒   If ((Modified ([Orders]Quantity) | (Modified ([Orders]Price))
        [Orders]Total :=[Orders]Quantity*[Orders]Price
    End if
```

Note that the same thing could be accomplished by using the second line as a subroutine called by the object methods for the [Orders]Quantity field and the [Orders]Price field.

2. You select a record for the table [anyTable], then you call multiple subroutines that may modify the field [anyTable]Important field, but do not save the record. At the end of the main method, you can use the Modified command to detect if you must save the record:

```
		` Here the record has been selected as current record
		` Then you perform actions using subroutines
	DO SOMETHING
	DO SOMETHING ELSE
	DO NOT FORGET TO DO THAT
		` ...
		` At then you test the field to detect if the record has to be saved
⇒	If (Modified([anyTable]Important field))
		SAVE RECORD([anyTable])
	End if
```

See Also

Old.

Old

Old (field) → Expression

Parameter	Type		Description
field	Field	→	Field for which to return old value
Function result	Expression	←	Original field value

Description

The command Old returns the value held in field before the field was programmatically assigned a value or modified in data entry.

Each time you change the current record for a table, 4D creates and maintains in memory a duplicated “image” of the new current record when it is loaded in memory. (For optimization, 4D disregards Text, Picture and BLOB fields.) When modifying a record, you work with the actual image of the record, not this duplicated image. This image is then discarded when you change the current record again.

Old returns the value from the duplicated image. In other words, for an existing record, it returns the value of the field as it is stored on disk. If a record is new, Old returns the default empty value for field according to its type. For example, if field is an Alpha field, Old returns an empty string. If field is a numeric field, Old returns zero (0), and so on.

Old works on field whether the field has been modified by a method or by the user during data entry.

Old cannot be applied to Text, Picture or BLOB fields. It can be applied to all other field types, including subfields, but has no meaning when applied to a subtable field itself.

To restore the original value of a field, assign it the value returned by Old.

See Also

Modified.

12 Date and Time

Current date

version 3

Current date {(*)} → Date

Parameter	Type		Description
*		→	Returns the current date from the server
Function result	Date	←	Current date

Description

The command Current date returns the current date as kept by the system clock.

4D Server: If you use the asterisk (*) parameter when executing this function on a 4D Client machine, it returns the current date from the server.

Examples

1. The following example displays an alert box containing the current date:

```
⇒   ALERT("The date is " + String(Current date)+".")
```

2. If you write an application for the international market, you may need to know if the version of 4D that you run works with dates formatted as MM/DD/YYYY (US version) or DD/MM/YYYY (French version). This is useful to know for customizing data entry fields.

The following project method allows you to do so:

```
        ` Sys date format global function
        ` Sys date format -> String
        ` Sys date format -> Default 4D data format

    C_STRING(31;$0;$vsDate;$vsMDY;$vsMonth;$vsDay;$vsYear)
    C_LONGINT($1;$vlPos)
    C_DATE($vdDate)

        ` Get a Date value where the month, day and year values are all different
⇒   $vdDate:=Current date
    Repeat
        $vsMonth:=String(Month of($vdDate))
        $vsDay:=String(Day of($vdDate))
        $vsYear:=String(Year of($vdDate)%100)
```

```
      If ](($vsMonth=$vsDay) | ($vsMonth=$vsYear) | ($vsDay=$vsYear))
         vOK:=0
         $vdDate:=$vdDate+1
      Else
         vOK:=1
      End if
   Until (vOK=1)
   $0:="" ` Initialize function result
   $vsDate:=String($vdDate)
   $vlPos:=Position("/";$vsDate) ` Find the first / separator in the string ../../..
   $vsMDY:=Substring($vsDate;1;$vlPos-1) ` Extract the first digits from the date
      ` Eliminate the first digits as well as the first / separator
   $vsDate:=Substring($vsDate;$vlPos+1)
   Case of
      : ($vsMDY=$vsMonth) ` The digits express the month
         $0:="MM"
      : ($vsMDY=$vsDay) ` The digits express the day
         $0:="DD"
      : ($vsMDY=$vsYear) ` The digits express the year
         $0:="YYYY"
   End case
   $0:=$0+"/" ` Start building the function result
   $vlPos:=Position("/";$vsDate) ` Find the second separator in the string ../..
   $vsMDY:=Substring($vsDate;1;$vlPos-1) ` Extract the next digits from the date
      ` Reduce the string to the last digits from the date
   $vsDate:=Substring($vsDate;$vlPos+1)
   Case of
      : ($vsMDY=$vsMonth) ` The digits express the month
         $0:=$0+"MM"
      : ($vsMDY=$vsDay) ` The digits express the day
         $0:=$0+"DD"
      : ($vsMDY=$vsYear) ` The digits express the year
         $0:=$0+"YYYY"
   End case
   $0:=$0+"/" ` Pursue building the function result
   Case of
      : ($vsDate=$vsMonth) ` The digits express the month
         $0:=$0+"MM"
      : ($vsDate=$vsDay) ` The digits express the day
         $0:=$0+"DD"
      : ($vsDate=$vsYear) ` The digits express the year
         $0:=$0+"YYYY"
   End case
      ` At this point $0 is equal to MM/DD/YYYY or DD/MM/YYYY or...
```

See Also

Date Operators, Day of, Month of, Year of.

Day of

version 3

Day of (date) → Number

Parameter	Type		Description
date	Date	→	Date for which to return the day
Function result	Number	←	Day of the month of date

Description

The command Day of returns the day of the month of date.

Note: Day of returns a value between 1 and 31. To get the day of the week for a date, use the command Day number.

Examples

1. The following example illustrates the use of Day of. The results are assigned to the variable vResult. The comments describe what is put in vResult:

```
⇒	vResult := Day of (!12/25/92!)  ` vResult gets 25
⇒	vResult := Day of (Current date)  ` vResult gets day of current date
```

2. See the example for the command Current date.

See Also

Day number, Month of, Year of.

Month of

Month of (date) → Number

Parameter	Type		Description
date	Date	→	Date for which to return the month
Function result	Number	←	Number indicating the month of date

Description

The command Month of returns the month of date.

Note: Month of returns the number of the month, not the name (see Example 1).

4th Dimension provides the following predefined constants:

Constants	Type	Value
January	Long Integer	1
February	Long Integer	2
March	Long Integer	3
April	Long Integer	4
May	Long Integer	5
June	Long Integer	6
July	Long Integer	7
August	Long Integer	8
September	Long Integer	9
October	Long Integer	10
November	Long Integer	11
December	Long Integer	12

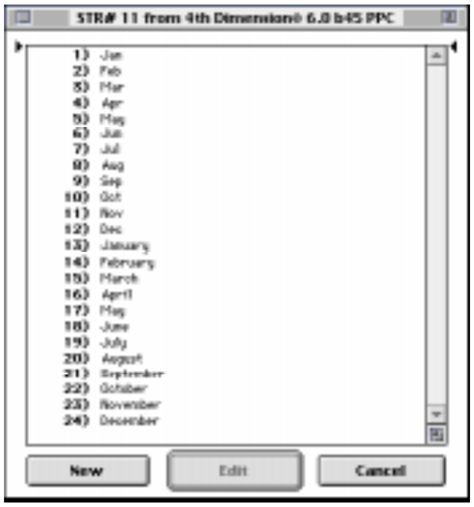
Examples

1. The following example illustrates the use of Month of. The results are assigned to the variable vResult. The comments describe what is put in vResult:

```
⇒      vResult := Month of (!12/25/92!)  ` vResult gets 12
⇒      vResult := Month of (Current date)  ` vResult gets month of current date
```

2. See example for the command Current date.

3. 4th Dimension's 'STR#' ID=11 resource includes the names of the months localized for the current country:



STR# 11 from 4th Dimension® 6.0 b45 PPC

```
 1) Jan
 2) Feb
 3) Mar
 4) Apr
 5) May
 6) Jun
 7) Jul
 8) Aug
 9) Sep
10) Oct
11) Nov
12) Dec
13) January
14) February
15) March
16) April
17) May
18) June
19) July
20) August
21) September
22) October
23) November
24) December
```

The following project method returns the name of the month for a date:

```
	` Month name of project method
	` Month name of ( Date ) -> String
	` Month name of ( Date ) -> Name of the month

⇒	$0:=Get indexed string(11;12+Month of ($1))
```

The following project method returns the abbreviation of the month for a date:

```
	` Month abbr of project method
	` Month abbr of ( Date ) -> String
	` Month abbr of ( Date ) -> Name of the month

⇒	$0:=Get indexed string(11;Month of ($1))
```

See Also

Day of, Year of.

Year of

version 3

Year of (date) → Number

Parameter	Type		Description
date	Date	→	Date for which to return the year
Function result	Number	←	Number indicating the year of date

Description

The command Year of returns the year of date.

Examples

1. The following example illustrates the use of Year of. The results are assigned to the variable vResult.

```
⇒	vResult := Year of (!12/25/92!)  ` vResult gets 1992
⇒	vResult := Year of (!12/25/1992!)  ` vResult gets 1992
⇒	vResult := Year of (!12/25/1892!)  ` vResult gets 1892
⇒	vResult := Year of (!12/25/2092!)  ` vResult gets 2092
⇒	vResult := Year of (Current date)  ` vResult gets year of current date
```

2. See example for the command Current date.

See Also

Day of, Month of.

Day number

Day number (date) → Number

Parameter	Type		Description
date	Date	→	Date for which to return the number
Function result	Number	←	Number representing the weekday on which date falls

Description

The command Day number returns a number representing the weekday on which date falls.

Note: Day number returns 2 for null dates.

4th Dimension provides the following predefined constants:

Constants	Type	Value
Monday	Long Integer	2
Tuesday	Long Integer	3
Wednesday	Long Integer	4
Thursday	Long Integer	5
Friday	Long Integer	6
Saturday	Long Integer	7
Sunday	Long Integer	1

Note: Day number of returns a value between 1 and 7. To get the day number within the month for a date, use the command Day of.

Example

The following example is a function that returns the current day as a string:

```
⇒   $viDay := Day number (Current date)  ` $viDay gets the current day number
    Case of
        : ($viDay = 1)
        $0 := "Sunday"
        : ($viDay = 2)
        $0 := "Monday"
        : ($viDay = 3)
        $0 := "Tuesday"
        : ($viDay = 4)
        $0 := "Wednesday"
        : ($viDay = 5)
        $0 := "Thursday"
        : ($viDay = 6)
        $0 := "Friday"
        : ($viDay = 7)
        $0 := "Saturday"
    End case
```

See Also

Day of.

Add to date

Add to date (date; years; months; days) → Date

Parameter	Type		Description
date	Date	→	Date to which to add days, months, and years
years	Number	→	Number of years to add to the date
months	Number	→	Number of months to add to the date
days	Number	→	Number of days to add to the date
Function result	Date	←	Resulting date

Description

The command Add to date adds years, months, and days to the date you pass in date, then returns the result.

Although you can use the Date Operators to add days to a date, Add to date allows you to quickly add months and years without having to deal with the number of days per month or leap years (as you would when using the + date operator).

Examples

```
	` This line calculates the date in one year, same day
$vdInOneYear:=Add to date(Current date;1;0;0)

	` This line calculates the date next month, same day
$vdNextMonth:=Add to date(Current date;0;1;0)

	` This line does the same thing as $vdTomorrow:=Current date+1
$vdTomorrow:=Add to date(Current date;0;1;0)
```

See Also

Date Operators.

Date

version 3

Date (dateString) → Date

Parameter	Type		Description
dateString	String	→	String representing the date to be returned
Function result	Date	←	Date

Description

The command Date evaluates dateString and returns a date.

The dateString parameter must follow the normal rules for the date format.

In the US version of 4D, the date must be in the order MM/DD/YY (month, day, year). The month and day can be one or two digits. The year can be two or four digits. If the year is two digits, then Date adds 19 to the beginning of the year, unless you have change this default using the command SET DEFAULT CENTURY. The following characters are valid date separators: slash (/), space, period (.), and hyphen (-).

Date does not check whether or not dateString is a valid date. If an invalid date (such as "13/35/94") is passed, Date will return the invalid date. However, if dateString could not possibly be interpreted as a date (for example, "aa/12/94"), the null date value (!00/00/00!) is returned.

It is your responsibility to verify that dateString is a valid date.

Examples

1. The following example uses a request box to prompt the user for a date. The string entered by the user is converted to a date and stored in the reqDate variable:

```
⇒	vdRequestedDate:=Date(Request ("Please enter the date:";String(Current date)))
	If (OK=1)
	   ` Do something with the date now stored in vdRequestedDate
	End if
```

2. The following example returns the string "12/12/94" as a date:

```
⇒	vdDate:=Date("12/12/94")
```

Current time

Current time {(*)} → Time

Parameter	Type		Description
*		→	Returns the current time from the server
Function result	Time	←	Current time

Description

The command Current time returns the current time from the system clock.

The current time is always between 00:00:00 and 23:59:59. Use String or Time string to obtain the string form of the time expression returned by Current time.

4D Server: If you use the asterisk (*) parameter when executing this function on a 4D Client machine, it returns the current time from the server.

Examples

1. The following example shows you how to time the length of an operation. Here, LongOperation is a method that needs to be timed:

```
⇒	$vhStartTime:=Current time  ` Save the start time
	LongOperation  ` Perform the operation
⇒	ALERT ("The operation took "+String(Current time–$vhStartTime))  ` Display how long it took
```

2. The following example extracts the hours, minutes, and seconds from the current time:

```
⇒	$vhNow:=Current time
	ALERT("Current hour is: "+String($vhNow\3600))
	ALERT("Current minute is: "+String(($vhNow\60)%60))
	ALERT("Current second is: "+String($vhNow%60))
```

See Also

Milliseconds, String, Tickcount, Time Operators.

Time string

Time string (seconds) → String

Parameter	Type		Description
seconds	Number	→	Seconds from midnight
Function result	String	←	Time as a string in 24-hour format

Description

The command Time string returns the string form of the time expression you pass in seconds.

The string is in the HH:MM:SS format.

If you go beyond the number of seconds in a day (86,400), Time string continues to add hours, minutes, and seconds. For example, Time string (86401) returns 24:00:01.

Note: If you need the string form of a time expression in a variety of formats, use String.

Example

The following example displays an alert box with the message, “46800 seconds is 13:00:00.”

⇒ **ALERT**("46800 seconds is "+**Time string**(46800))

See Also

String, Time.

Time

Time (timeString) → Time

Parameter	Type		Description
timeString	Time	→	Time for which to return number of seconds
Function result	Time	←	Time specified by timeString

Description

The command Time returns a time expression equivalent to the time specified as a string by timeString.

The timeString parameter must follow the HH:MM:SS format and be in 24-hour format.

Example

The following example displays an alert box with the message “1:00 P.M. = 13 hours 0 minute”:

```
⇒	ALERT ("1:00 P.M. = "+String(Time("13:00:00");Hour Min))
```

See Also

String, Time string.

Tickcount

Tickcount → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of ticks (60th of a second) elapsed since the machine was started

Description

Tickcount returns the number of ticks (60th of a second) elapsed since the machine was started.

Note: Tickcount returns a value of type Long Integer.

Example

See example for the command Milliseconds.

See Also

Current time, Milliseconds.

Milliseconds

version 6.0

Milliseconds → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of milliseconds elasped since the machine was started

Description

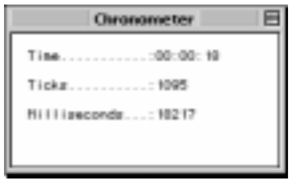
Milliseconds returns the number of milliseconds (1000th of a second) elapsed since the machine was started.

Note: Milliseconds returns a value of type Real.

Example

The following code displays the “Chronometer” window for one minute::

```
       Open window (100;100;300;200;0;"Chronometer")
       $vhTimeStart:=Current time
       $vlTicksStart:=Tickcount
⇒      $vrMillisecondsStart:=Milliseconds
       Repeat
          GOTO XY (2;1)
          MESSAGE ("Time...........:"+String (Current time -$vhTimeStart))
          GOTO XY (2;3)
          MESSAGE ("Ticks..........:"+String (Tickcount -$vlTicksStart))
          GOTO XY (2;5)
⇒         MESSAGE ("Milliseconds...:"+String (Milliseconds -$vrMillisecondsStart))
       Until ((Current time -$vhTimeStart)>=†00:01:00†)
       CLOSE WINDOW
```



```
Time............:00:00:10
Ticks...........:1095
Milliseconds....:10217
```

See Also

Current time, Tickcount.

SET DEFAULT CENTURY

version 6.0

SET DEFAULT CENTURY (century{; pivotYear})

Parameter	Type		Description
century	Number	→	Default century (minus one) for entry of date with two-digit year
pivotYear	Number	→	Pivot year for entry of date with two-digit year

Description

The command SET DEFAULT CENTURY allows you to specify the default century used by 4D when you enter a date with only two digits for the year.

By default, 4D sets the century to be the 20th century. For example:

• 01/25/97 means January 25, 1997
• 01/25/07 means January 25, 1907

To change this default, pass the new default century minus one in century.
For example, after the call:

```
SET DEFAULT CENTURY(20) ` Switch to 21st century for default century
```

• 01/25/97 means January 25, 2097
• 01/25/07 means January 25, 2007

In addition, if you specify the optional pivotYear parameter, 4D will interpret data entry of a date with a two-digit year as follows:

• If the year is greater than or equal to the pivot year, 4D uses the current default century.
• If the year is less than the pivot year, 4D uses the next century (relative to the current default).

For example, after this call, in which the pivot year is 1995:

```
SET DEFAULT CENTURY(19;95) ` Switch to 21st century for default century if year
is less than
```

• 01/25/97 means January 25, 1997
• 01/25/07 means January 25, 2007

Note: This command only affects how 4D interprets dates entered with a two-digit year.

In all cases:

• 01/25/1997 means January 25, 1997
• 01/25/2097 means January 25, 2097
• 01/25/1907 means January 25, 1907
• 01/25/2007 means January 25, 2007

This command only affects data entry. It has no effect on date storage, computation, and so on.

The effect of SET DEFAULT CENTURY is immediate.

13 Debugging

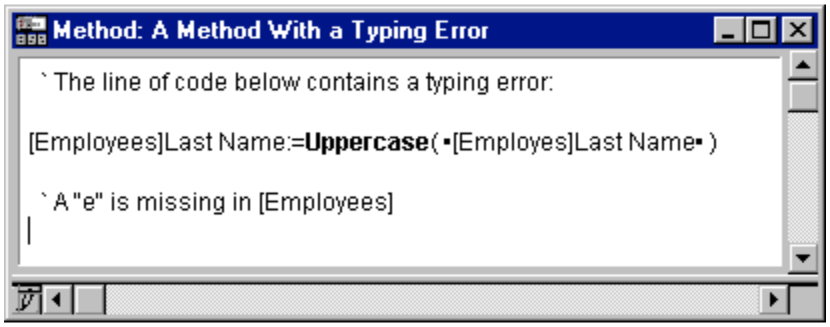
Why a Debugger?

When developing and testing your methods, it is important that you find and fix the errors they may contain.

There are several types of errors you can make when using the language: typing errors, syntax errors, environmental errors, design or logic errors, and runtime errors.

Typing Errors

Typing errors are detected by the **Method editor** and are marked with bullets (•). The following window shows a typing error:



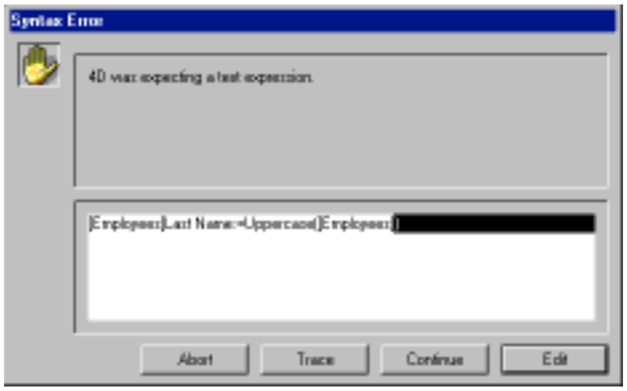
```
 ` The line of code below contains a typing error:

[Employees]Last Name:=Uppercase(•[Employes]Last Name•)

 ` A "e" is missing in [Employees]
```

Note: The comments have been manually inserted for the purpose of this manual. 4D only inserts the (•) at the location of the error.

When this occurs, fix the typing error and type Enter (on the numeric pad) to validate the fix. For more information about the Method editor, refer to the *4th Dimension Design Reference*.

Syntax Error

Syntax errors are caught when you execute the method. The Syntax Error window is displayed when a syntax error occurs. For example:



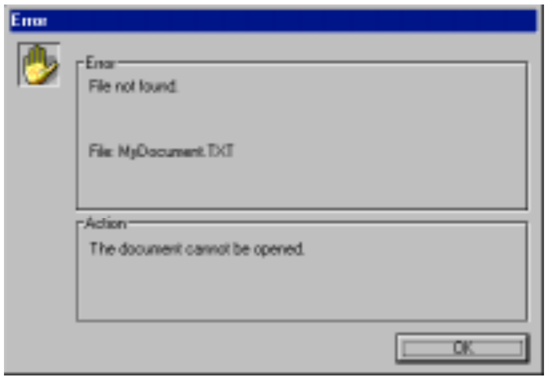
4D was expecting a text expression.

[Employees]Last Name:=Uppercase([Employees])

In this window, the error is that a table name is passed to the Uppercase command, which expects a text expression. To learn about this window and its button, see the section Syntax Error window.

Environmental Error

Occasionally, there there may not be enough memory to create an array or a BLOB. When you access a document on disk, the document may not exist or may already open by another application. In such cases, the Error window appears, describing the error and the action that could not be performed. For example:



File not found.

File: MyDocument.TXT

The document cannot be opened.

These errors do not directly occur because of your code or the way you wrote it; they occur because sometimes “bad things just happen.” Most of the time, these errors are easy to treat with an error catching method installed using the command ON ERR CALL. For more information, see the description of ON ERR CALL.

Design or Logic Error

These are generally the most difficult type of error to find—use the Debugger to detect them. Note that, other than typing errors, all the previous error types are to a certain extent covered by the expression “Design or logic error.” For example:

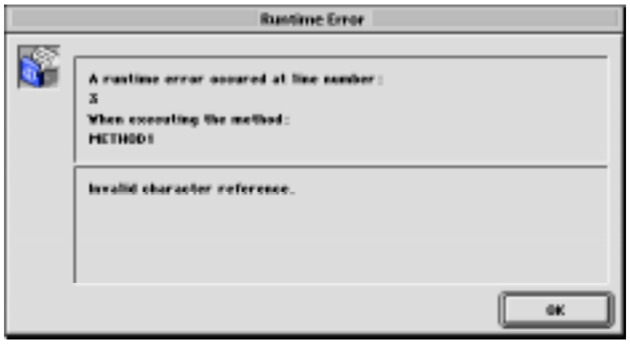
• A syntax error may occur because you try to use a variable that has not yet been initialized.
• An environmental error may occur because you try to open a document whose name is received by a subroutine which does not get the right value in the parameter. Note that in this example, the piece of code that actually “breaks” may be different than the code that is actually the origin of the problem.

Design or logic errors also include such situations as:

• A record is not properly updated because, while calling SAVE RECORD, you forgot to first test whether or not the record was locked.
• A method does not do exactly what you expect, because the presence of an optional parameter is not tested.

Runtime Error

In compiled mode, you can obtain errors that you never saw in interpreted mode. Here is an example:



```
A runtime error occured at line number :
3
When executing the method :
METHOD1

Invalid character reference.
```

This says “You are trying to access a character whose position is beyond the length of a string.” To quickly find the origin of the problem, note the name of the method and the line number, reopen the interpreted version of the structure file, and go to that method at the indicated line.

What To Do When an Error Occurs?

Errors are common. It would be unusual to write a substantial number of lines of code (let’s say several hundred) without generating any errors. Conversely, treating and/or fixing errors is normal, too!

With its multi-tasking environment, 4D enables you to quickly edit/run methods by simply switching windows. You will discover how quickly you can fix mistakes and errors when you do not have to rerun the whole thing each time. You will also discover how quickly you can track errors if you use the Debugger.

A common beginner mistake in dealing with error detection is to click Abort in the Syntax Error Window, go back to the Method Editor, and try to figure out what's going by looking at the code. Do not do that! You will save plenty of time and energy by **always** using the Debugger.

• If an unexpecting syntax error occurs, use the Debugger.
• If an environmental error occurs, use the Debugger.
• If any other type of error occurs, use the Debugger.

In 99% of the cases, the Debugger displays the information you need in order to understand why an error occurred. Once you have this information, you know how to fix the error.

Tip: A few hours spent in learning and experimenting with the Debugger can save days and weeks in the future when you have to track down errors.

Another reason to use the Debugger is for developing code. Sometimes you may write an algorithm that is more complex than usual. Despite all feelings of accomplishment, you are not totally sure that your coding is correct, even before trying it. Instead of running it “blind,” use the TRACE command at the beginning of your code. Then, execute it step by step to control what happens and to check whether your suspicion was correct or not. A purist may dislike this method, but somethimes pragmatism pays off more quickly. Anyway... use the Debugger.

General Conclusion
Use the Debugger.

See Also

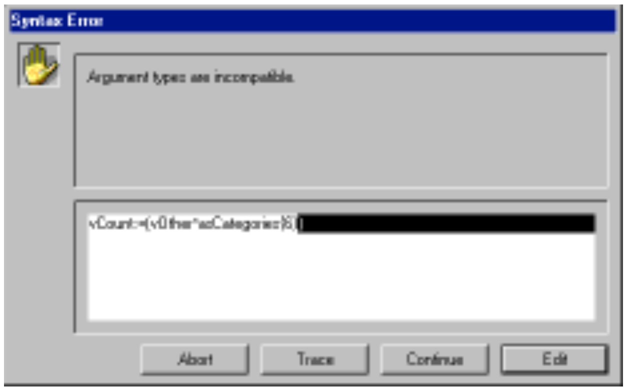
Break List Window, Debugger, Debugger Shortcuts, ON ERR CALL, Syntax Error Window, Tracing a Process not visible or not executing code.

Syntax Error Window

The Syntax Error Window is displayed when method execution is halted. Method execution can be halted for either of two reasons:

• 4th Dimension halts execution because there is a syntax error preventing further method execution.
• You generate a user interrupt by pressing Alt+Click (Windows) or Option+Click (Macintosh) while a method is executing.

The Syntax Error window is shown here:



Argument types are incompatible.

```
vCount:=(vOther*asCategories{6})
```

The upper text area of the Syntax Error window displays a message describing the error. The lower text area shows the line that was executing when the error occurred; the area where the error occurred is highlighted.

There are four option buttons at the bottom of the window: **Abort**, **Trace**, **Continue**, and **Edit**.

• **Abort**: The method is halted, and you return to where you were before you started executing the method. If a form or object method is executing in response to an event, it is stopped and you return to the form. If the method is executing from within the Custom Menu environment, you return to the Custom Menu environment.

• **Trace**: You enter Trace/Debugger mode, and the Debugger window is displayed. If the current line has been partially executed, you may have to click the Trace button several times. Once the line finishes, you end up in the Debugger window.

• **Continue**: Execution continues. The line with the error may be partially executed, depending on where the error was. Continue with caution—the error may prevent the remainder of your method from executing properly. Usually, you do not want to continue. You can click Continue if the error is in a trivial call, such as SET WINDOW TITLE, which does not prevent executing and testing the rest of your code. You can thus concentrate on more important code, and fix a minor error later.

• **Edit**: All method execution is halted. 4th Dimension switches to the Design environment. The method in which the error occurred is opened in the Method editor, allowing you to correct the error. Use this option when you immediately recognize the mistake and can fix it without further investigation.

See Also

Debugger, ON ERR CALL, Why a Debugger?.

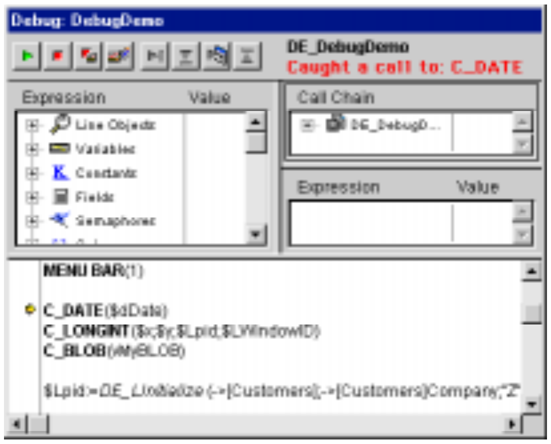
The term Debugger comes from the term bug. A bug in a method is a mistake that you want to eliminate. When an error has occurred, or when you need to monitor the execution of your methods, you use the debugger. A debugger helps you find bugs by allowing you to slowly step through your methods and examine method information. This process of stepping through methods is called **tracing**.

You can cause the Debugger window to display and then trace the methods in the following ways:

• Clicking the **Trace** button in the Syntax Error Window
• Using the TRACE command
• Pressing Alt+Click (Windows) or Option-Click (Macintosh) while a method is executing
• Choosing **Trace** from the **Process** menu in the Design environment for the process selected in the **Process List Window** (see section Tracing a Process not visible or not executing code)
• Creating or editing a Catch Command or Break Point in the Break List Window.

Note: If a password system exists for the database, only the designer and users belonging to the group that has structure access privileges can trace methods.

The Debugger window is displayed here:



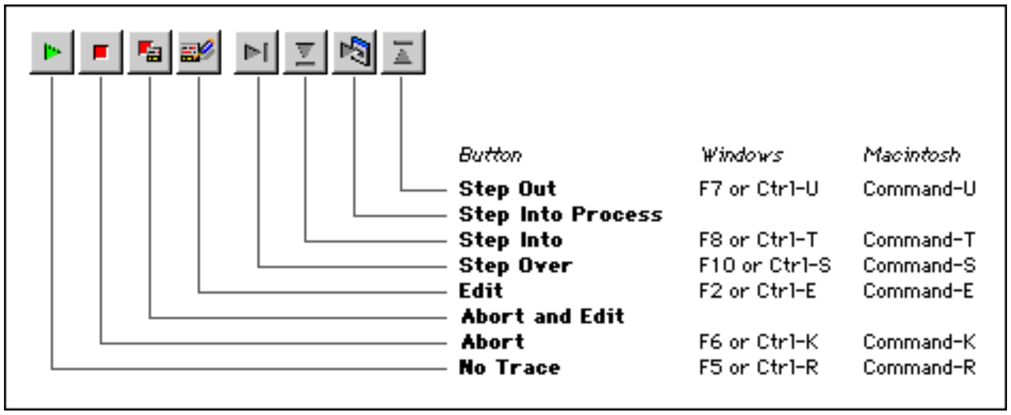
```
MENU BAR(1)

C_DATE($dDate)
C_LONGINT($x;$y;$Lpid;$LWindowID)
C_BLOB(vMyBLOB)

$Lpid:=DE_LInitialize(->[Customers];->[Customers]Company;"Z"
```

You can move the Debugger Window and/or resize any of its internal window panes as necessary.

4D is a multi-tasking environment. If you run several user processes, you can trace them independently. You can have one debugger window open for each process.

Execution Control Tool Bar Buttons

Eight buttons are located in the **Execution Control Tool Bar** at the top of the Debugger window:



Button	*Windows*	*Macintosh*
Step Out	F7 or Ctrl-U	Command-U
Step Into Process		
Step Into	F8 or Ctrl-T	Command-T
Step Over	F10 or Ctrl-S	Command-S
Edit	F2 or Ctrl-E	Command-E
Abort and Edit		
Abort	F6 or Ctrl-K	Command-K
No Trace	F5 or Ctrl-R	Command-R

No Trace Button

Tracing is halted and normal method execution resumes.

Note: ALT+F5 (Windows) and Option-Command-R (Macintosh) resumes execution. They also disable all the next TRACE calls for the current process.

Abort Button

The method is halted, and you return to where you were before you started executing the method. If you were tracing a form or object method executing in response to an event, it is stopped and you return to the form. If you were tracing a method executing from within the Custom Menu environment, you return to the Custom Menu environment.

Abort and Edit Button

The method is halted as if you clicked on Abort. Also, if necessary, 4th Dimension opens and brings the Design environment process to the front, then opens a Method Editor window for the method that was executing at the time the **Abort and Edit** button was clicked.

Tip: Use this button when you know which changes are required in your code and when these changes are required to pursue the testing of your methods. After you are finished with the changes, rerun the method.

Edit Button

Clicking the Edit button does the same as Clicking **Abort and Edit** button, but does not abort the current execution. The method execution is paused at that point. If necessary, 4th Dimension opens and brings the Design environment process to the front, then opens a Method Editor window for the method that was executing at the time the **Edit** button was clicked.

Important: You can modify this method; however, these modifications will not appear or execute in the instance of the method currently being traced in the debugger window. After the method has either aborted or completed successfully, the modifications will appear on the next execution of this method. In other words, the method must be reloaded so its modifications will be taken into account.

Tip: Use this button when you know which changes are required in your code and when they do not interfere with the rest of the code to be executed or traced.

Tip: Object Methods are reloaded for each event. If you are tracing an object method (i.e., in response to a button click), you do not need to leave the form. You can edit the object method, save the changes, then switch back to the form and retry. For tracing/changing form methods, you must exit the form and reopen it in order to reload the form method. When doing extensive debugging of a form, a trick is to put the code (that you are debugging) into a project method that you use as subroutine from within a form method. In doing so, you can stay in the form while you trace, edit, and retest your form, because the subroutine is reloaded each time it is called by the form method.

Step Over Button

The current method line (the one indicated by the yellow arrow—called the **program counter**) is executed, and the Debugger steps to the next line. The **Step Over** button does not step into subroutines and functions; it stays at the level of the method you are currently tracing. If you want to also trace subroutines and functions calls, use the **Step Into** button.

Step Into Button

On execution of a line that calls another method (subroutine or function), this button causes the Debugger window to display the method being called and allows you to step through this method. The new method becomes the current (top) method in the Call Chain pane of the Debugger window. On execution of a line that does not call another method, this button acts in the same manner as the **Step Over** button.

Step Into Process Button

On execution of a line that creates a new process (i.e., calling the command New process), this button opens a new Debugger window that allows you to trace the process method of the newly created process. On execution of a line that does not creates a new process, this button acts in the same manner as the **Step Over** button.

Step Out Button

If you are tracing subroutines and functions, clicking on this button allows you to execute the entire method currently being traced and to step back to the caller method. The Debugger window is brought back to the previous method in the call chain. If the current method is the last method in the call chain, the Debugger window is closed.

Execution Control Tool Bar Information

On the right side of the execution control tool bar, the debugger provides the following information:
• The name of the method you are currently tracing (displayed in black)
• The problem caused the appearance of the Debugger window (displayed in red)

Using the example window shown above, the following information is displayed:
• The method DE_DebugDemo is the method being traced.
• The debugger window appeared because it detected a call to the command C_DATE and this command was one of the commands to be caught.

Here are the possible reasons for the debugger to appear and for the message (displayed in red):
• **TRACE Command**: A call to TRACE has been issued.
• **Break Point Reached**: A temporary or persistent break point has been encountered.
• **User Interrupt**: You used ALT+Click (Windows) or Option-Click (Macintosh) or you used the **Trace** menu command from the Design environment **Process** menu.
• **Caught a call to: Name of the command**: A call to a 4D command to be caught is on the point of being performed.
• **Stepping into a new process**: You used the **Step Into Process** button and this message is displayed by the Debugger window opened for the newly created process.

The Debugger Window's Panes

The Debugger window consists of the previously described Execution Control Tool Bar and four resizable panes:

• Watch Pane
• Call Chain Pane
• Custom Watch Pane
• Source Code Pane

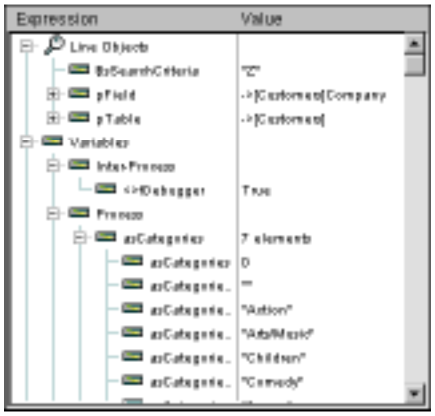
The first three panes use easy-to-navigate hierarchical lists to display pertinent debugging information. The fourth one, Source Code Pane, displays the source code of the method being traced. Each pane has its own function to assist you in your debugging efforts. You can use the mouse to vertically and horizontally resize the debugger window and also each pane. In addition, the first three panes include a dotted separation line between the two columns they display. Using the mouse, you can move this dotted line to horizontally resize the columns, at your convenience.

See Also

Break List Window, Call Chain Pane, Custom Watch Pane, Debugger Shortcuts, ON ERR CALL, Source Code Pane, Syntax Error Window, TRACE, Watch Pane, Why a Debugger?.

Watch Pane

The **Watch pane** is displayed in the top left corner of the Debugger window, below the Execution Control Tool Bar. Here is an example:



Expression	Value
Line Objects	
$sSearchCriteria	"Z"
pField	->[Customers]Company
pTable	->[Customers]
Variables	
Inter-Process	
<>fDebugger	True
Process	
asCategories	7 elements
asCategories	0
asCategorie...	""
asCategorie...	"Action"
asCategorie...	"Arts/Music"
asCategorie...	"Children"
asCategorie...	"Comedy"

The Watch pane displays useful general information about the system, the 4D environment, and the execution environment.

The **Expression** column displays the names of the objects or expressions. The **Value** column displays the current value of corresponding the object or expression.

Clicking on any value on the right side of the pane allows you to modify the value of the object, if this is permitted for that object.

The multi-level hierarchical lists are organized by theme at the main level. The themes are:

- Line Objects
- Variables
- Constants
- Fields
- Semaphores
- Sets
- Processes
- Named Selections
- Information

Depending on the theme, each item may have one or several sublevels. Clicking the list node next to a theme name expands or collapses the theme. If the theme is expanded, the items in that theme are visible. If the theme has several levels of information, click the list node next to each item for exploring all the information provided by the theme.

At any point, you can drag and drop themes, theme sublists (if any), and theme items to the Custom Watch pane.

Information: Displays general information, such the current Default Table (if any). The expressions from this theme cannot be modified.

Named Selections: Lists the process named selections that are defined in the current process (the one you're currently tracing); it also lists the interprocess named selections. For each named selection, the Value column displays the number of records and the table name. This list may be empty if you do not use named selections. The expressions from this theme cannot be modified.

Processes: Lists the processes started since the beginning of the working session. The value column displays the current state for each process (i.e., Executing, Paused, and so on). The expressions from this theme cannot be modified.

Sets: Lists the sets defined in the current process (the one you're currently tracing); it also lists the interprocess sets. For each set, the Value column displays the number of records and the table name. This list may be empty if you do not use sets. The expressions from this theme cannot be modified.

Semaphores: Lists the local and global semaphores currently being set. For each semaphore, the Value column provides the name of the process that sets the semaphore. This list may be empty if you do not use semaphores. The expressions from this theme cannot be modified.

Fields: This theme lists the tables and fields in the database; it does not list subfields. For each Table item, the Value column displays the size of the current selection for the current process. For each Field item, the Value column displays the value of the field (except picture, subtable, and BLOB) for the current record, if any. In this theme, the field values can be modified (there is no undo), but the table information cannot.

Constants: Displays predefined constants provided by 4D. like the Constants page of the Explorer window. The expressions from this theme cannot be modified.

Variables: This theme is composed of the following subthemes:
• **Interprocess**: Displays the list of the interprocess variables being used at this moment. This list can be empty if you do not use interprocess variables. The values of the interprocess variables can be modified.
• **Process**: Displays the list of the process variables being used by the current process. This list is rarely empty. The values of the process variables can be modified.
• **Local**: Displays the list of the local variables being used by the method being traced (the one being shown in the source code pane). This list can be empty if no local variable is used or has not yet been created. The values of the local variables can be modified.

• **Parameters**: Displays the list of parameters received by the method. This list can be empty if no parameter were passed to the method being traced (the one being shown in the source code pane). The values of the parameters can be modified.
• **Self Pointer**: Displays a pointer to the current object if you are tracing an Object Method. This value cannot be modified

Note: You can modifiy String, Text, Numeric, Date, and Time variables; in other words, you can modify the variables whose value can be entered with the keyboard.

Arrays, like other variables, appear in the Interprocess, Process, and Locals subthemes, depending on their scope. The debugger displays each array with an additional hierarchical level; this enables you to obtain or change the values of the array elements, if any. The debugger displays the first 100 elements, including the element zero. The Value column displays the size of the array in regard to its name. After you have deployed the array, the first sub-item displays the current selected element number, then the element zero, then the other elements (up to 100). You can modifiy String, Text, Numeric, and Date arrays. You can modify the selected element number, the element zero, and the other elements (up to 100). You cannot modify the size of the array.

Reminder: At any time, you can drag and drop an item from the Watch pane to the Custom Watch pane, including an individual array element.

Line Objects

This theme displays the values of the objects or expressions that are:
• used in the line of code to be executed (the one marked with the program counter—the yellow arrow in the Source Code pane), or
• used in the previous line of code.

Since the previous line of code is the one that was just executed before, the Line Objects theme therefore shows the objects or expressions of the current line before and after that the line was executed. Let's say you execute the following method:

```
TRACE
a:=1
b:=a+1
c:=a+b
   ` ...
```

1. You enter the Debugger window with the Source Code pane program counter set to the line a:=1. At this point the Line Objects theme displays:

```
a:        Undefined
```

The a variable is shown because it is used in the line to be executed (but has not yet been initialized).

2. You step one line. The program counter is now set to the line b:=a+1. At this point, the Line Objects theme displays:

a:	1
b:	Undefined

The a variable is shown because it is used in the line that was just executed and was assigned the numeric value 1. It is also shown because it is used in the line to be executed as the expression to be assigned to the variable b. The b variable is shown because it is used in the line to be executed (but has not yet been initialized).

3. Again, you step one line. The program counter is now set to the line c:=a+b. At this point the Line Objects theme displays:

c:	Undefined
a:	1
b:	2

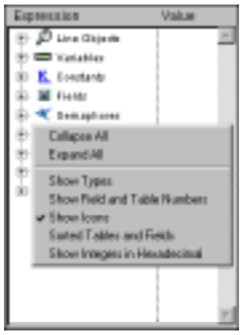
The c variable is shown because it is used in the line to be executed (but has not yet been initialized). The a and b variables are shown because there were used in the previous line and are used in the line to be executed. And so on...

The Line Objects theme is a very convenient tool—each time you execute a line, you do not need to enter an expression in the Custom Watch pane, just watch the values displayed by the Line Objects theme.

Speed Menu

Addtional options are provided by the Speed Menu of the Watch pane. To display this menu:
• On Windows, click anywhere in the Watch pane using the **right** mouse button.
• On Macintosh, Control-Click anywhere in the Watch pane.

The Speed Menu of the Watch pane is shown here:



• **Collapse All**: Collapses all levels of the Watch hierarchical list.

• **Expand All**: Collapses all levels of the Watch hierarchical list.

• **Show Types**: Displays the object type for each object (when appropriate).

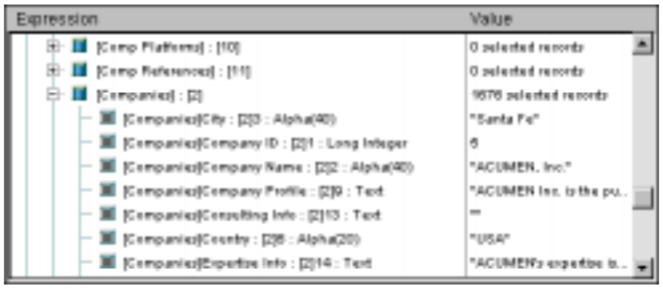
• **Show Field and Table Numbers**: Displays the number of each table or field of the **Fields**. If you work with table or field numbers, or with pointers using the commands such as Table or Field, this option is very useful.

• **Show Icons**: Displays an icon denoting the object type for each object. You can turn this option off in order to speed up the display, or just because you prefer to use only the **Show Types** option.

• **Sorted Tables and Fields**: Forces the table and fields to be displayed in alphabetical order, within their respective lists.

• **Show Integers in Hexadecimal**: Numbers are usually displayed in decimal notation. This option displays them in hexadecimal notation. **Note:** To enter a numeric value in hexadecimal, type 0x (zero + "x"), followed by the hexadecimal digits.

The following is a view of the Watch pane with all options selected:



Expression	Value
[Comp Platforms] : [10]	0 selected records
[Comp References] : [11]	0 selected records
[Companies] : [2]	1676 selected records
[Companies]City : [2]3 : Alpha(40)	"Santa Fe"
[Companies]Company ID : [2]1 : Long Integer	6
[Companies]Company Name : [2]2 : Alpha(40)	"ACUMEN, Inc."
[Companies]Company Profile : [2]9 : Text	"ACUMEN Inc. is the pu...
[Companies]Consulting Info : [2]13 : Text	""
[Companies]Country : [2]8 : Alpha(20)	"USA"
[Companies]Expertise Info : [2]14 : Text	"ACUMEN's expertise is...

See Also

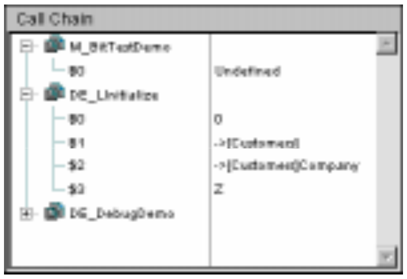
Call Chain Pane, Custom Watch Pane, Debugger, Debugger Shortcuts, Source Code Pane.

Call Chain Pane

Debugging

version 6.0

One method may call other methods, which may call other methods. For this reason, it is very helpful to see the chain of methods, or **Call Chain,** during the debugging process. The Call Chain pane, which provides this useful function, is the top right pane of the Debugger window. This pane is displayed using a hierarchical list. Here is an example of the Call Chain pane:



```
Call Chain
⊟ M_BitTestDemo
  └ $0              Undefined
⊟ DE_LInitialize
  ├ $0              0
  ├ $1              ->[Customers]
  ├ $2              ->[Customers]Company
  └ $3              Z
⊞ DE_DebugDemo
```

• Each main level item is a name of a method. The top item is the method you are currently tracing, the next main level item is the name of the caller method (the method that called the method you are currently tracing), the next one is the caller's caller method, and so on. In the example above, the method M_BitTestDemo is being traced; it has been called by the method DE_LInitialize, which has been called by DE_DebugDemo.

• Double-clicking the name of a method in the Call Chain pane "transports" you back to the caller method, displaying its source code in the Source code pane. In doing so, you can quickly see "how" the caller method made its call to the called method. You can examine any stage of the call chain this way.

• Clicking the node next to a Method name expands or collapses the parameter ($1, $2...) and the optional function result ($0) list for the method. The values appear on the right side of the pane. Clicking on any value on the right side allows you to change the value of any parameter or function result. In the figure above:

1. M_BitTestDemo has not received any parameter.
2. M_BitTestDemo's $0 is currently undefined, as the method did not assign any value to $0 (because it has not executed this assignment yet or because the method is a subroutine and not a function).
3. DE_LInitialize has received three parameters from DE_DebugDemo. $1 is a pointer to the table [Customers], $2 is a pointer to the field [Customers]Company, and $3 is an alphanumeric parameter whose value is "Z".

• After you have deployed the parameter list for a method, you can also drag and drop parameters and function results to the Custom Watch pane.

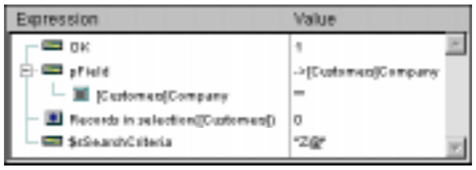
See Also

Custom Watch Pane, Debugger, Debugger Shortcuts, Source Code Pane, Watch Pane.

Directly below the Call Chain pane is the Custom Watch pane. This pane is used to evaluate expressions. Any type of expression can be evaluated, including fields, variables, pointers, calculations, built-in functions, your own functions, and anything else that returns a value.

You can evaluate any expression that can be shown in text form. This does not cover picture and BLOB fields or variables. On the other hand, the Debugger uses deployed hierarchical lists to let you display arrays and pointers. To display BLOB contents, you can use BLOB commands, such as BLOB to text.

In the following example, you can see several of these items: two variables, a field pointer variable and the result of a built-in function, and a calculation.



Expression	Value
OK	1
pField	->[Customers]Company
[Customers]Company	""
Records in selection([Customers])	0
$vSearchCriteria	"Z@"

Inserting a new expression

You can add an expression to be evaluated in the Custom Watch pane in the following way:
• Drag and drop an object or expression from the Watch pane
• Drag and drop an object or expression from the Call Chain pane
• In the Source Code pane, click on an expression that can be evaluated

To create a blank expression, double-click somewhere in the empty space of the Custom Watch pane. This adds an expression ` New expression and then goes into editing mode so you can edit it. You can enter any 4D formula that returns a result.

After you have entered the formula, type **Enter** or **Return** (or click somewhere else in the pane) to evaluate the expression.

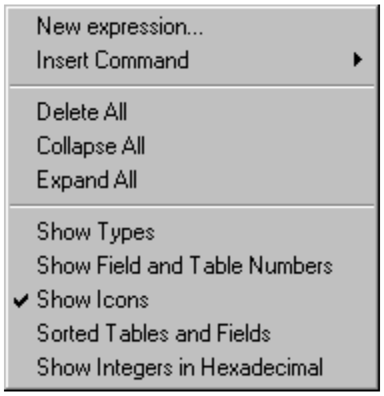
To change the expression, click on it to select it, then click again (or press Enter —numeric key pad) to go into editing mode.

If you no longer need an expression, click on it to select it, then press **Backspace** or **Delete**.

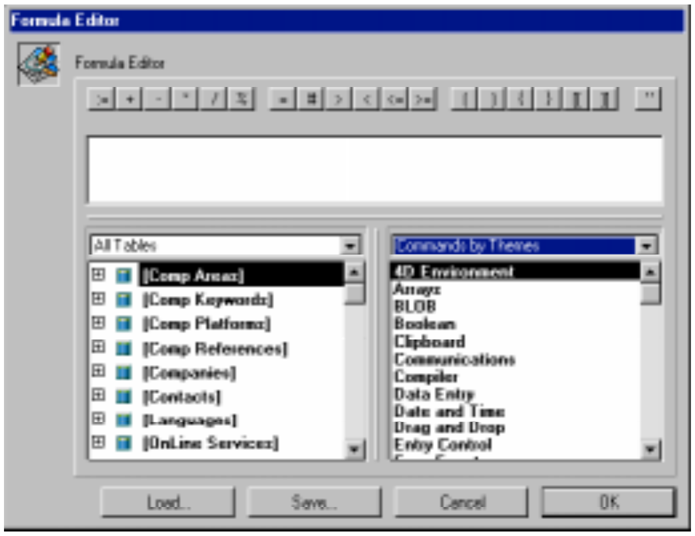
Custom Watch Pane Speed Menu

To help you enter and edit an expression, the Custom Watch Pane's Speed menu gives you access the 4D formula editor. In fact, the speed menu also proposes additional options.

To present this menu:
• On Windows, click anywhere in the Custom Watch pane using the **right** mouse button
• On Macintosh, Control-Click anywhere in the Custom Watch pane.



• **New Expression**: This inserts a new expression and displays the 4D Formula Editor (as shown) so you can edit the new expression.



For more information about the Formula Editor, See the *4th Dimension User Reference Manual*.

• **Insert Command**: This hierarchical menu item is a shortcut for inserting a command as a new expression, without using the Formula Editor.

• **Delete All**: Deletes all the expressions currently present.

• **Collapse All/Expand All**: Collapses or Expands all the expressions whose evaluation is done by the means of a hierarchical list (i.e., pointers, arrays,...)

• **Show Types**: Displays the object type for each object (when appropriate).

• **Show Field and Table Numbers**: Displays the number of each table or field of the **Fields**. If you work with table or field number or pointers using the commands such as Table or Field, this option is very useful.

• **Show Icons**: Displays an icon denoting the object type for each object. You can turn this option off in order to speed up the display, or just because you prefer to use only the **Show Types** option.

• **Sorted Tables and Fields**: Forces the table and fields to be displayed in alphabetical order, within their respective lists.

• **Show Integers in Hexadecimal**: Numbers are displayed using the decimal notation. This option displays them hexadecimal notation. **Note:** To enter a numeric value in hexadecimal, type 0x (zero + "x"), followed by the hexadecimal digits.

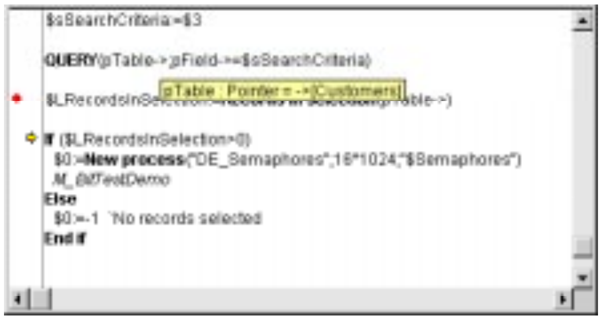
See Also

Call Chain Pane, Debugger, Debugger ShorTcuts, Source Code Pane, Watch Pane.

The **Source Code pane** shows the source code of the method being traced.

• If the method is too long to fit in the text area, you can scroll to view other parts of the method.
• Moving the mouse pointer over any expression that can be evaluated (field, variable, pointer, array,...) will cause a **Tool Tip** to display the current value of the object or expression and its declared type.

Here is an example of the Source Code pane:



```
$sSearchCriteria:=$3

QUERY(pTable->;pField->=$sSearchCriteria)

$LRecordsInSe[pTable : Pointer = ->[Customers]]able->)

If ($LRecordsInSelection>0)
  $0:=New process("DE_Semaphores";16*1024;"$Semaphores")
  M_BitTestDemo
Else
  $0:=-1  `No records selected
End if
```

A tool tip is displayed because the mouse pointer was over the variable pTable which, according to the tool tip, is a pointer to the table [Customers].

Tip: In the Source Code pane, clicking on an expression (that can be evaluated) copies the expression or object to the Custom Watch pane.

Program Counter

A yellow arrow in the left margin of the Source Code pane (see the figure above) marks the next line that will be executed. This arrow is called the **program counter**. The program counter always indicates the line on the verge of being executed.

For debugging purposes, you can **change** the program counter for the method being on top of the call chain (the method actually being executed). To do so, click and drag the yellow arrow vertically, to the line you want.

WARNING: Use this feature with caution!

Moving the program counter forward does NOT mean that the debugger is rapidly executing the lines you skip. Similarily, moving the program counter backward does NOT mean that the debugger is reversing the effect of the lines that has already been executed.

Moving the program counter simply tells the debugger to "pursue tracing or executing from here." All current settings, fields, variables, and so on are not affected by the move.

Here is an example of moving the program counter. Let's say you are debugging the following code:

```
   ` ...
If (This condition)
   DO SOMETHING
Else
   DO SOMETHING ELSE
End if
   ` ...
```

The program counter is set to the line If (This condition). You step once and you see that the program counter moves to the line DO SOMETHING ELSE. This is unfortunate, because you wanted to execute the other alternative of the branch. In this case, and provided that the expression This condition does not perform operations affecting the next steps in your testing, just move the program counter back to the line DO SOMETHING. You can now continuing tracing the part of the code in which you are interested.

Setting Break Points

In the debugging process, you may need to skip the tracing of some parts of the code. The debugger offers you several ways to execute code **up to a certain point**:

• While stepping, you can click on the **Step Over** button instead of **Step Into** button. This is useful when you do not want to enter into possible subroutines or functions called in the program counter line.
• If you mistakenly entered into a subroutine, you can execute it and directly go back to the caller method by clicking on the **Step Out** button.
• If you have a TRACE call placed at some point, you can click the **No Trace** button, which resumes the execution up to that TRACE call.

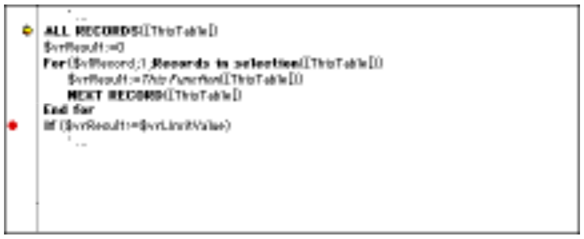
Now, let’s say you are executing the following code, with the program counter set to the line ALL RECORDS([ThisTable]):

```
    ` ...
ALL RECORDS([ThisTable])
$vrResult:=0
For($vlRecord;1;Records in selection([ThisTable]))
    $vrResult:=This Function([ThisTable]))
    NEXT RECORD([ThisTable])
End for
If ($vrResult>=$vrLimitValue)
    ` ...
```

Your goal is to evaluate the value of $vrResult after the For loop has been completed. Since it takes quite some execution time to reach this point in your code, you do not want to abort the current execution, then edit the method in order to insert a TRACE call before the line If ($vrResult....

One solution is to step through the loop, however, if the table [ThisTable] contains several hundreds records, you are going to spend the entire day for this operation. In this type of situation, the debugger offers you **break points**. You can insert break points by clicking in the left margin of the Source Code pane.

For example:
You click in the left margin of the Source Code pane at the level of the line If ($vrResult...:



```
    ` ...
ALL RECORDS([ThisTable])
$vrResult:=0
For($vlRecord;1;Records in selection([ThisTable]))
    $vrResult:=This Function([ThisTable]))
    NEXT RECORD([ThisTable])
End for
If ($vrResult>=$vrLimitValue)
    ` ...
```

This inserts a break point for the line. The break point is indicated by a red bullet. Then click the **No Trace** button.

This resumes the normal execution **up to** the line marked with the break point. That line is not executed itself—you are back to the trace mode. In this example, the whole loop has consequently been executed normally. Then, when reaching the break point, you just need to move the mouse button over $vrResult to evaluate its value at the exit point of the loop.

Setting a break point beyond the program counter and clicking the No Trace button allows you to skip **portions** of the method being traced.

A **red break point** is a **persistent** break point. Once you created it, it “stays.” Even though you quit the database, then reopen it later on, the break point will be there.

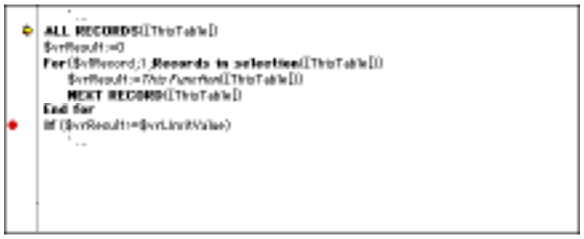
There are two ways to eliminate a persistent break point:
• If you are through with it, just remove it by clicking on the red bullet—the break point disappears.
• If you are not totally through with it, you may want to keep the break point. You can temporarily disable the break point by editing it. This explained in the section Break Points.

See Also

Break Points, Call Chain Pane, Custom Watch Pane, Debugger, Watch Pane.

Break Points

As explained in the Source Code pane section, you set a break point by clicking in the left margin of the Source Code pane at the same level as the line of code on which you want to break. In the following figure, a break point has been set on the line If($vrResult>=$vrLimitValue):



```
    ...
ALL RECORDS([ThisTable])
$vrResult:=0
For($vRecord;1;Records in selection([ThisTable]))
    $vrResult:=ThisFunction([ThisTable])
    NEXT RECORD([ThisTable])
End for
If ($vrResult>=$vrLimitValue)
    ...
```

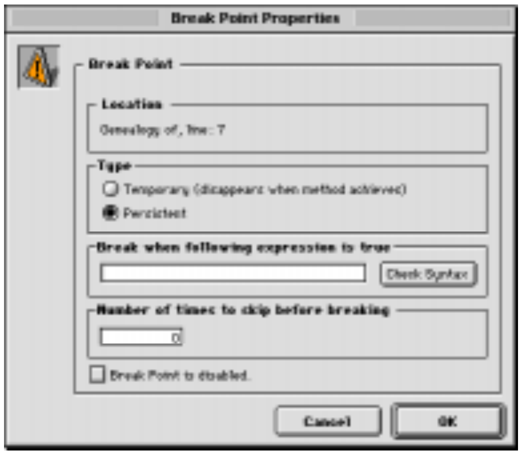
If you click again on the red bullet, the break point is deleted.

Editing a Break Point

Pressing Alt-click (Windows) or Option-click (Macintosh) in the left margin of the source code pane, for a line of code, gives you access to the **Break Point Properties** window.

• If you click on an existing break point, the window is displayed for that break point.
• If you click on a line where no break point was set, the debugger creates one and displays the window for the newly created break point.

The **Break Point Properties** window is shown here:



Break Point Properties

Break Point

Location

Genealogy of, line: 7

Type

Temporary (disappears when method achieved) [✓ Persistent]

Break when following expression is true

Number of times to skip before breaking: 0

☐ Break Point is disabled.

Here are the properties:

Location: This tells you the name of the method and the line number where the break point is set. You cannot change this information.

Type: By default, the debugger lets you create **persistent** break points, depicted by a red bullet in the source code pane of the debugger window. To create a temporary break point, select the **Temporary** option. A temporary break point is useful when you want to break just once in a method. A temporary break point is identified by a green bullet in the source code pane of the Debugger window. **Note:** You can also set a temporary break point directly in the source code pane by clicking in the left margin while pressing ALT+Shift (Windows) or Option+Shift (Macintosh).

Break when following expression is true: You can create conditional break points by entering a 4D formula that returns True or False. For example, if you want to break at a line only when Records in selection([aTable])=0, enter this formula, and the break will occur only if there no record selected for the table [aTable], when the debugger encounters the line with this break point. If you are not sure about the syntax of your formula, click the **Check Syntax** button.

Number of times to skip before breaking: You can set a break point to a line of code located in a loop structure (While, Repeat, or For) or located in subroutine or function called from within a loop. For example, you know that the "problem" you are tracking does not occur before at least the 200th iteration of the loop. Enter 200, and the break point will activate at the 201st iteration.

Break Point is disabled: If you currently do not need a persistent break point, but you may need it later, you can temporarily disable the break point by editing it. A disabled break point appears as a dash (-) instead of a bullet (•) in the source code pane of the debugger window and in the Break List window.

You create and edit break point from within the Debugger window. You can also edit existing break points using the Design environment Break List window. For more information, see the section Break List window.

See Also

Break List Window, Debugger, Source Code Pane.

Break List Window

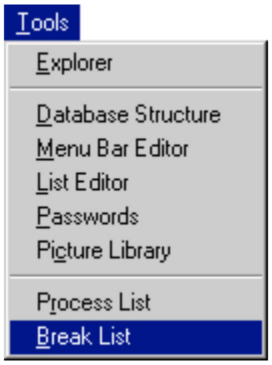
The Break List window is a Design Environment window that enables you to:

• Manage the Break Points created in the Debugger Window.
• Add additional breaks to your code by catching calls to 4D commands.

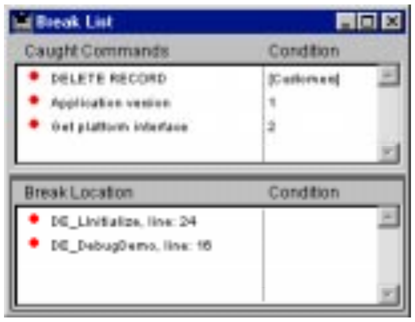
To open the Break List window:

1. Switch to the Design environment if you are not already there.

2. Choose **Break List** from the **Tools** menu.



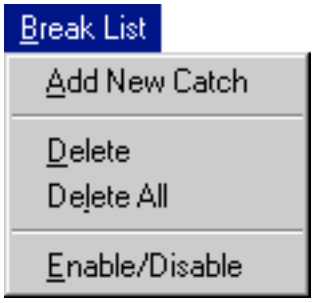
The Break List window appears.



Caught Commands	Condition
● DELETE RECORD	[Customers]
● Application version	1
● Get platform interface	2

Break Location	Condition
● DE_I_Initialize, line: 24	
● DE_DebugDemo, line: 16	

Note that when the Break List window is the frontmost window of the Design environment, the **Break List** menu appears in the main menu bar:



The Break List window has two panes, each composed of two columns:

• The top pane lists the commands to be caught during execution. The left column displays the Enable/Disable status of the caught command, followed by the name of the command. The right column displays the condition associated with the caught command, if any.

• The lower pane shows the persistent Break Points. The left column displays the Enable/Disable status of the break point, followed by the name of the method and the line number where the break point has been set (using the Debugger window). The right column displays the condition associated with the break point, if any.

3. To select a pane as the active pane of the window, click somewhere in the pane or use the Tab key.

Catching Commands

Catching a command enables you to start tracing the execution of any process as soon as a command is called by that process. Unlike a break point, which is located in a particular project method (and therefore triggers a trace exception only when it is reached), the scope of catching a command includes all the processes that execute 4D code and call that command.

Catching a command is a convenient way to trace large portions of code without setting break points at arbitrary locations. For example, if a record that should not be deleted is deleted after you have executed one or several processes, you can try to reduce the field of your investigation by catching commands such as DELETE RECORD and DELETE SELECTION. Each time these commands are called, you can check if the record in question has been deleted, and thus isolate the faulty part of the code.

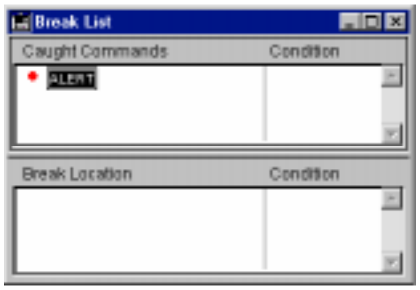
With some experience, you can combine the use of break points and command catching.

Adding a New Command to be Caught

To add a new command:

1. Choose **Add New Catch** from the **Break List** menu.
   OR
   Double-click the left mouse button in the Caught Commands list.

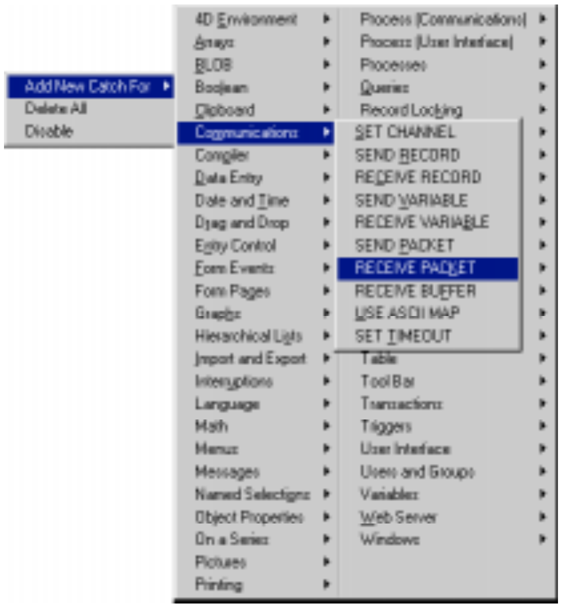
In both cases, a new entry is added to the list with the ALERT command as default. The entry is set to the edit mode.



2. Enter the name of the command you want to catch.

3. Press Enter or Return to validate your choice.

4. Press the right mouse button (Control-Click on Macintosh) to display the speed menu:



5. Select **Add New Catch**, then select the desired command from the command themes and names submenus. A new entry is added with the command you selected.

Editing the Name of a Caught Command

To edit the name of a caught command:

1. Select the entry by clicking it or by using the arrow keys to navigate through the list (if the current selected entry is not already in edit mode).

2. To toggle an entry between edit mode and select mode, press Enter or Return.

3. Enter or modify the name of the command.

4. To validate your changes, press Enter or Return. If name you entered does not correspond to an existing 4D command, the entry is set to its previous value. If the entry is a new one, it is reset to ALERT.

Disabling/Enabling a Caught Command

To disable or enable a caught command:

1. Select the entry by clicking it or by using the arrow keys to navigate through the list (if the current selected entry is not already in edit mode).

2. If the entry is in edit mode, press Enter or Return to switch to select mode.

3. Choose **Enable/Disable** from the Break List menu or from the speed menu.

Shortcut: Each entry in the list may be disabled/enabled by clicking on the bullet (•). The bullet changes to a dash (–) when disabled.

Deleting a Caught Command

To delete a caught command:

1. Select the entry by clicking it or by using the arrow keys to navigate through the list (if the current selected entry is not already in edit mode).

2. If the entry is in edit mode, press Enter or Return to switch to select mode.

3. Press the Delete key, choose **Delete** from the Break List menu or choose **Delete** from the speed menu.

Note: To delete all the caught commands, choose **Delete All** from the Break List menu or from the speed menu.

Setting a Condition for Catching a Command

To set a condition for catching a command:

1. Click on the entry in the right column.

2. Enter a 4D formula (expression, command call or project method) that returns a Boolean value.

Note: To remove a condition, delete its formula.

Break Points

The Break Point pane displays only the persistent break points created in the Debugger window. Unlike the Caught Commands pane, you cannot add a new persistent break point from this pane. Persistent break points can only be created from within the Debugger window.

Disabling/Enabling a Break Point

To disable or enable a break point:

1. Select the entry by clicking on it or by using the arrows to navigate through the list (if the current selected entry is not already in edit mode).

2. If the entry is in edit mode, press Enter or Return to switch it to select mode.

3. Choose **Enable/Disable** from the Break List menu or from the speed menu.

Shortcut: Each entry in the list may be disabled/enabled by clicking directly on the bullet (•). The bullet changes to a dash (–) when disabled.

Deleting a Break Point

To delete a break point:

1. Select the entry by clicking on it or by using the arrows to navigate through the list (if the current selected entry is not already in edit mode).

2. If the entry is in edit mode, press Enter or Return to switch it to select mode.

3. Press the Delete key, choose **Delete** from the Break List menu or choose **Delete** from the speed menu.

Note: To delete all the break points, choose **Delete All** from the Break List menu or from the speed menu.

Setting a Condition for a Break Point

To set a condifition for a break point, proceed as follows:

1. Click on the entry in the right column

2. Enter a 4D formula (expression or command call or project method) that returns a Boolean value.

Note: To remove a condition, delete its formula.

Tips

• Adding conditions to caught commands or break points slows the execution, because the condition has to be evaluated each time an exception is met. On the other hand, adding consitions accelerates the debugging process, because it automatically skips occurrences that do not match the conditions.

• Disabling a caught command or break point has almost the same effect as deleting it. During execution, the debugger spends almost no time on the entry. The advantage of disabling an entry is that you do not have to recreate it when you need it again.

See Also

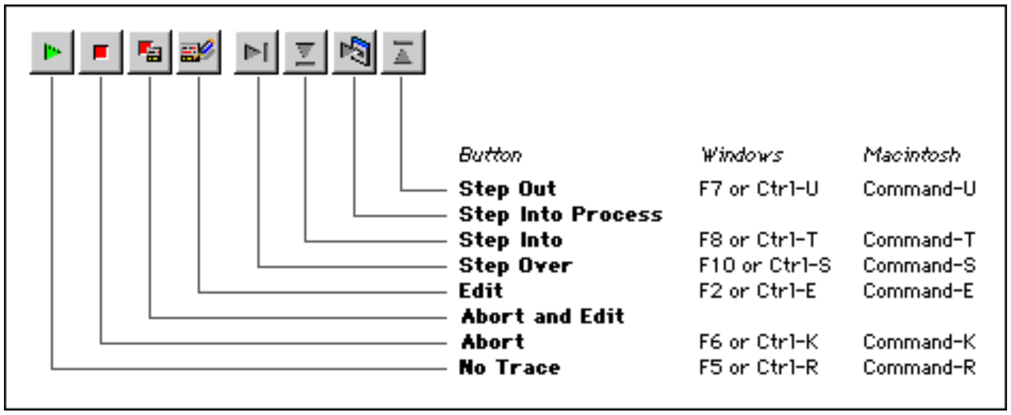
Break Points, Debugger, Source Code Pane, Why a Debugger?.

Debugger Shortcuts

This section lists all the shortcuts provided by the Debugger window.

Execution Control Tool Bar

→ The following figure shows the shortcuts for the eight buttons located in the top left corner of the Debugger Window:



Button	*Windows*	*Macintosh*
Step Out	F7 or Ctrl-U	Command-U
Step Into Process		
Step Into	F8 or Ctrl-T	Command-T
Step Over	F10 or Ctrl-S	Command-S
Edit	F2 or Ctrl-E	Command-E
Abort and Edit		
Abort	F6 or Ctrl-K	Command-K
No Trace	F5 or Ctrl-R	Command-R

→ ALT+F5 (Windows) and Option-Command-R (Macintosh) resume the execution. Also, they disable all the next TRACE calls for the current process.

Watch Pane

→ Right mouse button click (Windows) or Control-Click (Macintosh) in the Watch pane pulls down the Watch Speed menu.

→ Double-click on an item of the Watch pane copies the item to the Custom Watch pane.

Call Chain Pane

→ Double-Click on a method name in the Call chain pane displays the method in the Source Code pane at the line corresponding to the call in the call chain.

Custom Watch Pane

→ Right mouse button click (Windows) or Control-Click (Macintosh) in the Custom Watch pane pulls down the Custom Watch Speed menu.

→ Double-Click in the Custom Watch pane creates a new watch.

Source Code Pane

→ Click in the left margin sets (persistent) or removes break points.

→ ALT-Shift-Click (Windows) or Option-Shift Click (Macintosh) sets a temporary break point.

→ Alt-Click (Windows) or Option-Click displays the **Edit Break** window for a new or existing break point.

→ Click on an evaluable expression in the source code pane copies the expression or object to the Custom Watch pane.

All Panes

→ When no item is selected in any pane, typing **Enter** steps by one line.

→ When an item value is selected, use the arrows keys to navigate through the list.

→ When an item is being edited, use the arrow keys to move the cursor; use Ctrl-A/X/C/V (Windows) or Command-A/X/C/V (Macintosh) as shortcuts to the Select All/Cut/Copy/Paste menu commands of the **Edit** menu.

See Also

Call Chain Pane, Custom Watch Pane, Debugger, Source Code Pane, Watch Pane.

Tracing a Process not visible or not executing code

The Debugger's **Step Into Process** button allows you to trace a process at the moment you start it using the command New Process.

You may also want to trace a process long after the process has been started.

If the process has at least one visible window and if it is the frontmost window, pressing Alt+Click (Windows) or Option-Click (Macintosh) in that window starts the trace mode for the process.

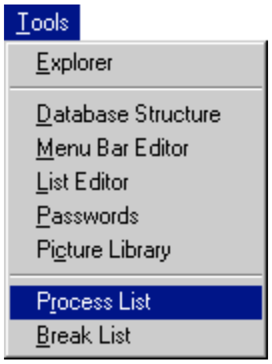
Using Alt+Click or Option+Click could prove quite difficult if the process:
• is executing code, but its windows are behind other windows and you do want to move them around to access the process
• is executing code but does not have any user interface (no windows)
• is in data entry and waiting for an event (*)
• is currently paused (*)
• is currently delayed (*)

(*) means running but not executing code.

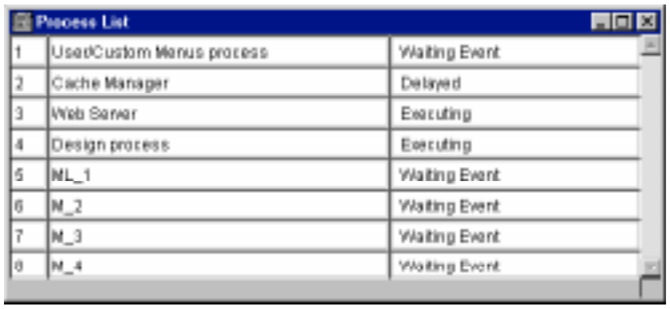
4D provides another convenient way to start tracing a process; this technique does not require the process to be “visible” or to be executing code.

1. Switch to the Design environment if you are not already in it.

2. Choose **Process List** from the **Tools** menu.



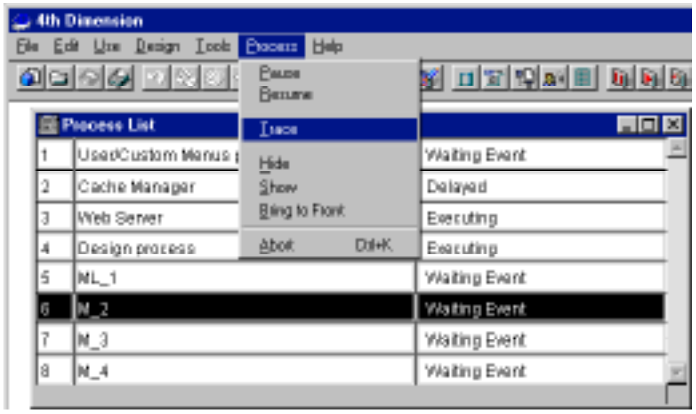
3. This displays the Process List window shown. The **Process List** window displays the processes that are currently running (whether or not they are executing).


Process List

1	User/Custom Menus process	Waiting Event
2	Cache Manager	Delayed
3	Web Server	Executing
4	Design process	Executing
5	ML_1	Waiting Event
6	M_2	Waiting Event
7	M_3	Waiting Event
8	M_4	Waiting Event

4. Select the process that you want to trace, by clicking on it.

5. Choose **Trace** from the **Process** menu.



Process List

1	User/Custom Menus	Waiting Event
2	Cache Manager	Delayed
3	Web Server	Executing
4	Design process	Executing
5	ML_1	Waiting Event
6	M_2	Waiting Event
7	M_3	Waiting Event
8	M_4	Waiting Event

The interesting point here is that 4D "memorizes" the Trace request:
• If the process is currently executing code, the Debugger immediately appears for that process.
• If the process is not currently executing code (i.e., the process is waiting for an event in data entry mode), the Debugger will appear right after the process resumes executing the code.

Tip: You may want to trace the object method for a button when you click on it. Alt+Click (Windows) or Option-Click (Macintosh) may or may not work, depending on the "speed" of the click. In this case, use the **Trace** menu command from the **Process** menu. As soon as the object method starts, you'll get the Debugger. Otherwise, you can also place a TRACE call in the method itself.

14 Drag and Drop

Drag and Drop

Version 6 of 4th Dimension introduces built-in drag and drop capability between objects in your forms. You can drag and drop one object to another, in the same window or in another window. In other words, drag and drop can be performed within a process or from one process to another.

Version 6.0 does not include built-in drag and drop to and from the desktop or another application. However, this functionality is provided by plug-ins developed by ACI Partners.

Note: As an introduction, we assume that a drag and drop action “transports” some data from one point to another. Later, we will see that drag and drop can also be a metaphor for an operation.

Dragable and Dropable Object Properties

To drag and drop an object to another object, you must select the **Dragable property** for that object in the Object Properties window. In a drag and drop operation, the object that you drag is the **source object**.

To make an object the destination of a drag and drop operation, you must select the **Dropable property** for that object in the Object Properties window. In a drag and drop operation, the object that receives data is the **destination object**.

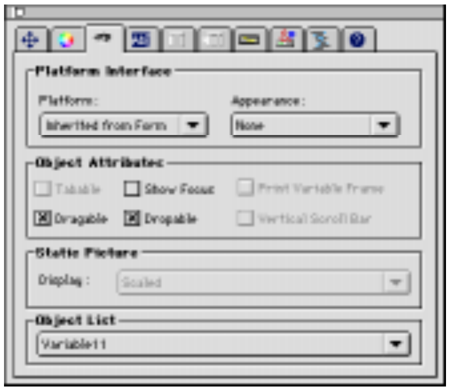
By default, newly created objects can be neither dragged nor dropped. It is up to you to set these properties.

All objects in an input or dialog form can be made to be dragged and dropped. Individual elements of an array (i.e., scrollable area) or items of a hierarchical list can be dragged and dropped. Conversely, you can drag and drop an object over an individual element of an array or item of a hierarchical list. However, you cannot drag and drop objects from the detail area of an output form.

You can easily create a drag and drop user interface, because 4D allows you to use any type of active object (field or variable) as source or destination objects. For example, you can drag and drop a button.

Note: An object that is capable of being both dragged and dropped can also be dropped onto itself, unless you reject the operation. For details, see the discussion below.

The following figure shows the Object Properties window with the Dropable and Dragable properties set for the selected objects:


Platform Interface

Platform: Inherited from Form

Appearance: None

Object Attributes

- [ ] Tabable
- [ ] Show Focus
- [ ] Print Variable Frame
- [x] Dragable
- [x] Dropable
- [ ] Vertical Scroll Bar

Static Picture

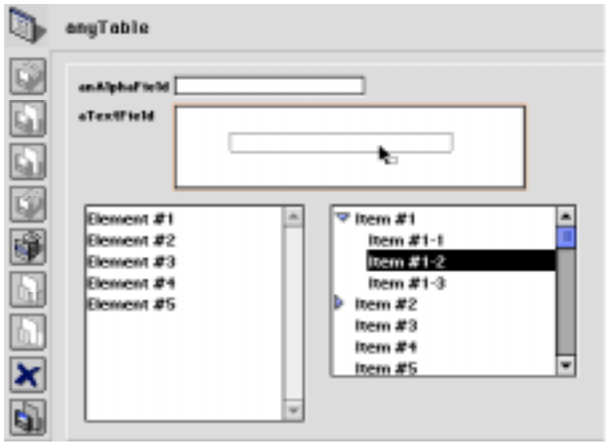
Display: Scaled

Object List

Variable11

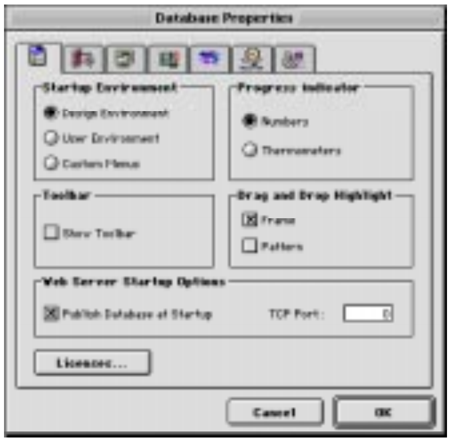
Drag and Drop User Interface Handling

4th Dimension insures the user interface part of the drag and drop capability. If you click on a dragable object and then drag the mouse, 4D drags the object; it reflects this operation on the screen with a dotted rectangle that follows the movements of the mouse. In the following figure, a hierarchical list item is being dragged over a text field:



Note the reverse gray frame highlight around the text field area. The highlight indicates the destination object (in this case, the text field). If you release the mouse button at this point, 4D assumes that you want to drop the dragged object onto the highlighted destination object.

In the **Database Properties** dialog box, you can set the drag and drop highlight of the destination object to be a frame or a pattern (or both):


Database Properties

Startup Environment: [● Design Environment] ○ User Environment ○ Custom Menus

Progress Indicator: [● Numbers] ○ Thermometers

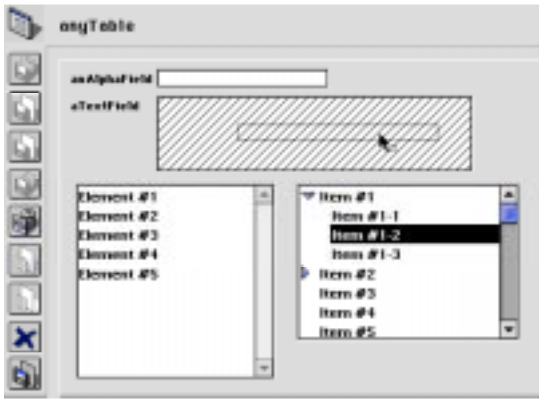
Toolbar: ☐ Show Toolbar

Drag and Drop Highlight: [☒ Frame] ☐ Pattern

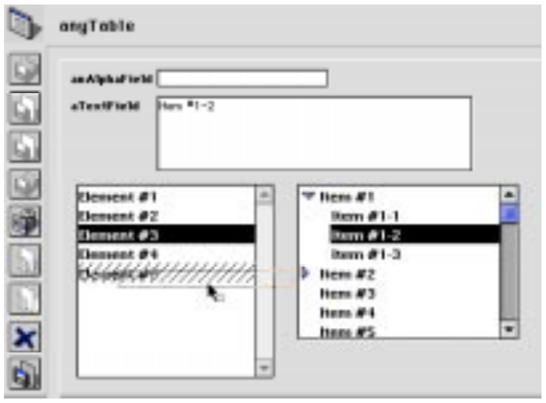
Web Server Startup Options: [☒ Publish Database at Startup] TCP Port: 0

The default highlight is **Frame**. It is a rectangular, gray, reverse highlight around the object. If you use colored background or object frames, using this highlight may be confusing. You can alternatively use the **Pattern** highlight, which fills the destination object with a diagonal lines pattern, as shown.

Here a hierarchical list item is dragged over a text field:



Here an array element is dragged over its array:



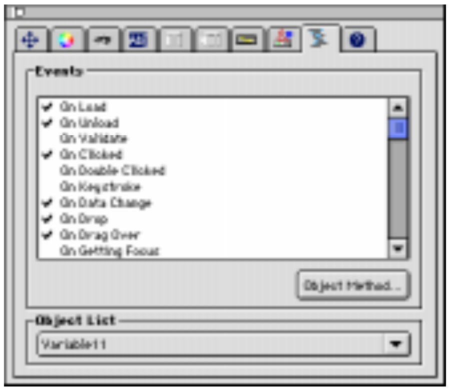
You can also choose both types of highlight.

Note: The highlight of the destination object "follows" elements or items when the destination object is an array (scrollable area) or a hierarchical list.

Drag and Drop Programmatical Handling

4th Dimension performs the user interface part of a drag and drop—it is up to you to perform the programmatical part. To enable you to do so, 4D provides you with two form events: On Drag Over and On Drop. Both events are sent to the **destination object**. During a drag and drop operation, the object method of the source object is never involved.

In order to accept On Drag Over and On Drop, the destination object must have these two events activated in the Object Properties window, as shown here:



Events

✓ On Load
✓ On Unload
On Validate
✓ On Clicked
On Double Clicked
On Keystroke
✓ On Data Change
✓ On Drop
✓ On Drag Over
On Getting Focus

Object List: Variable11

On Drag Over

The On Drag Over event is repeatedly sent to the destination object when the mouse pointer is moved over the object. In response to this event, you usually:

• Call the DRAG AND DROP PROPERTIES command, which informs you about the source object.
• Depending on the nature and type of both the destination object (whose object method is currently being executed) and the source object, you **accept** or **reject** the drag and drop.

To accept the drag, the destination object method must return 0 (zero), so you write $0:=0. To reject the drag, the object method must return -1 (minus one), so you write $0:=-1. During an On Drag Over event, 4D treats the object method as a function. If no result is returned, 4D assumes that the drag is accepted.

If you accept the drag, the destination object is highlighted. If you reject the drag, the destination is not highlighted. Accepting the drag does not mean that the dragged data is going to be inserted into the destination object. It only means that if the mouse button was released at this point, the destination object would accept the dragged data.

If you do not process the On Drag Over event for a dropable object, that object will be highlighted for all drag over operations, no matter what the nature and type of the dragged data.

The On Drag Over event is the means by which you control the first phase of a drag and drop operation. Not only can you test if the dragged data is of a type compatible with the destination object, and then accept or reject the drag; you can simultaneously notify the user of this fact, because 4D highlights (or not) the destination object, based on your decision.

The code handling an On Drag Over event should be short and execute quickly, because that event is sent repeatedly to the current destination object, due to the movements of the mouse.

WARNING: If the drag and drop is an **interprocess drag and drop**, which means the source object is located in a process (window) other than that of the destination object, the object method of the destination object for an On Drag Over event is executed **within the context** of the **source process** (the source object's process), and not in the process of the destination object. This is the only case in which such an execution occurs. The advantages of this type of execution are described at the end of this section.

On Drop

The On Drop event is sent once to the destination object when the mouse pointer is released over the object. This event is the second phase of the drag and drop operation, in which you perform an operation in response to the user action.

This event is not sent to the object if the drag was not accepted during the On Drag Over events. If you process the On Drag Over event for an object and reject a drag, the On Drop event does not occur. Thus, if during the On Drag Over event you have tested the data type compatibility between the source and destination objects and have accepted a possible drop, you do not need to re-test the data during the On Drop. You already know that the data is suitable for the destination object.

An interesting aspect of the 4D drag and drop implementation is that 4D lets you do whatever you want. Examples:

• If a hierarchical list item is dropped over a text field, you can insert the text of the list item at the beginning, at the end, or in the middle of the text field.
• Your form contains a two-state picture button, which could represent an empty or full trash can. Dropping an object onto that button could mean (from the user interface standpoint) “delete the object that has been dragged and dropped into the trash can.” Here, the drag and drop does not transport data from one point to another; instead, it performs an action.
• Dragging an array element from a floating window to an object in a form could mean “in this window, show the Customer record whose name you just dragged and dropped from the floating window listing the Customers stored in the database.”
• And so on.

So, the 4D drag and drop interface is a framework which enables you to implement any user interface metaphor you may devise.

Drag and drop commands

The DRAG AND DROP PROPERTIES command returns:
• a pointer to the dragged object (field or variable)
• the element or item number, if the dragged object is an array element or a list item
• the process number of the source process

The Drop position command returns the element number of the item position of the target element or list item, if the destination object is an array (i.e., scrollable area) or a hierarchical list,

Commands like RESOLVE POINTER and Type are useful for testing the nature and type of the source object.

When the drag and drop operation is intended to copy the dragged data, the functionality of these commands depend on how many processes are involved:
• If the drag and drop is limited to one process, use these commands to perform the appropriate actions (i.e., simply assigning the source object to the destination object).
• If the drag and drop is an interprocess drag and drop, you need to be careful while getting access to the dragged data; you must access the data instance from the source process. If the dragged data comes from a variable, use GET PROCESS VARIABLE to get the right value. If the dragged data comes from a field, remember that the current record for a table is probably different for the two processes, so you need to access the right record.

In this last case, several solutions are available:
• If the On Drag Over event for the destination object method is executed in the context of the source process, you can copy the field data or the record number to an interprocess variable that will be reused during the On Drop event.
• You can get the required data by starting an interprocess communication during the On Drop event.

If the drag and drop is not intended to move data, but is instead a user interface metaphor for a particular operation, you can perform whatever you want.

See Also

DRAG AND DROP PROPERTIES, Drop position, Form event, GET PROCESS VARIABLE, Is a list, RESOLVE POINTER, Type.

Drop position

Drop position → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Destination element number or item position, or -1 if drop occurred beyond the last array element or list item

Description

The command Drop position returns the array element number or list item position onto which an object has been dragged and dropped.

Typically, you will use Drop position while handling a drag and drop event that occurred over an array or a hierarchical list.

If the destination object is an array, the command returns an element number. If the destination object is a hierarchical list, the command returns an item position. In both cases, the command may return -1 if the source object has been dropped beyond the last element or the last item.

If you call Drop position while handling an event that is not a drag and drop event and that occurred over an array or a hierarchical list, the command returns -1.

Important: A form object accepts dropped data if its **Dropable** property has been selected. Also, its object method must be activated for On Drag Over and/or On Drop, in order to process these events.

Example

See examples for the command DRAG AND DROP PROPERTIES.

See Also

Drag and drop, DRAG AND DROP PROPERTIES.

DRAG AND DROP PROPERTIES

version 6.0

DRAG AND DROP PROPERTIES (srcObject; srcElement; srcProcess)

Parameter	Type		Description
srcObject	Pointer	←	Pointer to drag and drop source object
srcElement	Number	←	Dragged array element number, or Dragged hierarchical list item, or -1 if source object is neither an array nor a list
srcProcess	Number	←	Source process number

Description

The command DRAG AND DROP PROPERTIES enables you to obtain the information about the source object when an On Drag Over or On Drop event occurs for an object.

Typically, you use DRAG AND DROP PROPERTIES from within the object method of the object (or from one of the subroutines it calls) for which the On Drag Over or On Drop event occurs (the destination object).

Important: A form object accepts dropped data if its **Dropable** property has been selected. Also, its object method must be activated for On Drag Over and/or On Drop, in order to process these events.

After the call:

• The parameter srcObject is a pointer to the source object (the object that has been dragged and dropped). Note that this object can be the destination object (the object for which the On Drag Over or On Drop event occurs) or a different object. Dragging and dropping data from and to the same object is useful for arrays and hierarchical lists—it is a simple way of allowing the user to sort an array or a list manually.

• If the dragged and dropped data is an array element (an element of the source object being an array), the parameter srcElement returns the number of that element. Otherwise, if the drag and dropped data is a list item (an item of the source object being a hierarchical list), the parameter srcElement returns the position of that item. Otherwise, if the source object is neither an array nor a hierarchical list, srcElement is equal to -1.

• Drag and drop operations can occur between processes. The parameter srcProcess is equal to the number process to which the source object belongs. It is important to test the value of this parameter. You can respond to a drag and drop within the same process by simply copying the source data to the destination object. On the other hand, while treating an interprocess drag and drop, you will use the command GET PROCESS VARIABLE to get the source data from the source process object instance. If the source object is a field, you must get the value from the source process via interprocess communication or handle that particular case while responding to the On Drag Over event (see below).

However, you will usually implement drag and drop user interface from source variables (i.e., arrays and lists) toward data entry areas (fields or variables).

If you call DRAG AND DROP PROPERTIES while there is no drag and drop event, srcObject returns a NIL pointer, srcElement returns -1 and srcProcess returns 0.

Tip: 4th Dimension automatically handles the graphical aspect of a drag and drop. You must then respon to the event in the appropriate way. In the following examples, the response is to copy the data that has been dragged. Alternatively, you can implement sophisticated user interfaces where, for example, dragging and dropping an array element from a floating window will fill in the destination window (the window where the destination object is located) with structured data (i.e., several fields coming from a record uniquely identified by the source array element).

You use DRAG AND DROP PROPERTIES during an On Drag Over event in order to decide whether the destination object accepts the drag and drop operation, depending on the type and/or the nature of the source object (or any other reason). If you accept the drag and drop, the object method must return $0:=0. If you do not accept the drag and drop, the object method must return $0:=-1. Accepting or refusing the drag and drop is reflected at the screen—the object is or is not highlighted as the potential destination of the drag and drop operation.

Tip: During an On Drag Over event, the object method of the destination object is executed within the context of the source object's process. If the source object of an interprocess drag and drop is a field, you can use the opportunity of this event to copy the source data into an interprocess variable. In doing so, then later on, during the On Drop event, you will not have to initiate an interprocess communication with the source process in order to get the value of the field that was dragged. If an interprocess drag and drop involves a variable as source object, you can use the GET PROCESS VARIABLE command during the On Drop event.

Examples

1. In several of your database forms, there are scrollable areas in which you want to manually reorder the elements by simple drag and drop from one part of the scrollable area into another within it. Rather than writing specific code for each case, you may implement a generic project method that will handle any one of these scrollable areas. You could write something like:

```
      ` Handle self array drag and drop project method
      ` Handle self array drag and drop ( Pointer ) -> Boolean
      ` Handle self array drag and drop ( -> Array ) -> Was a self array drag and drop

   Case of
      : (Form event=On Drag Over)
         DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
         If ($vpSrcObj=$1)
```

```
            ` Accept the drag and drop if it is from the array to itself
            $0:=0
        Else
            $0:=-1
        End if
    : (Form event=On Drop)
            ` Get the information about the drag and drop source object
        DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
            ` Get the destination element number
        $vlDstElem:=Drop position
            ` If the element was not dropped over itself
        If ($vlDstElem # $vlSrcElem)
                ` Save dragged element in element 0 of the array
            $1->{0}:=$1->{$vlSrcElem}
                ` Delete the dragged element
            DELETE ELEMENT($1->;$vlSrcElem)
                ` If the destination element was beyond the dragged element
            If ($vlDstElem>$vlSrcElem)
                    ` Decrement the destination element number
                $vlDstElem:=$vlDstElem-1
            End if
                ` If the drag and drop occured beyond the last element
            If ($vlDstElem=-1)
                    ` Set the destination element number to a new element
                    ` at the end of the array
                $vlDstElem:=Size of array($1->)+1
            End if
                ` Insert this new element
            INSERT ELEMENT($1->;$vlDstElem)
                ` Set its value which was previously saved in the element zero of the array
            $1->{$vlDstElem}:=$1->{0}
                ` The element becomes the new selected element of the array
            $1->:=$vlDstElem
        End if
End case
```

Once you have implemented this project method, you can use it in the following way:

```
    ` anArray Scrollable Area Object Method

Case of
            `...
    : (Form event=On Drag Over)
        $0:=Handle self array drag and drop (Self)
    : (Form event=On Drop)
        Handle self array drag and drop (Self)
            ` ...
End case
```

2. In several of your database forms, you have text enterable areas in which you want to drag and drop data from various sources. Rather than writing specific code for each case, you may implement a generic project method that will handle any one of these text enterable areas. You could write something like:

```
      ` Handle dropping to text area project method
      ` Handle dropping to text area ( Pointer )
      ` Handle dropping to text area ( -> Text or String variable )

   Case of
         ` Use this event for accepting or rejecting the drag and drop
      : (Form event=On Drag Over)
            ` Initialize $0 for rejecting
         $0:=-1
            ` Get the information about the drag and drop source object
         DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
            ` In this example, we do not allow drag and drop from an object to itself
         If ($vpSrcObj # $1)
               ` Get the type of the data which is being dragged
            $vlSrcType:=Type($vpSrcObj->)
            Case of
               : ($vlSrcType=Is Alpha Field)
                     ` Alphanumeric Field is OK
                  $0:=0
                     ` Copy the value now into an IP variable
                  <>vtDraggedData:=$vpSrcObj->
               : ($vlSrcType=Is Text)
                     ` Text Field or Variable is OK
                  $0:=0
                  RESOLVE POINTER($vpSrcObj;$vsVarName;$vlTableNum;$vlFieldNum)
                     ` If it is a field
                  If (($vlTableNum>0) & ($vlFieldNum>0))
                        ` Copy the value now into an IP variable
                     <>vtDraggedData:=$vpSrcObj->
                  End if
               : ($vlSrcType=Is String Var)
                     ` String Variable is OK
                  $0:=0
               : (($vlSrcType=String array) | ($vlSrcType=Text array))
                     ` String and Text Arrays are OK
                  $0:=0
               : (($vlSrcType=Is LongInt) | ($vlSrcType=Is Real)
                  If (Is a list($vpSrcObj->))
                        ` Hierarchical list is OK
                     $0:=0
                  End if
            End case
         End if
         ` Use this event for performing the actual drag and drop action
      : (Form event=On Drop)
         $vtDraggedData:=""
            ` Get the information about the drag and drop source object
         DRAG AND DROP PROPERTIES($vpSrcObj;$vlSrcElem;$vlPID)
         RESOLVE POINTER($vpSrcObj;$vsVarName;$vlTableNum;$vlFieldNum)
            ` If it is field
         If (($vlTableNum>0) & ($vlFieldNum>0))
               ` Just grab the IP variable set during the On Drag Over event
            $vtDraggedData:=<>vtDraggedData
         Else
               ` Get the type of the variable which has been dragged
            $vlSrcType:=Type($vpSrcObj->)
            Case of
                  ` If it is an array
               : (($vlSrcType=String array) | ($vlSrcType=Text array))
                  If ($vlPID # Current process)
                        ` Read the element from the source process instance
                        ` of the variable
                     GET PROCESS VARIABLE($vlPID;
                                          $vpSrcObj->{$vlSrcElem};$vtDraggedData)
                  Else
                        ` Copy the array element
                     $vtDraggedData:=$vpSrcObj->{$vlSrcElem}
                  End if
               : (($vlSrcType=Is LongInt) | ($vlSrcType=Is Real)
                     ` If it is a hierarcical list
                  If (Is a list($vpSrcObj->))
                        ` If it is a list from another process
                     If ($vlPID # Current process)
                           ` Get the List Reference from the other process
                        GET PROCESS VARIABLE($vlPID;$vpSrcObj->;$vlList)
                     Else
                        $vlList:=$vpSrcObj->
                     End if
                        ` Get the text of the item whose position was obtained
                     GET LIST ITEM($vlList;$vlSrcElem;$vlItemRef;$vsItemText)
                     $vtDraggedData:=$vsItemText
                  End if
            Else
               ` It is a string or a text variable
               If ($vlPID # Current process)
                  GET PROCESS VARIABLE($vlPID;$vpSrcObj->;$vtDraggedData)
               Else
                  $vtDraggedData:=$vpSrcObj->
               End if
            End case
         End if
```

```
          ` If there is actually something to drop (the source object may be empty)
      If ($vtDraggedData # "")
              ` Check that the length of the text variable will not exceed 32,000 char.
          If ((Length($1->)+Length($vtDraggedData))<=32000)
              $1->:=$1->+$vtDraggedData
          Else
              BEEP
              ALERT("The drag and drop cannot be completed because
                                                  the text would become too long.")
          End if
      End if

End case
```

Once you have implemented this project method, you can use it in the following way:

```
    ` [anyTable]aTextField Object Method

Case of
          ` ...
    : (Form event=On Drag Over)
        $0:=Handle dropping to text area (Self)

    : (Form event=On Drop)
        Handle dropping to text area (Self)
          ` ...
End case
```

See Also

Drag and Drop, Drop position, Form event, GET PROCESS VARIABLE, Is a list, RESOLVE POINTER.

15 Entry Control

ACCEPT

Parameter	**Type**	**Description**

This command does not require any parameters

Description

The command ACCEPT is used in form or object methods (or in subroutines) to:

• accept a new or modified record or subrecord, for which data entry has been initiated using ADD RECORD, MODIFY RECORD, ADD SUBRECORD, or MODIFY SUBRECORD
• accept a form displayed with the DIALOG command
• exit a form displaying a selection of records, using DISPLAY SELECTION or MODIFY SELECTION

ACCEPT performs the same action as if a user had pressed the Enter key. After the form is accepted, the OK system variable is set to 1.

ACCEPT is commonly executed as a result of choosing a menu command. ACCEPT is also commonly used in the object method of a “no action” button.

It is also often used in the optional close box method for the Open window command. If there is a Control-menu box on a window, ACCEPT or CANCEL can be called, in the method to be executed, when the Control-menu box is double-clicked or the Close menu command is chosen.

ACCEPT cannot be queued up. In response to an event, executing two ACCEPT commands in a row from within a method would have the same effect as executing one.

See Also

CANCEL.

CANCEL

Parameter	**Type**	**Description**

This command does not require any parameters

Description

The command CANCEL is used in form or object methods (or in a subroutine) to:

• cancel a new or modified record or subrecord, for which data entry has been initiated using ADD RECORD, MODIFY RECORD, ADD SUBRECORD, or MODIFY SUBRECORD.
• cancel a form displayed with the DIALOG command.
• exit a form displaying a selection of records, using DISPLAY SELECTION or MODIFY SELECTION.

CANCEL performs the same action as if the user had pressed the cancel key combination (Ctrl-Period on Windows, Command-Period on Macintosh). After the form is canceled, the OK system variable is set to 0 (zero).

CANCEL is commonly executed as a result of a menu command being chosen. CANCEL is also commonly used in the object method of a “no action” button.

It is also often used in the optional close box method for the Open window command. If there is a Control-menu box on a window, ACCEPT or CANCEL can be called, in the method to be executed, when the Control-menu box is double-clicked or the Close menu command is chosen.

CANCEL cannot be queued up. Executing two CANCEL commands in a row from within a method in response to an event would have the same effect as executing one.

See Also

ACCEPT.

Keystroke

Entry Control

version 6.0

Keystroke → string

Parameter	Type		Description
This command does not require any parameters			
Function result	string	←	character entered by user

Description

Keystroke returns the character entered by the user into a field or an enterable area.

Usually, you will call Keystroke within a form or object method while handling an On Keystroke event form. To detect keystroke events, use the command Form event.

To replace the character actually entered by the user with another character, use the command FILTER KEYSTROKE.

IMPORTANT NOTE: If you want to perform some "on the fly" operations depending on the current value of the enterable area being edited, as well as the new character to be entered, remember that the text you see on screen is NOT YET the value of the data source field or variable for the area being edited. The data source field or variable is assigned the entered value after the data entry for the area is validated (e.g., tabulation to another area, click on a button, and so on). It is therefore up to you to "shadow" the data entry into a variable and then to work with this shadow value. You must do so if you need to know the current text value for executing any particular actions.

You will use the command Keystroke for:
• Filtering characters in a customized way
• Filtering data entry in a way that you cannot produce using data entry filters
• Implement dynamic lookup or type-ahead areas

Examples

1. See examples for the command FILTER KEYSTROKE.

2. When you process an On Keystroke event, you are dealing with the editing of the current text area (the one where the cursor is), not with the "future value" of the data source (field or variable) for this area. The Handle keystroke project method allows to shadow any text area data entry into a second variable, which you can use to perform the actions while entering characters into the area. You pass a pointer to the area's data source as the first parameter and a pointer to the shadow variable as second parameter. The method returns the new value of the text area in the shadow variable, and returns True if the value is different from it what was before the last entered character was inserted.

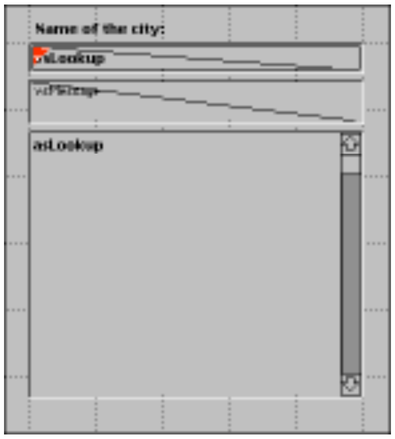
```
    ` Handle keystroke project method
    ` Handle keystroke ( Pointer ; Pointer ) -> Boolean
    ` Handle keystroke ( -> srcArea ; -> curValue ) -> Is new value

C_POINTER ($1;$2)
C_TEXT ($vtNewValue)

    ` Get the text selection range within the enterable area
GET HIGHLIGHT ($1->;$vlStart;$vlEnd)
    ` Start working with the current value
$vtNewValue:=$2->
    ` Depending on the key pressed or the character entered,
    ` Perform the appropriate actions
Case of

        ` The Backspace (Delete) key has been pressed
⇒       : (Ascii (Keystroke)=Backspace )
            ` Delete the selected characters or the character at the left of the text cursor
        $vtNewValue:=Substring ($vtNewValue;1;$vlStart-1-Num($vlStart=$vlEnd))
                                                            +Substring($vtNewValue;$vlEnd)

        ` An acceptable character has been entered
⇒       : (Position (Keystroke;"abcdefghjiklmnopqrstuvwxyz -0123456789")>0)
        If ($vlStart#$vlEnd)
                ` One or several characters are selected,
                ` the keystroke is going to override them
⇒           $vtNewValue:=Substring($vtNewValue;1;$vlStart-1)
                                            +Keystroke+Substring($vtNewValue;$vlEnd)
        Else
                ` The text selection is the text cursor
            Case of
                    ` The text cursor is currently at the begining of the text
                : ($vlStart<=1)
                        ` Insert the character at the begining of the text
⇒                   $vtNewValue:=Keystroke+$vtNewValue
                        ` The text cursor is currently at the end of the text
                : ($vlStart>=Length($vtNewValue))
                        ` Append the character at the end of the text
⇒                   $vtNewValue:=$vtNewValue+Keystroke
            Else
                    ` The text cursor is somewhere in the text, insert the new character
⇒               $vtNewValue:=Substring($vtNewValue;1;$vlStart-1)
                                            +Keystroke+Substring($vtNewValue;$vlStart)
            End case
        End if
```

```
              ` An Arrow key has been pressed
              ` Do nothing, but accept the keystroke
⇒         : (Ascii(Keystroke)=Left Arrow Key )
⇒         : (Ascii(Keystroke)=Right Arrow Key )
⇒         : (Ascii(Keystroke)=Up Arrow Key )
⇒         : (Ascii(Keystroke)=Down Arrow Key )
              `
      Else
          ` Do not accept characters other than letters, digits, space and dash
          FILTER KEYSTROKE ("")
      End case
          ` Is the value now different?
      $0:=($vtNewValue#$2->)
          ` Return the value for the next keystroke handling
      $2->:=$vtNewValue
```

After this project method is added to your application, you can use it as follows:

```
          ` myObject enterable area object method
      Case of
          : (Form event=On Load)
              MyObject:=""
              MyShadowObject:=""
          : (Form event=On Keystroke)
              If (Handle keystroke (->MyObject;->MyShadowObject))
                  ` Perform appropriate actions using the value stored in MyShadowObject
              End if
      End case
```

Let's examine the following part of a form:

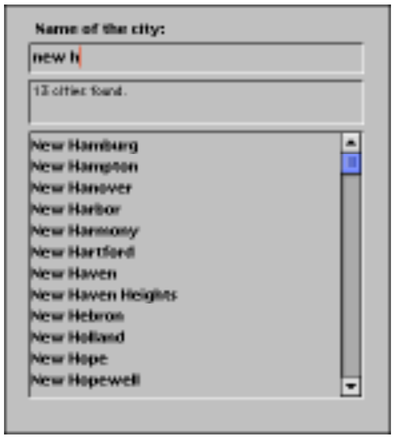


It is composed of the following objects: an enterable area vsLookup, a non-enterable area vsMessage, and a scrollable area asLookup. While entering characters in vsLookup, the method for that object performs a query on a [US Zip Codes] table, allowing the user to find US cities by typing only the first characters of the city names.

The vsLookup object method is listed here:

```
	` vsLookup enterable area object method
Case of
	: (Form event=On Load )
		vsLookup:=""
		vsResult:=""
		vsMessage:="Enter the first characters of the city you are looking for."
		CLEAR VARIABLE(asLookup)
	: (Form event=On Keystroke )
		If (Handle keystroke (->vsLookup;->vsResult))
			If (vsResult#"")
				QUERY([US Zip Codes];[US Zip Codes]City=vsResult+"@")
				MESSAGES OFF
				DISTINCT VALUES([US Zip Codes]City;asLookup)
				MESSAGES ON
				$vlResult:=Size of array(asLookup)
				Case of
					: ($vlResult=0)
						vsMessage:="No city found."
					: ($vlResult=1)
						vsMessage:="One city found."
				Else
					vsMessage:=String($vlResult)+" cities found."
				End case
			Else
				DELETE ELEMENT(asLookup;1;Size of array(asLookup))
				vsMessage:="Enter the first characters of the city you are looking for."
			End if
		End if
End case
```

Here is the form in the User environment:


Name of the city:

new h

13 cities found.

New Hamburg
New Hampton
New Hanover
New Harbor
New Harmony
New Hartford
New Haven
New Haven Heights
New Hebron
New Holland
New Hope
New Hopewell

Using the interprocess communication capabilities of 4th Dimension, you can similarily build user interfaces in which Lookup features are provided in floating windows that communicate with processes in which records are listed or edited.

See Also

FILTER KEYSTROKE, Form event.

FILTER KEYSTROKE

version 6.0

FILTER KEYSTROKE (filteredChar)

Parameter	Type		Description
filteredChar	String	→	Filtered keystroke character or Empty string to cancel the keystroke

Description

FILTER KEYSTROKE enables you to replace the character entered by the user into a field or an enterable area with the first character of the string filteredChar you pass.

If you pass an empty string, the keystroke is cancelled and ignored.

Usually, you will call FILTER KEYSTROKE within a form or object method while handling an On Keystroke form event. To detect keystroke events, use the command Form event. To obtain the actual keystroke, use the command Keystroke.

IMPORTANT NOTE: The command FILTER KEYSTROKE allows you to cancel or replace the character entered by the user with another character. On the other hand, if you want to insert more than one character for a specific keystroke, remember that the text you see on the screen is NOT YET the value of the data source field or variable for the area being edited. The data source field or variable is assigned the entered value after the data entry for the area is validated. It is therefore up to you to “shadow” the data entry into a variable and then to work with this shadow value and reassign the enterable area (see the example in this section).

You will use the command FILTER KEYSTROKE for:
• Filtering characters in a customized way
• Filtering data entry in a way that you cannot produce using data entry filters
• Implement dynamic lookup or type-ahead areas

WARNING: If you call the command Keystroke after calling FILTER KEYSTROKE, the character you pass to this command is returned instead of the character actually entered.

Examples

1. Using the following code:

```
          ` myObject enterable area object method
      Case of
          : (Form event=On Load )
          myObject:=""
          : (Form event=On Keystroke )
             If(Position(Keystroke;"0123456789")>0)
⇒               FILTER KEYSTROKE("*")
             End if
      End case
```

All the digits entered in the area myObject are transformed into star characters.

2. This code implements the behavior of a Password enterable area in which all the entered characters are replaced (on the screen) by random characters:

```
          ` vsPassword enterable area object method
      Case of
          : (Form event=On Load )
             vsPassword:=""
             vsActualPassword:=""
          : (Form event=On Keystroke )
             Handle keystroke (->vsPassword;->vsActualPassword)
             If (Position(Keystroke;Char(Backspace )+Char(Left Arrow Key )+
                   Char(Right Arrow Key )+Char(Up Arrow Key )+Char(Down Arrow Key ))=0)
⇒               FILTER KEYSTROKE(Char(65+(Random%26)))
             End if
      End case
```

After the data entry is validated, you retrieve the actual password entered by the user in the variable vsActualPassword. Note: The method Handle keystroke is listed in the Example section for the command Keystroke.

3. In your application, you have some text areas into which you can enter a few sentences. Your application also includes a dictionary table of terms commonly used throughout your database. While editing your text areas, you would like to be able to quickly retrieve and insert dictionary entries based on the selected characters in a text area. You have two ways to do this:
- Provide some buttons with associated keys, or
- Intercept special keystrokes during the editing of the text area

This example implements the second solution, based on the Help key.

As explained above, during the editing of the text area, the data source for this area will be assigned the entered value after you validate the data entry. In order to retrieve and insert dictionary entries into the text area while this area is being edited, you therefore need to shadow the data entry. You pass pointers to the enterable area and the shadow variable as the first two parameters, and you pass a string of the "forbidden" characters as the third parameter. No matter how the keystroke will be treated, the method returns the original keystroke. The "forbidden" characters are those that you do not want to be inserted into the enterable area and you want to treat as special characters.

```
    ` Shadow keystroke project method
    ` Shadow keystroke ( Pointer ; Pointer ; String ) -> String
    ` Shadow keystroke ( -> srcArea ; -> curValue ; Filter ) -> Old keystroke
C_STRING(1;$0)
C_POINTER($1;$2)
C_TEXT($vtNewValue)
C_STRING(255;$3)
    ` Return the original keystroke
$0:=Keystroke
    ` Get the text selection range within the enterable area
GET HIGHLIGHT($1->;$vlStart;$vlEnd)
    ` Start working with the current value
$vtNewValue:=$2->
    ` Depending on the key pressed or the character entered,
    ` Perform the appropriate actions
Case of
        ` The Backspace (Delete) key has been pressed
    : (Ascii($0)=Backspace )
            ` Delete the selected characters or the character at the left of the text cursor
        $vtNewValue:=Delete text ($vtNewValue;$vlStart;$vlEnd)
            ` An Arrow key has been pressed
            ` Do nothing, but accept the keystroke
    : (Ascii($0)=Left Arrow Key )
    : (Ascii($0)=Right Arrow Key )
    : (Ascii($0)=Up Arrow Key )
    : (Ascii($0)=Down Arrow Key )

        ` An acceptable character has been entered
    : (Position($0;$3)=0)
        $vtNewValue:=Insert text ($vtNewValue;$vlStart;$vlEnd;$0)
Else
        ` The character is not accepted
⇒       FILTER KEYSTROKE("")
End case
    ` Return the value for the next keystroke handling
$2->:=$vtNewValue
```

This method uses the two following submethods:

```
    ` Delete text project method
    ` Delete text ( String ; Long ; Long ) -> String
    ` Delete text ( -> Text ; SelStart ; SelEnd ) -> New text
C_TEXT($0;$1)
C_LONGINT($2;$3)
$0:=Substring($1;1;$2-1-Num($2=$3))+Substring($1;$3)

    ` Insert text project method
    ` Insert text ( String ; Long ; Long ; String ) -> String
    ` Insert text ( -> srcText ; SelStart ; SelEnd ; Text to insert ) -> New text
C_TEXT($0;$1;$4)
C_LONGINT($2;$3)
$0:=$1
If ($2#$3)
    $0:=Substring($0;1;$2-1)+$4+Substring($0;$3)
Else
    Case of
        : ($2<=1)
            $0:=$4+$0
        : ($2>Length($0))
            $0:=$0+$4
    Else
        $0:=Substring($0;1;$2-1)+$4+Substring($0;$2)
    End case
End if
```

After you have added these project methods to your project, you can use them in this way:

```
    ` vsDescription enterable area object method
Case of
    : (Form event=On Load )
        vsDescription:=""
        vsShadowDescription:=""
            ` Establish the list of the “forbidden” characters to be treated as special keys
            ` ( here, in this example, only the Help Key is filtered)
        vsSpecialKeys:=Char(HelpKey)
    : (Form event=On Keystroke )
        $vsKey:=Shadow keystroke (->vsDescription;->vsShadowDescription;vsSpecialKeys)
        Case of
            : (Ascii($vsKey)=Help Key )
                ` Do something when the Help key is pressed
                    ` Here, in this example, a Dictionary entry must be searched and inserted
                LOOKUP DICTIONARY (->vsDescription;->vsShadowDescription)
        End case
End case
```

The LOOKUP DICTIONARY project method is listed below. Its purpose is to use the shadow variable for reassigning the enterable area being edited:

```
    ` LOOKUP DICTIONARY project method
    ` LOOKUP DICTIONARY ( Pointer ; Pointer )
    ` LOOKUP DICTIONARY ( -> Enterable Area ; ->ShadowVariable )

C_POINTER($1;$2)
C_LONGINT($vlStart;$vlEnd)

    ` Get the text selection range within the enterable area
GET HIGHLIGHT($1->;$vlStart;$vlEnd)
    ` Get the selected text or the word on the left of the text cursor
$vtHighlightedText:=Get highlighted text ($2->;$vlStart;$vlEnd)
    ` Is there something to look for?
If ($vtHighlightedText#"")
        ` If the text selection was the text cursor,
        ` the selection now starts at the word preceeding the text cursor
    If ($vlStart=$vlEnd)
        $vlStart:=$vlStart-Length($vtHighlightedText)
    End if
        ` Look for the first avaliable dictionary entry
    QUERY([Dictionary];[Dictionary]Entry=$vtHighlightedText+"@")
        ` Is there one?
    If (Records in selection([Dictionary])>0)
            ` If so, insert it in the shadow text
        $2->:=Insert text ($2->;$vlStart;$vlEnd;[Dictionary]Entry)
            ` Copy the shadow text to the enterable being edited
        $1->:=$2->
            ` Set the selection just after the insert dictionary entry
        $vlEnd:=$vlStart+Length([Dictionary]Entry)
        HIGHLIGHT TEXT(vsComments;$vlEnd;$vlEnd)
    Else
            ` There is no corresponding entry in the Dictionary
        BEEP
    End if
Else
        ` There is no highlighted text
    BEEP
End if
```

The Get highlighted text method is listed here:

```
    ` Get highlighted text project method
    ` Get highlighted text ( String ; Long ; Long ) -> String
    ` Get highlighted text ( Text ; SelStart ; SelEnd ) -> highlighted text
C_TEXT($0;$1)
C_LONGINT($2;$3)
If ($2<$3)
    $0:=Substring($1;$2;$3-$2)
Else
    $0:=""
    $2:=$2-1
    Repeat
        If ($2>0)
            If (Position($1[[$2]];"   ,.!?:;()-_——")=0)
                $0:=$1[[$2]]+$0
                $2:=$2-1
            Else
                $2:=0
            End if
        End if
    Until ($2=0)
End if
```

See Also

Form event, Keystroke.

GOTO AREA

Entry Control

version 3

GOTO AREA (area)

Parameter	Type		Description
area	Field | Variable	→	Enterable field or variable to go to

Description

The command GOTO AREA is used to select the data entry area area as the active area of the form. It is equivalent to the user's clicking on or tabbing into the field or variable.

Note: This command has no effect on data entry areas located in subform List forms.

Example

See the example for the command REJECT.

See Also

REJECT.

REJECT

version 3

REJECT {(field)}

Parameter	Type		Description
field	Field	→	Field to reject

Description

REJECT has two forms. The first form has no parameters. It rejects the entire data entry and forces the user to stay in the form. The second form rejects only the field and forces the user to stay in the field.

Note: You should consider the built-in data validation tools before using this command.

The first form of REJECT prevents the user from accepting a record that is not complete. You can achieve the same result without using REJECT—you associate the Enter key with a No Action button and use the ACCEPT and CANCEL commands to accept or cancel the record, after the fields have been entered correctly. It is recommended that you use this second technique and do not use the first form of REJECT.

If you use the first form, you execute REJECT to prevent the user from accepting a record, usually because the record is not complete or has inaccurate entries. If the user tries to accept the record, executing REJECT prevents the record from being accepted; the record remains displayed in the form. The user must continue with data entry until the record is acceptable, or cancel the record.

The best place to put this form of REJECT is in the object method of an Accept button associated with the Enter key. This way, validation occurs only when the record is accepted, and the user cannot bypass the validation by pressing the Enter key.

The second form of REJECT is executed with the field parameter. The cursor stays in the field area. This form of REJECT forces the user to enter a correct value. It must be used immediately following a modification to the field. You can test for modification by using the Modified function. You can also use REJECT in the object method for the data entry area. This command has no effect on fields in subform areas.

You must put either form of the REJECT command in the form method or object method for the form that is being modified. If you are using REJECT for the subform's Detail Form for a table, put it in the form method or object method for the Detail Form.

You can use HIGHLIGHT TEXT to select the data in the field that is being rejected.

Examples

1. The following example is for a bank transaction record. It shows the first form of REJECT being used in an Accept button object method. The Enter key is set as an equivalent for the button. This means that even if the user presses the Enter key to accept the record, the button's object method will be executed. If the transaction is a check, then there must be a check number. If there is no check number, the validation is rejected:

```
    Case of
          ` If it is a check with no number...
        : (([Operation]Transaction="Check") & ([Operation]Check Number = ""))
          ALERT ("Please fill in the check number.")  ` Alert the user
⇒         REJECT  ` Reject the entry
          GOTO AREA ([Operation]Check Number)  ` Go to the check number field
    End case
```

2. The following example is part of an object method for an [Employees]Salary field. The object method tests the [Employees]Salary field and rejects the field if it is less than $10,000. You could perform the same operation by specifying a minimum value for the field in the form editor:

```
    If ([Employees]Salary<10000)
        ALERT ("Salary must be greater than $10,000")
⇒       REJECT ([Employees]Salary)
    End if
```

See Also

ACCEPT, CANCEL, GOTO AREA.

16 Form Events

Form event

Form event → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Form event number

Description

Form event returns a numeric value identifying the type of form event that has just occurred. Usually, you will use Form event from within a form or object method.

4th Dimension provides the following predefined constants:

Constant	Value	Description
On Load	1	The form is about to be displayed or printed
On Unload	24	The form is about to be exited and released
On Validate	3	The record data entry has been validated
On Clicked	4	A click occurred on an object
On Double Clicked	13	A double click occurred on an object
On Keystroke	17	A character is about to be entered in the object that has the focus
On Getting Focus	15	A form object is getting the focus
On Losing Focus	14	A form object is losing the focus
On Activate	11	The form's window becomes the frontmost window
On Deactivate	12	The form's window ceases to be the frontmost window
On Outside Call	10	The form received a CALL PROCESS call
On Drop	16	Data has been dropped onto an object
On Drag Over	21	Data could be dropped onto an object
On Menu Selected	18	A menu item has been chosen
On Data Change	20	Object Data has been modified
On External Area	19	An External object requested its object method to be executed
On Printing Header	5	The form's header area is about to be printed
On Printing Details	23	The form's details area is about to be printed
On Printing Break	6	One of the form's break areas is about to be printed
On Printing Footer	7	The form's footer area is about to be printed
On Close Box	22	The window's close box has been clicked
On Display Detail	8	A record is about to be displayed in a list
On Open Detail	25	A record is double clicked and you are going to the input form
On Close Detail	26	You left the input form and are going back to the output form

Events and Methods

When a form event occurs, 4th Dimension performs the following actions:

• First, it browses the objects of the form and calls the object method for any object (involved in the event) whose corresponding object event property has been selected.

• Second, it calls the form method if the corresponding form event property has been selected.

Do not assume that the object methods, if any, will be invoked in a particular order. The rule of thumb is that the object methods are always called before the form method. If an object is a subform, the object methods of the subform's list form are called, then the form method of the list form is called; 4D then continues to call the object methods of the parent form. In other words, when an object is a subform, 4D uses the same rule of thumb for the object and form methods within the subform object.

If the form event property is not selected for a given event, this does not prevent calls to the object methods for the objects whose same event property is selected. In other words, enabling or disabling an event at the form level has no effect on the object event properties.

The number of objects involved in an event depends on the nature of the event:

• On Load event - All the objects of the form (from any page) whose On Load object event property is selected will have their object method invoked. Then, if the On Load form event property is selected, the form will see its form method invoked.

• On Activate event - No object method will be invoked, because this event applies to the form as a whole and not to a particular object. Consequently, if the On Activate form event property is selected, only the form will see its form method invoked.

• On Drag Over event - Only the droppable object involved in the event will see its object method invoked if its On Drag Over object event property is selected. The form method will not be called.

WARNING: Contrary to all other events, during an On Drag over event, the object method for an object is executed in the context of the process of the drag and drop source object, not in the context of the process of the drag and drop destination object. For more information, see the commands DRAG AND DROP PROPERTIES and Drag and drop position.

The following table summarizes how object and form methods are called for each event type:

Event	Object Methods	Form Method	Which Objects
On Load	Yes	Yes	All objects
On Unload	Yes	Yes	All objects
On Validate	Yes	Yes	All objects
On Clicked	Yes (if clickable) (*)	Yes	Involved object only
On Double Clicked	Yes (if clickable) (*)	Yes	Involved object only
On Keystroke	Yes (if keyboard enter.) (*)	Yes	Involved object only
On Getting Focus	Yes (if tabbable) (*)	Yes	Involved object only
On losing Focus	Yes (if tabbable) (*)	Yes	Involved object only
On Activate	Never	Yes	None
On Deactivate	Never	Yes	None
On Outside Call	Never	Yes	None
On Drop	Yes (if droppable) (*)	Yes	Involved object only
On Drag Over	Yes (if droppable) (*)	Never	Involved object only
On Menu Selected	Never	Yes	None
On Data Change	Yes (if modifiable) (*)	Yes	Involved object only
On External Area	Yes	Yes	Involved object only
On Printing Header	Yes	Yes	All objects
On Printing Details	Yes	Yes	All objects
On Printing Break	Yes	Yes	All objects
On Printing Footer	Yes	Yes	All objects
On Close Box	Never	Yes	None
On Display Details	Yes	Yes	All objects
On Open Details	Never	Yes	None
On Close Details	Never	Yes	None

(*) For more infomation, see the section Events, Objects and Properties below.

IMPORTANT: Always keep in mind that, for any event, the method of a form or an object is called if the corresponding event property is selected for the form or objects. The benefit of disabling events in the Design environment (using the Form and Object Properties windows) is that you can greatly reduce the number of calls to methods and therefore significantly optimize the execution speed of your forms.

WARNING: The On Load and On Unload events are generated for objects if the events are enabled for both objects and the form to which belong. If the events are enabled for objects only, they will not occur; these two events must also be enabled at the form level.

Events, Objects and Properties

An object method is called if the event can actually occur for the object, depending on its nature and properties. The following section details the events you will generally use to handle the various types of objects.

Clickable Objects

Clickable objects are mainly handled using the mouse. They include:

• Boolean enterable fields or variables
• Picture fields or variables whose display format has been set to On Background
• Buttons, default buttons, radio buttons, check boxes, button grid
• 3DbButtons, 3D radio buttons, 3D check boxes
• Pop-up menus, hierarchical pop-up menus, picture menus
• Drop-down lists, menus/drop-down lists
• Scrollable areas, hierarchical lists
• Invisible buttons, highlight buttons, radio pictures
• Thermometers, rulers, dials (also known as slider objects)
• Tab controls

After the On Clicked and On Double Clicked object event properties are selected for one of these objects, you can detect and handle the clicks within or on the object, using the Form event command that returns On Clicked or On Double Clicked, depending on the case.

For all these objects, the On Clicked event occurs once the mouse button is released. However, there are two exceptions:
• Invisible buttons - The On Clicked event occurs as soon as the click is made and does not wait for the mouse button to be released.
• Slider objects (thermometers, rulers, and dials) - If the display format indicates that the object method must be called while you are sliding the control, the On Clicked event occurs as soon as the click is made.

Note: Some of these objects can be activated with the keyboard. For example, once a check box gets the focus, it can be entered using the space bar. In such a case, an On Clicked event is still generated.

WARNING: Combo boxes are not considered to be clickable objects. A combo box must be treated as an enterable text area whose associated drop-down list provides default values. Consequently, you handle the data entry within a combo box through the On Keystroke and On Data Change events.

Keyboard Enterable Objects

Keyboard enterable objects are objects into which you enter data using the keyboard and for which you may filter the data entry at the lowest level by detecting On Keystroke events. These include:

• All enterable field objects (except Picture, Subtable, and BLOB)
• All enterable variables (except Picture, BLOB, Pointer, and Array)
• Combo boxes
• Hierarchical lists

After the On Keystroke object event property is selected for one of these objects, you can detect and handle the keystrokes within the object, using the command Form event that will return On keystroke.

Modifiable Objects

Modifiable objects have a data source whose value can be changed using the mouse or the keyboard; they are not truly considered as user interface controls handled through the On Clicked event. They include:

• All enterable field objects (except Subtable and BLOB)
• All enterable variables (except BLOB, Pointer, and Array)
• Combo boxes
• External objects (for which full data entry is accepted by the 4D Extension)

These objects receive On Data Change events. After the On Data Change object event property is selected for one of these objects, you can detect and handle the change of the data source value, using the command Form event that will return On Data Change.

Tabbable Objects

Tabbable objects get the focus when you use the Tab key to reach them and/or click on them. The object having the focus receives the characters (typed on the keyboard) that are not accelerators (Windows) or shortcuts (MacOS) to a menu item or to an object such as a button.

All objects are tabbable, EXCEPT the following:

• Non enterable fields or variables
• Buttons (when used on MacOS)
• Button grid
• 3D buttons, 3D radio buttons, 3D check boxes
• Pop-up menus, hierarchical pop-up menus
• Menus/drop-down lists (when used on MacOS)
• Picture menus
• Scrollable areas
• Invisible buttons, highlight buttons, radio pictures
• Graphs
• External objects (for which full data entry is not accepted by the 4D Extension)
• Tab control

After the On Getting Focus and/or On losing Focus object event properties are selected for a tabbable object, you can detect and handle the change of focus, using the command Form event that will return On Getting Focus or On losing Focus, depending on the case.

Event Categories

Form events can be classified in the following categories:

- General events
  On Load, On Unload, On Validate, On Display Details
- Events proper to the form
  On Activate, On Deactivate, On Outside Call, On Close Box, On Menu Selected
  On Open Details, On Close Details
- Events related to user actions
  On Clicked, On Double Clicked, On Keystroke, On Getting Focus, On losing Focus
  On Data Change, On External Area
- Drag and drop events
  On Drop, On Drag Over
- Printing Events
  On Printing Header, On Printing Details, On Printing Break, On Printing Footer

Compatibility between V6 and V3

The following table summarizes the equivalence between V6 form events and V3 layout execution cycles.

V6 Events	V3 Layout Execution cycles	V3 command
On Load	Before phase	Before
On Unload	No equivalent execution cycle	None
On Validate	After phase	After
On Clicked	Generic During phase	During
On Double Clicked	Generic During phase	During
On Keystroke	No equivalent execution cycle	None
On Getting Focus	No equivalent execution cycle	None
On losing Focus	No equivalent execution cycle	None
On Activate	Activated phase	Activated
On Deactivate	Deactivated phase	Deactivated
On Outside Call	Outside call phase	Outside call
On Drop	No equivalent execution cycle	None
On Drag Over	No equivalent execution cycle	None
On Menu Selected	Generic During phase	During plus Menu selected
On Data Change	Generic During phase	During
On External Area	Generic During phase	During
On Printing Header	Printing header phase	In header
On Printing Details	Generic During phase	During
On Printing Break	Printing break phase	In break
On Printing Footer	Printing footer phase	In footer
On Close Box	No equivalent execution cycle	OPEN WINDOW (with Close box)
On Display Details	Before and During phase	Before & During
On Open Details	Generic During phase	During
On Close Details	Generic During phase	During

When you open a V3 database using 4th Dimension V6, the program performs two operations:
• Converts the structure file to the new format.
• Converts the data file to the new format.

When using the database after the conversion, if a form has not been edited or modified in the Design environment, it is still stored in the structure file in the way it was with V3. In order to insure compatibility with your existing V3 applications, the form and object event properties are automatically set to reflect the settings "ala V3". This means that the V6 event properties will be automatically selected and the "old V3 commands" will act as they did with version 3:

V6 Events	V3 Layout Execution cycles	V3 command
On Load	Before phase	Before
On Validate	After phase	After
On Clicked	Generic During phase	During
On Double Clicked	Generic During phase	During
On Activate	Activated phase	Activated
On Deactivate	Deactivated phase	Deactivated
On Outside Call	Outside call phase	Outside call
On Menu Selected	Generic During phase	During plus Menu selected
On Data Change	Generic During phase	During
On External Area	Generic During phase	During
On Printing Header	Printing header phase	In header
On Printing Details	Generic During phase	During
On Printing Break	Printing break phase	In break
On Printing Footer	Printing footer phase	In footer
On Display Details	Before and During phase	Before & During
On Open Details	Generic During phase	During
On Close Details	Generic During phase	During

If an object (field or variable) has the V3 Script only if modified option selected, the event properties are reduced to those corresponding to any During execution cycle that could occur during data entry in V3:

V6 Events	V3 Layout Execution cycles	V3 command
On Clicked	Generic During phase	During
On Double Clicked	Generic During phase	During
On Data Change	Generic During phase	During
On External Area	Generic During phase	During

Once you start editing a form and its objects in V6, the form and object event properties are, by default, set according to the same scheme. To take advantage of the new events introduced by V6, select the event properties for the form and objects in the Design environment, and modify the form and object methods using the new Form event command.

The new events without corresponding V3 execution cycle are:

V6 Events	V3 Layout Execution cycles	V3 command
On Unload	No equivalent execution cycle	None
On Keystroke	No equivalent execution cycle	None
On Getting Focus	No equivalent execution cycle	None
On losing Focus	No equivalent execution cycle	None
On Drop	No equivalent execution cycle	None
On Drag Over	No equivalent execution cycle	None
On Close Box	No equivalent execution cycle	OPEN WINDOW (with Close box)

The new events that allow you to perform actions better tuned to the nature of the events are:

V6 Events	V3 Layout Execution cycles	V3 command
On Clicked	Generic During phase	During
On Double Clicked	Generic During phase	During
On Menu Selected	Generic During phase	During plus Menu selected
On Data Change	Generic During phase	During
On External Area	Generic During phase	During
On Printing Details	Generic During phase	During
On Display Details	Before and During phase	Before & During
On Open Details	Generic During phase	During
On Close Details	Generic During phase	During

Examples

In all the examples discussed here, it is assumed that the event properties of the forms and objects have been selected appropriately.

1. This example sorts a selection of subrecords for the subtable [Parents]Children before a form for the table [Parents] is displayed on the screen:

```
          ` Method of a form for the table [Parents]
       Case of
⇒         : (Form event=On Load)
             ORDER SUBRECORDS BY([Parents]Children;[Parents]Children'First name;>)
             ` ...
       End case
```

2. This example shows the On Validate event being used to automatically assign (to a field) the date that the record is modified:

```
          ` Method of a form
       Case of
             ` ...
⇒         : (Form event=On Validate)
             [aTable]Last Modified On:=Current date
       End case
```

3. In this example, the complete handling of a drop-down list (initialization, user clicks, and object release) is encapsulated in the method of the object:

```
          ` asBurgerSize Drop-down list Object Method
      Case of
⇒         : (Form event=On Load)
             ARRAY STRING(31;asBurgerSize;3)
             asBurgerSize{1}:="Small"
             asBurgerSize{1}:="Medium"
             asBurgerSize{1}:="Large"
⇒         : (Form event=On Clicked)
             If (asBurgerSize#0)
                ALERT("You chose a "+asBurgerSize{asBurgerSize}+" burger.")
             End if
⇒         : (Form event=On Unload)
             CLEAR VARIABLE(asBurgerSize)
      End case
```

4. This example shows how, in an object method, to accept and later handle a drag and drop operation for a field object that only accepts picture values.

```
          ` [aTable]aPicture enterable picture field object method
      Case of
⇒         : (Form event=On Drag Over)
                ` A drag and drop operation has started
                ` and the mouse is currently over the field
                ` Get the information about the source object
             DRAG AND DROP PROPERTIES ($vpSrcObject;$vlSrcElement;$lSrcProcess)
                ` Note that we do not need to test the source process ID number
                ` for the object method is exceptionally here executed
                ` in the context of that process
             $vlDataType:=Type ($vpSrcObject->)
                ` Is the source data a picture (field, variable or array) ?
             If (($vlDataType=Is Picture) | ($vlDataType=Picture Array))
                   ` If so, accept the drag.
                   ` Note that the mouse button is still pressed, the only effect while
                   ` accepting the drag is to let 4D highlighting the object so the user
                   ` knows the source data could be dropped onto that object
                $0:=0
             Else
                   ` If so, refuse the drag
                $0:=-1
                   ` In this case, the object is not highlighted
             End if
```

```
⇒         : (Form event=On Drop)
                ` The source data has been dropped on the object,
                ` we therefore need to copy it into the object
                ` Get the information about the source object
            DRAG AND DROP PROPERTIES ($vpSrcObject;$vlSrcElement;$lSrcProcess)
            $vlDataType:=Type ($vpSrcObject->)
            Case of
                    ` The source object is Picture field or variable
                : ($vlDataType=Is Picture)
                        ` Is the source object from the same process
                        ` (thus from the same window and form)?
                    If ($lSrcProcess=Current process)
                            ` If so, just copy the source value
                        [aTable]aPicture:=$vpSrcObject->
                    Else
                            ` If not, is the source object a variable?
                        If (Is a variable ($vpSrcObject))
                                ` If so, get the value from the source process
                            GET PROCESS VARIABLE ($lSrcProcess;
                                                        $vpSrcObject->;$vgDraggedPict)
                            [aTable]aPicture:=$vgDraggedPict
                        Else
                            ` If not, use CALL PROCESS to get the field value
                            ` from the source process
                        End if
                    End if
                    ` The source object is an array of pictures
                : ($vlDataType=Picture Array)
                        ` Is the source object from the same process
                        ` (thus from the same window and form)?
                    If ($lSrcProcess=Current process)
                            ` If so, just copy the source value
                        [aTable]aPicture:=$vpSrcObject->{$vlSrcElement}
                    Else
                            ` If not, get the value from the source process
                        GET PROCESS VARIABLE ($lSrcProcess;$vpSrcObject
                                ->{$vlSrcElement};$vgDraggedPict)
                        [aTable]aPicture:=$vgDraggedPict
                    End if
            End case
        End case
```

Note: For other examples showing how to handle On Drag Over and On Drop events, see the examples of the command DRAG AND DROP PROPERTIES.

5. This example is a template for a form method. It shows each of the possible events that can occur while a summary report uses a form as an output form:

```
          ` Method of a form being used as output form for a summary report
      $vpFormTable:=Current form table
      Case of
            ` ...
⇒        : (Form event=On Printing Header)
                ` A header area is about to be printed
            Case of
                : (Before selection($vpFormTable->))
                    ` Code for the first break header goes here
                : (Level = 1)
                    ` Code for a break header level 1 goes here
                : (Level = 2)
                    ` Code for a break header level 2 goes here
                    ` ...
            End case
⇒        : (Form event=On Printing Details)
                ` A record is about to be printed
                ` Code for each record goes here
⇒        : (Form event=On Printing Break)
                ` A break area is about to be printed
            Case of
                : (Level = 0)
                    ` Code for a break level 0 goes here
                : (Level = 1)
                    ` Code for a break level 1 goes here
                    ` ...
            End case
⇒        : (Form event=On Printing Footer)
            If(End selection($vpFormTable->))
                ` Code for the last footer goes here
            Else
                ` Code for a footer goes here
            End if
      End case
```

6. This example shows the template of a form method that handles the events that can occur for a form displayed using the commands DISPLAY SELECTION or MODIFY SELECTION. For didactic purposes, it displays the nature of the event in the title bar of the form window.

```
          ` A Form method
      Case of
⇒        : (Form event=On Load)
            $vsTheEvent:="The form is about to be displayed"
```

```
⇒          : (Form event=On Unload)
              $vsTheEvent:="The output form has been exited and
                                                  is about to disappear from the screen"
⇒          : (Form event=On Display Details)
              $vsTheEvent:="Displaying record #"
                                             +String(Selected record number([TheTable]))
⇒          : (Form event=On Menu Selected)
              $vsTheEvent:="A menu item has been selected"
⇒          : (Form event=On Printing Header")
              $vsTheEvent:="The header area is about to be drawn"
⇒          : (Form event=On Open Details)
              $vsTheEvent:="The record #"
                              +String(Selected record number([TheTable]))+" is double-clicked"
⇒          : (Form event=On Close Details)
              $vsTheEvent:="Going back to the output form"
⇒          : (Form event=On Activate)
              $vsTheEvent:="The form's window just become the frontmost window"
⇒          : (Form event=On Deactivate)
              $vsTheEvent:="The form's window is no longer the frontmost window"
⇒          : (Form event=On Menu Selected)
              $vsTheEvent:="A menu item has been choosen"
⇒          : (Form event=On Outside call)
              $vsTheEvent:="A call from another has been received"
           Else
⇒             $vsTheEvent:="What's going on? Event #"+String(Form event)
        End case
        SET WINDOW TITLE ($vsTheEvent)
```

7. For examples on how to handle On Keystroke events, see examples for the commands Keystroke and FILTER KEYSTROKE.

9. This example shows how to treat clicks and double clicks in the same way as a scrollable area:

```
           ` asChoices scrollable area object method
        Case of
⇒          : (Form event=On Load)
              ARRAY STRING (...;asChoices;...)
              asChoices:=0
⇒          : ((Form event=On Clicked) | (Form event=On Double Clicked))
              If (asChoices#0)
                    ` An item has been clicked, do something here
              End if
                  ` ...
        End case
```

10. This example shows how to treat clicks and double clicks using a different response. Note the use of the element zero for keeping track of the selected element:

```
          ` asChoices scrollable area object method
      Case of
⇒         : (Form event=On Load)
             ARRAY STRING (...;asChoices;...)
                ` ...
             asChoices:=0
             asChoices{0}:="0"
⇒         : (Form event=On Clicked)
             If (asChoices#0)
                If (asChoices#Num(asChoices))
                      ` A new item has been clicked, do something here
                      ` ...
                      ` Save the new selected element for the next time
                   asChoices{0}:=String (asChoices)
                End if
             Else
                asChoices:=Num(asChoices{0})
             End if
⇒         : (Form event=On Double Clicked)
             If (asChoices#0)
                   ` An item has been double clicked, do something different here
             End if
                ` ...
      End case
```

11. This example shows how to maintain a status text information area from within a form method, using the On Getting Focus and On losing Focus events:

```
          ` [Contacts];"Data Entry" form method
      Case of
⇒         : (Form Event=On Load)
             C_TEXT(vtStatusArea)
             vtStatusArea:=""
⇒         : (Form Event=On Getting Focus)
             RESOLVE POINTER (Last object;$vsVarName;$vlTableNum;$vlFieldNum)
             If (($vlTableNum#0) & ($vlFieldNum#0))
                Case of
                   : ($vlFieldNum=1) ` Last name field
                      vtStatusArea:="Enter the Last name of the Contact,
                                                         it will be automatically capitalized"
                         ` ...
```

```
                    : ($vlFieldNum=10) ` Zip Code field
                        vtStatusArea:="Enter a 5-digit zip code,
                                                       it will be automatically checked and validated"
                            ` ...
                End case
            End if
⇒        : (Form Event=On Losing Focus)
            vtStatusArea:=""
                ` ...
    End case
```

12. This example shows how to respond to a close window event with a form used for record data entry:

```
        ` Method for a data entry form
    $vpFormTable:=Current form table
    Case of
            ` ...
⇒        : (Form Event=On Close Box)
            If (Modified record($vpFormTable->))
                CONFIRM ("This record has been modified. Save Changes?")
                If (OK=1)
                    ACCEPT
                Else
                    CANCEL
                End if
            Else
                CANCEL
            End if
                ` ...
    End case
```

13. This example shows how to capitalize a text or alphanumeric field each time its data source value is modified:

```
        ` [Contacts]First Name Object method
    Case of
            ` ...
        : (Form event=On Data Change)
            [Contacts]First Name:= Uppercase(Substring([Contacts]First Name;1;1))
                        +Lowercase(Substring([Contacts]First Name;2))
                ` ...
    End case
```

See Also

CALL PROCESS, Current form table, DRAG AND DROP PROPERTIES, FILTER KEYSTROKE, Keystroke.

Before

version 3

Compatibility Note

This command has been kept in 4D for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Load event.

Before → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

In order for the Before execution cycle to be generated, make sure that the On Load event property for the form and/or the objecs has been selected in the Design environment.

See Also

Form event.

During

Form Events

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an event such as On Clicked.

During → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

In order for the During execution cycle to be generated, make sure that the appropriate event properties, such as On Clicked, for the form and/or the objects have been selected in the Design environment.

See Also

Form event.

After

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Validate event.

After → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

In order for the After execution cycle to be generated, make sure that the On Validate event property for the form and/or the objects has been selected in the Design environment.

See Also

Form event.

In header

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Printing Header event.

In header → Boolean

Parameter	Type	Description

This command does not require any parameters

Description

In order for the In header execution cycle to be generated, make sure that the On Printing Header event property for the form and/or the objects has been selected in the Design environment.

See Also

During, In break, In footer.

In break

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Printing Break event.

In break → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

In order for the In break execution cycle to be generated, make sure that the On Printing Break event property for the form and/or the objects has been selected in the Design environment.

See Also

During, In footer, In header.

In footer

version 3

Compatibility Note

This command has been kept for compatibility reason. Starting with version 6, you may want to start using the command Form event and check if it returns an On Printing Footer event.

In footer → Boolean

Parameter	Type	Description

This command does not require any parameters

Description

In order for the In footer execution cycle to be generated, make sure that the On Printing footer event property for the form and/or the objects has been selected in the Design environment.

See Also

During, In break, In header.

Activated

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Activate event.

Activated → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	Returns TRUE if the execution cycle is an activation

Description

The command Activated returns TRUE in a form method when the window containing the form becomes the frontmost window of the frontmost process.

WARNING: Do not place a command such as TRACE or ALERT in the Activated phase of the form, as this will cause an endless loop.

Note: In order for the Activated execution cycle to be generated (for compatibility with V3 databases), make sure that the On Activate event property of the form has been selected in the Design environment. This is done automatically when a database is converted.

See Also

Deactivated, Form event.

Deactivated

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Deactivate event.

Deactivated → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	Returns TRUE if the execution cycle is a deactivation

Description

The command Deactivated returns TRUE in a form or object method when the frontmost window of the frontmost process, containing the form, moves to the back.

In order for the Deactivated execution cycle to be generated, make sure that the On Deactivate event property of the form and/or the objects has been selected in Design environment.

See Also

Activated, Form event.

Outside call

version 3

Compatibility Note

This command has been kept for compatibility reasons. Starting with version 6, you should consider using the command Form event and checking if it returns an On Outside call event.

Outside call → Boolean

Parameter	Type	Description
This command does not require any parameters		

Description

In order for the Outside call execution cycle to be generated, make sure that the On Outside call event property for the form and/or the objects has been selected in the Design environment.

See Also

CALL PROCESS, Form event.

17 Form Pages

Form Pages

Form Pages

version 6.0

The commands in this section enable you to manipulate form pages.

Automatic action buttons perform the same tasks as the FIRST PAGE, LAST PAGE, NEXT PAGE, and PREVIOUS PAGE commands. In addition, version 6 introduces a new automatic action equivalent to GOTO PAGE that you can apply to objects such as tab controls, drop-down list boxes, and so on. Whenever appropriate, use automatic action buttons instead of commands.

Page commands can be used with input forms or with forms displayed in dialogs. Output forms use only the first page. A form always has at least one page—the first page. Remember that regardless of the number of pages a form has, only one form method exists for each form.

Use the Current form page command to find out which page is being displayed.

Note: When **designing** a form, you can work with pages 1 through N, as well as with page 0, in which you put objects that will appear in all of the pages. When **using** a form, and therefore when calling the form pages commands, you work with pages 1 through N; page 0 is automatically combined with the page being displayed.

FIRST PAGE

FIRST PAGE

Parameter	Type	Description
This command does not require any parameters		

Description

FIRST PAGE changes the currently displayed form page to the first form page. If a form is not being displayed, or if the first form page is already displayed, FIRST PAGE does nothing.

Example

The following example is a one-line method called from a menu command. It displays the first form page:

```
⇒	FIRST PAGE
```

See Also

Current form page, GOTO PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE.

LAST PAGE

LAST PAGE

Parameter	Type	Description
This command does not require any parameters		

Description

LAST PAGE changes the currently displayed form page to the last form page. If a form is not being displayed, or if the last form page is already displayed, LAST PAGE does nothing.

Example

The following example is a one-line method called from a menu command. It displays the last form page:

```
⇒	LAST PAGE
```

See Also

Current form page, FIRST PAGE, GOTO PAGE, NEXT PAGE, PREVIOUS PAGE.

NEXT PAGE

NEXT PAGE

Parameter	Type	Description

This command does not require any parameters

Description

NEXT PAGE changes the currently displayed form page to the next form page. If a form is not being displayed, or if the last form page is already displayed, NEXT PAGE does nothing.

Example

The following example is a one-line method called from a menu command. It displays the form page that follows the one currently displayed:

```
⇒	NEXT PAGE
```

See Also

Current form page, FIRST PAGE, GOTO PAGE, LAST PAGE, PREVIOUS PAGE.

PREVIOUS PAGE

Form Pages

version 3

PREVIOUS PAGE

Parameter	Type	Description
This command does not require any parameters		

Description

PREVIOUS PAGE changes the currently displayed form page to the previous form page. If a form is not being displayed, or if the first form page is already displayed, PREVIOUS PAGE does nothing.

Example

The following example is a one-line method called from a menu command. It displays the form page that precedes the one currently displayed:

```
⇒	PREVIOUS PAGE
```

See Also

Current form page, FIRST PAGE, GOTO PAGE, LAST PAGE, NEXT PAGE.

GOTO PAGE

GOTO PAGE (pageNumber)

Parameter	Type		Description
pageNumber	Number	→	Form page to display

Description

GOTO PAGE changes the currently displayed form page to the form page specified by pageNumber.

If a form is not being displayed, GOTO PAGE does nothing. If pageNumber is greater than the number of pages, the last page is displayed. If pageNumber is less than one, the first page is displayed.

Examples

The following example is an object method for a button. It displays a specific page, page 3:

```
⇒	GOTO PAGE (3)
```

See Also

Current form page, FIRST PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE.

Current form page

Current form page → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of currently displayed form page

Description

The Current form page command returns the number of the currently displayed form page.

Example

In a form, when you select a menu item from the menu bar or when the form receives a call from another process, you can perform different actions depending on the form page currently displayed. In this example, you write:

```
     ` [myTable];"myForm" Form Method
   Case of
      : (Form event=On Load)
         ` ...
      : (Form event=On Unload)
         ` ...
      : (Form event=On Menu selected)
         $vlMenuNumber:=Menu Selected >> 16
         $vlItemNumber:=Menu Selected & 0xFFFF
         Case of
            : ($vlMenuNumber=...)
               Case of
                  : ($vlItemNumber=...)
⇒                       : (Current form page=1)
                           ` Do appropriate action for page 1
⇒                       : (Current form page=2)
                           ` Do appropriate action for page 2
                           ` ...
                  : ($vlItemNumber=...)
                           ` ...
               End case
            : ($vlMenuNumber=...)
               ` ...
         End case
```

```
          : (Form event=On Outside call)
               Case of
⇒                   : (Current form page=1)
                         ` Do appropriate reply for page 1
⇒                   : (Current form page=2)
                         ` Do appropriate reply for page 2
               End case
                    ` ...
     End case
```

See Also

FIRST PAGE, GOTO PAGE, LAST PAGE, NEXT PAGE, PREVIOUS PAGE.

18 Graphs

GRAPH

version 6.0 (Modified)

Version 6 Note: Starting with version 6, graphs are now supported by the 4D Chart Plug-in, which is integrated within 4th Dimension. The Graph commands from the previous version of 4D are transparently redirected to 4D Chart. In addition, to use the additional 4D Chart commands for customizing a Graph Area located in a form, use the graphArea parameter (described in this command) as an external area reference for the 4D Chart commands. For detailed information about the 4D Chart commands, refer to the *4D Chart Reference* manual.

GRAPH (graphArea; graphNumber; xLabels; yElements{; yElements2; ...; yElementsN})

Parameter	**Type**		**Description**
graphArea	Variable	→	Graph area in the form
graphNumber	Number	→	Graph type number
xLabels	Array or Subfields	→	Labels for the x-axis
yElements	Array or Subfields	→	Data to graph (up to eight allowed)

Description

GRAPH draws a graph for a Graph area located in a form. The data can come from either arrays or subfields.

The graphArea parameter is the name of the Graph area that displays the graph. The Graph area is created in the Form editor, using the graph object type. The graph name is the name entered for the variable name. For information about creating a Graph area, see the *4th Dimension Design Reference.*

The graphNum parameter defines the type of graph that will be drawn. It must be a number from 1 to 8. The graph types are described in Example 1. After a graph has been drawn, you can change the type by changing graphNum and executing the GRAPH command again.

The xLabels parameter defines the labels that will be used to label the x-axis (the bottom of the graph). This data can be of string, date, time, or numeric type. There should be the same number of subrecords or array elements in xLabels as there are subrecords or array elements in each of the yElements.

The data specified by yElements is the data to graph. The data must be numeric. Up to eight data sets can be graphed. Pie charts graph only the first yElements.

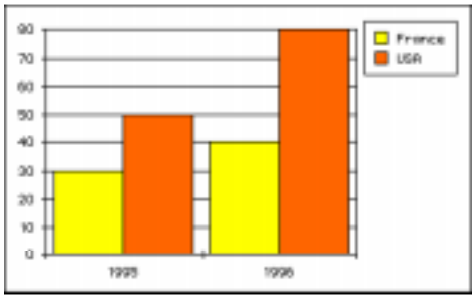
Examples

1. The following example shows how to use arrays to create a graph. The code would be inserted in a form method or object method. It is not intended to be realistic, since the data is constant:

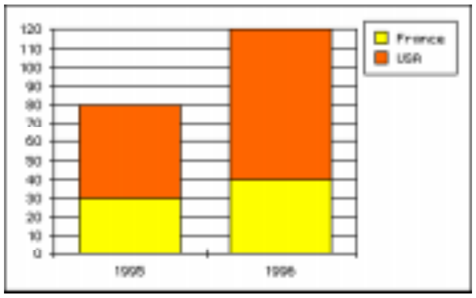
```
    ARRAY STRING (4; X; 2)  ` Create an array for the x-axis
    X{1}:="1995"  ` X Label #1
    X{2}:="1996"  ` X Label #2
    ARRAY REAL (A; 2)  ` Create an array for the y-axis
    A{1}:=30  ` Insert some data
    A{2}:=40
    ARRAY REAL (B; 2) ` Create an array for the y-axis
    B{1}:=50  ` Insert some data
    B{2}:=80
⇒   GRAPH (vGraph;vType; X; A; B) ` Draw the graph
    GRAPH SETTINGS (vGraph;0;0;0;0;False;False;True;"France";"USA") ` Set the legends
for the graph
```

The following figure shows the resulting graph.

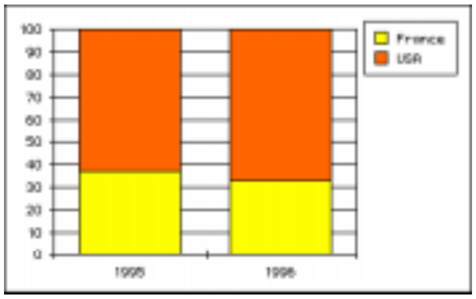
• With vType equal to 1, you obtain a **Column** graph:



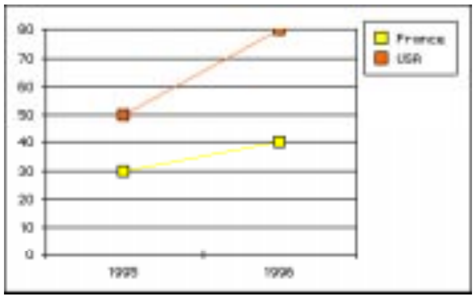
• With vType equal to 2, you obtain a **Proportional Column** graph:



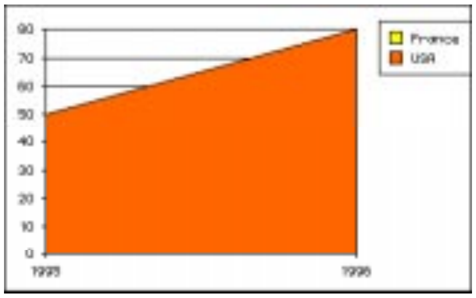
• With vType equal to 3, you obtain a **Stacked Column** graph:



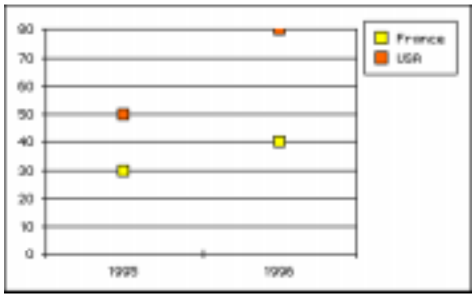
• With vType equal to 4, you obtain a **Line** graph:



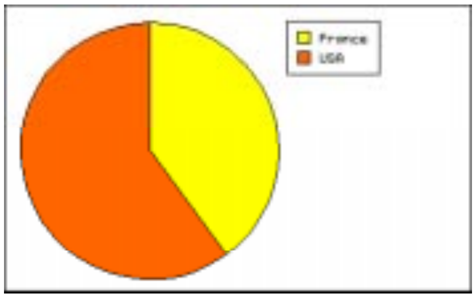
• With vType equal to 5, you obtain a **Area** graph:



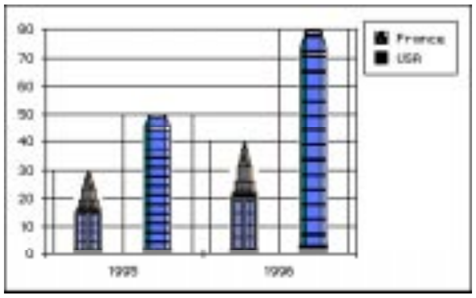
• With vType equal to 6, you obtain a **Scatter** graph:



• With vType equal to 7, you obtain a **Pie** graph:



• With vType equal to 8, you obtain a **Picture** graph:



2. The following example graphs the sales in dollars for sales people in a subtable. The subtable has three fields: Name, LastYearTot, and ThisYearTot. The graph will show the sales for each of the sales people for the last two years:

```
⇒      GRAPH (vGraph;1;[Employees]Sales'Name;
                     [Employees]Sales'LastYearTot;[Employees]Sales'ThisYearTot)
```

See Also

GRAPH SETTINGS, GRAPH TABLE.

GRAPH SETTINGS

version 6.0 (Modified)

Version 6 Note: Starting with version 6, graphs are now supported by the 4D Chart Plug-in, which is integrated within 4th Dimension. The Graph commands from the previous version of 4D are transparently redirected to 4D Chart. In addition, to use the additional 4D Chart commands for customizing a Graph Area located in a form, use the graph parameter (described in this command) as an external area reference for the 4D Chart commands. For detailed information about the 4D Chart commands, refer to the *4D Chart Reference* manual.

GRAPH SETTINGS (graph; xmin; xmax; ymin; ymax; xprop; xgrid; ygrid; title{; title2; ...; titleN})

Parameter	**Type**		**Description**
graph	Variable	→	Name of the Graph area
xmin	Number or date or time	→	Minimum x-axis value for proportional graph (line or scatter plot only)
xmax	Number or date or time	→	Maximum x-axis value for proportional graph (line or scatter plot only)
ymin	Number	→	Minimum y-axis value
ymax	Number	→	Maximum y-axis value
xprop	Boolean	→	TRUE for proportional x-axis; FALSE for normal x-axis (line or scatter plot only)
xgrid	Boolean	→	TRUE for x-axis grid; FALSE for no x-axis grid (only if xprop is TRUE)
ygrid	Boolean	→	TRUE for y-axis grid; FALSE for no y-axis grid
title	String	→	Title(s) for graph legend(s)

Description

GRAPH SETTINGS changes the graph settings for graph displayed in a form. The graph must have already been displayed with the GRAPH command. GRAPH SETTINGS has no effect on a pie chart.

The xmin, xmax, ymin, and ymax parameters all set the minimum and maximum values for their respective axes of the graph. If the value of any pair of these parameters is a null value (0, ?00:00:00?, or !00/00/00!, depending on the data type), the default graph values will be used.

The xprop parameter turns on proportional plotting for line graphs (type 4) and scatter graphs (type 6). When TRUE, it will plot each point on the x-axis according to the point's value, and then only if the values are numeric, time, or date.

The xgrid and ygrid parameters display or hide grid lines. A grid for the x-axis will be displayed only when the plot is a proportional scatter or line graph.

The title parameter(s) labels the legend.

Compatibility Note (March 97): The parameters xmin, xmax and xprop are not currently supported. They will be supported in a future update of 4D Chart.

Example

See example for the command GRAPH.

See Also

GRAPH, GRAPH TABLE.

GRAPH TABLE

version 6.0 (Modified)

Version 6 Note: Starting with version 6, graphs are now supported by the 4D Chart Plug-in, which is integrated within 4th Dimension. The Graph commands from the previous version of 4D are transparently redirected to 4D Chart. For detailed information about the 4D Chart commands, refer to the *4D Chart Reference* manual.

GRAPH TABLE {(table)}

or:

GRAPH TABLE ({table; }graphType; x field; y field{; y field2; ...; y fieldN})

Parameter	**Type**		**Description**
table	Table	→	Table to graph, or Default table, if omitted
graphType	Number	→	Graph type number
x field	Field	→	Labels for the x-axis
y field	Field	→	Fields to graph (up to eight allowed)

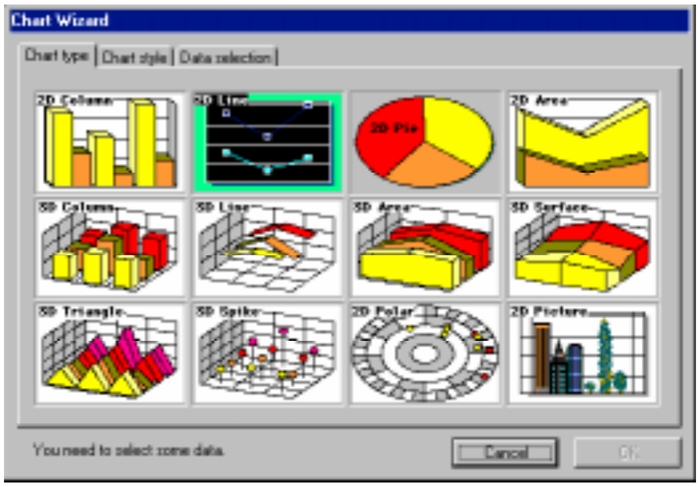
Description

GRAPH TABLE has two forms. The first form displays the Chart Wizard and allows the user to select the fields to be graphed. The second form specifies the fields to be graphed and does not display the Chart Wizard.

GRAPH TABLE graphs data from a table's fields. Only data from the current selection of the current process is graphed.

Using the first form is equivalent to choosing Graph from the Report menu in the User environment.

The following figure shows the Chart Wizard, which allows the user to define the graph.



You need to select some data.

The second form of the command graphs the fields specified for table.

The graphType parameter defines the type of graph that will be drawn. It must be a number from 1 to 8. See the graph types listed in the example for the command Graph.

The x field defines the labels that will be used to label the x-axis (the bottom of the graph). The field type can be Alpha, Integer, Long integer, Real or Date.

The y field is the data to graph. The field type must be Integer, Long integer or Real. Up to eight y fields can be graphed, each set off by a semicolon.

In either form, GRAPH TABLE opens a Chart window for working with the newly created graph. For more information about using the Chart window, see the *4th Dimension User Reference* manual.

Note: You can also use the Quick Report editor to generate graphs from field data, by using the Print Destination menu.

Examples

1. The following example illustrates the use of the first form of GRAPH TABLE. It presents the Chart Wizard window and allows users to select the fields to graph. The code queries records in the [People] table, sorts them, and then displays the Chart Wizard:

```
      QUERY ([People])
      If (OK=1)
         ORDER BY ([People])
         If (OK=1)
⇒           GRAPH TABLE([People])
         End if
      End if
```

2. The following example illustrates the use of the second form of GRAPH TABLE. It first queries and orders records from the [People] table. It then graphs the salaries of the people:

```
      QUERY([People];[People]Title="Manager")
      ORDER BY([People];[People]Salary;>)
⇒     GRAPH TABLE([People];1;[People]Last Name;[People]Salary)
```

See Also

Graph.

19 Hierarchical Lists

Load list

version 6.0

Load list (listName) → ListRef

Parameter	Type		Description
listName	String	→	Name of a list created in the Design environment List Editor
Function result	ListRef	←	List reference number of newly created list

Description

The command Load list creates a new hierarchical list whose contents are copied from the list and whose name you pass in listName. It then returns the list reference number to the newly created list.

If the list specified by listName does not exist, the list is not created and Load list returns zero (0).

Note that the new list is a copy of the list defined in the Design environment. Consequently, any modifications made to the new list will not affect the list defined in the Design environment. Conversely, any subsequent modifications made to the list defined in the Design environment will not affect the list that you just created.

If you modify the newly created list and want to permanently save the changes, call SAVE LIST.

Remember to call CLEAR LIST in order to dispose of the newly created list when you have finished with it. Otherwise, it will stay in memory until the end of the working session or until the process in which it was created ends or is aborted.

Tip: If you associate a list to a form object (hierarchical list, tab control, or hierarchical pop-up menu) using the Choice List property within the Form Editor Object Properties window, you do not need to call Load list or CLEAR LIST from the method of the object. 4th Dimension loads and clears the list automatically for you.

Example

You create a database for the international market and you need to switch to different languages while using the database. In a form, you present a hierarchical list, named hlList, that proposes a list of standard options. In the Design environment, you have prepared various lists, such as “Std Options US” for the English version, “Std Options FR” for the French version, “Std Options SP” for the Spanish version, and so on. In addition, you maintain an interprocess variable, named <>gsCurrentLanguage, where you store a 2-character language code, such as “US” for the English version, “FR” for the French version, “SP” for the Spanish version, and so on. To make sure that your list will always be loaded using the current selected language, you can write:

```
          ` hlList Hierarchical List Object Method
      Case of
          : (Form event = On Load)
             C_LONGINT (hlList)
⇒            hlList:=Load list("Std Options"+<>gsCurrentLanguage)
          : (Form event = On Unload)
             CLEAR LIST(hlList;*)
      End case
```

See Also

CLEAR LIST, SAVE LIST.

SAVE LIST

version 6.0

SAVE LIST (list; listName)

Parameter	Type		Description
list	ListRef	→	List reference number
listName	String	→	Name of the list as it will appear in the Design environment List Editor

Description

The command SAVE LIST saves the list whose reference number you pass in list, within the Design environment List Editor, under the name you pass in listName.

If there is already a list with this name, its contents are replaced.

See Also

Load list.

New list

version 6.0

New list → ListRef

Parameter	Type		Description
This command does not require any parameters			
Function result	ListRef	←	List reference number

Description

New list creates a new, empty hierarchical list in memory and returns its unique list reference number.

WARNING: Hierarchical lists are held in memory. When you are finished with a hierarchical list, it is important to dispose of it and free the memory, using the command CLEAR LIST.

Several other commands allow you to create hierarchical lists:
• Copy list duplicates a list from an existing list.
• Load list creates a list by loading a Choice List created (manually or programmatically) in the Design enviornment List Editor.
• BLOB to list creates a list from the contents of a BLOB in which a list was previously saved.

After you have created a hierarchical list using New list, you can:
• Add items to that list, using the command APPEND LIST ITEM or INSERT LIST ITEM.
• Delete items from that list, using the command DELETE LIST ITEM.

Example

See example for the command APPEND TO TO LIST.

See Also

APPEND TO LIST, BLOB to list, CLEAR LIST, Copy list, DELETE LIST ITEM, INSERT LIST ITEM, Load list.

Copy list

version 6.0

Copy list (list) → ListRef

Parameter	Type		Description
list	ListRef	→	Reference to list to be copied
Function result	ListRef	←	List reference number to duplicated list

Description

The command Copy list duplicates the list whose reference number you pass in list, and returns the list reference number of the new list.

After you have finished with the new list, call CLEAR LIST to delete it.

See Also

CLEAR LIST, Load list, New list.

CLEAR LIST

version 6.0

CLEAR LIST (list{; *})

Parameter	Type		Description
list	ListRef	→	List reference number
*		→	If specified, clear sublists from memory, if any If omitted, sublists, if any, are not cleared

Description

The command CLEAR LIST disposes of the hierarchical list whose list reference number you pass in list.

Usually you will pass the optional * parameter, so all the sublists, if any, attached to items or subitems of the list will be disposed of too.

You do not need to clear a list attached to a form object via the Object Properties window. 4D loads and clears the list for you. On the other hand, each time you load, copy, extract from a BLOB, or create a list programmatically, call CLEAR LIST when you are through with the list.

To clear a sublist attached to an item (of any level) of another list currently displayed in a form, proceed as follows:
1. Call GET LIST ITEM on the parent item to get the list reference of the sublist.
2. Call SET LIST ITEM on the parent item to detach the sublist from the list item before clearing it.
3. Call CLEAR LIST to clear the sublist whose reference number you obtained with GET LIST ITEM.
4. Call REDRAW LIST for the list displayed in the form, to recalculate its items and sublists.

Examples

1. Within a clean-up routine that clears all objects and data that you no longer need (i.e., when a window is closed and a form unloaded), you may end up clearing a hierarchical list that may have already been cleared, depending on the user actions within the form. Use Is a list to clear the list only if necessary:

```
          ` Extract of clean up routine
       If (Is a list(hlList))
⇒          CLEAR LIST(hlList;*)
       End if
```

2. See example for the command Load list.
3. See example for the command BLOB to list.

See Also

BLOB to list, Load list, New list.

version 6.0

Count list items (List) → Long

Parameter	Type		Description
List	ListRef	→	List reference number
Function result	Long	←	Number of items in expanded lists

Description

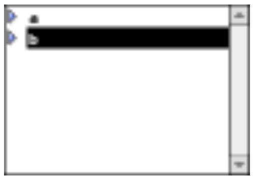
The command Count list items returns the number of items currently "visible" in the list whose reference number you pass in list.

Count list items does not return the total number of items in the list. It returns the number of items that are visible, depending on the current expanded/collapsed state of the list and its sublists.

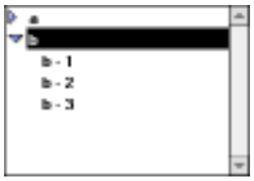
You apply this command to a list displayed in a form.

Examples

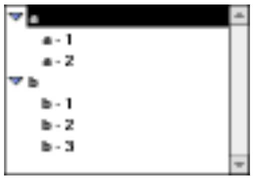
Here a list named hList shown in the User environment:



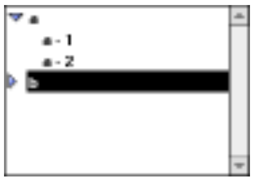
⇒ $vlNbItems:=**Count list items**(hList) ` at this point $vlNbItems gets 2



⇒ $vlNbItems:=**Count list items**(hList) ` at this point $vlNbItems gets 5



⇒ $vlNbItems:=**Count list items**(hList) ` at this point $vlNbItems gets 7



⇒ $vlNbItems:=**Count list items**(hList) ` at this point $vlNbItems gets 4

See Also

List item position, Selected list item.

Is a list

version 6.0

Is a list (list) → Boolean

Parameter	Type		Description
list	Number	→	ListRef value to be tested
Function result	Boolean	←	TRUE if list is a hierarchical list FALSE if list is not a hierarchical list

Description

The command Is a list returns TRUE if the value you pass in list is a valid reference to a hierarchical list. Otherwise, it returns FALSE.

Examples

1. See example for the command CLEAR LIST.
2. See examples for the command DRAG AND DROP PROPERTIES.

See Also

DRAG AND DROP PROPERTIES.

REDRAW LIST

REDRAW LIST (list)

Parameter	Type		Description
list	Variable	→	List reference number

Description

The command REDRAW LIST recalculates the positions of all the items and sublists (if any) of the list whose reference number you pass in list.

You MUST call this command at least once when you modify one or several aspects of a list or one of its sublists in a form.

Warning: Pass the actual variable instance of the list, not an expression or variable. For example, if you have a list named hList in a form:

```
     ` Recalculate the list after changes were made
REDRAW LIST (hList) ` GOOD
     ` ...

$vlList:=hList
     ` ...
     ` Recalculate the list after changes were made
REDRAW LIST ($vlList) ` WRONG
     ` ...
```

SET LIST PROPERTIES

version 6.0

SET LIST PROPERTIES (list; appearance{; icon{; lineHeight}})

Parameter	Type		Description
list	ListRef	→	List reference number
appearance	Number	→	Graphical style of the list 1 Hierarchical list ala Macintosh 2 Hierarchical list ala Windows
icon	Number	→	‘cicn’ MacOS-based resource ID or 0 for default platform node icon
lineHeight	Number	→	Minimal line height expressed in pixels

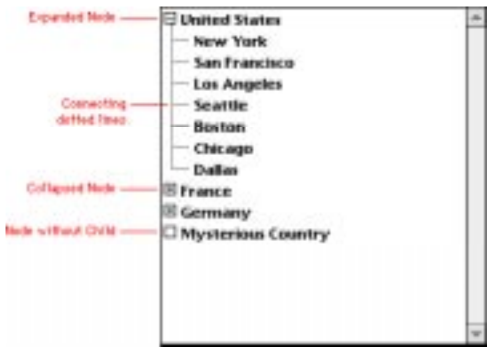
Description

The command SET LIST PROPERTIES sets the appearance of the hierarchical list whose list reference you pass in list.

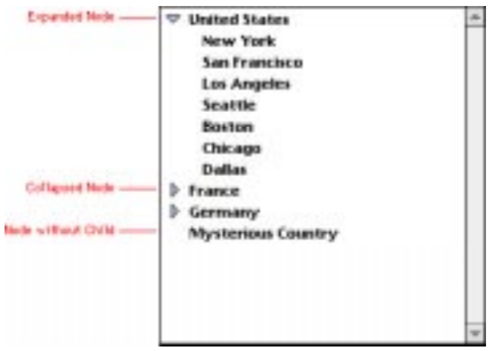
The parameter appearance can be one of the following predefined constants provided by 4th Dimension:

Constant	Type	Value
ala Macintosh	Long Integer	1
ala Windows	Long Integer	2

In the Windows appearance, the list has connecting dotted lines between the nodes and branches. One icon denotes the collapsed nodes, a second one the expanded nodes, a third one the nodes without child items. Here is a default hierarchical list in Windows appearance:



In the Macintosh appearance, the list has no connecting dotted lines. One icon denotes the collapsed nodes, a second one the expanded nodes. Nodes without child items have no icon. Here is a default hierachical list in Macintosh appearance:



Note: If you display a hierarchical list object without calling SET LIST PROPERTIES, the list appears with the default Windows or Macintosh appearances, depending on the Platform Interface property choosen for the object in the Design environment's Form Editor.

The parameter icon indicates the icons that will be displayed for each node. The value passed in icon sets the icon for collapsed nodes, icon+1 sets the icon for expanded nodes, and icon+2 sets the icon for nodes without child items (if the appearance is set to Windows).

For example, if you pass 15000, the color icon 'cicn' ID=15000 will be displayed for each collapsed node, the color icon 'cicn' ID=15001 will be displayed for each expanded node, and the color icon 'cicn' ID=15002 will be displayed for each node without child items.

It is therefore important to have these 'cicn' color icon resources present in your database structure file. If a color icon resource is missing, the corresponding nodes are displayed with no icons. (You can actually take advantage of this to display a list with no icons.)

WARNING: When creating 'cicn' color icon resources, use resource IDs greater than or equal to 15000. Resource IDs less than 15000 are reserved for 4th Dimension.

The resource IDs of the default Macintosh and Windows nodes are expressed by the following predefined constants provided by 4th Dimension:

Constant	**Type**	**Value**
Macintosh node	Long Integer	860
Windows node	Long Integer	138

In other words, 4th Dimension provides the following 'cicn' resources:

ID Number	Description
860	Collapsed node ala Macintosh
861	Expanded node ala Macintosh
138	Collapsed node ala Windows
139	Expanded node ala Windows
140	Node without child ala Windows

If you do not pass the parameter icon, the nodes are displayed with the default icons of the chosen appearance type.

Color icon resources can be of various sizes. For example, you can create 16x16 or 32x32 color icons.

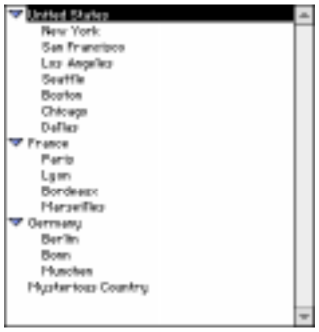
If you do not pass the parameter lineHeight, the line height of a hierarchical list is determined by the font and font size used for the object. If you use a color icons that is too tall or too wide, it will be displayed truncated and/or will be overidden by the connecting dotted lines (if appearance is Windows), as well as by the text of the nodes above or below it.

Choose color icon size, font, and font size accordingly, otherwise pass in the parameter lineHeight the minimal line height of the hierarchical list. If the value you pass is greater than the line height derived from the font and font size used, the line height of the hierarchical list will be forced to the value you pass.

Note: SET LIST PROPERTIES affects the way nodes are displayed in the hierarchical list. If you would rather customize the icon of each item in the list, use the command SET LIST ITEM PROPERTIES.

Examples

The following hierarchical list has been defined in the Design environment List Editor:



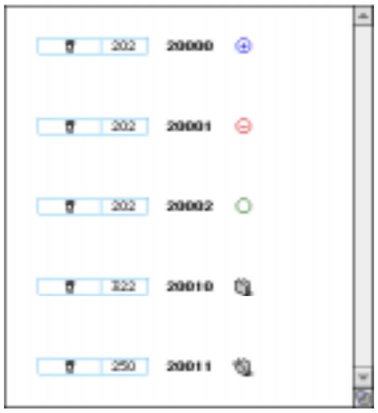
```
▼ United States
      New York
      San Francisco
      Los Angeles
      Seattle
      Boston
      Chicago
      Dallas
▼ France
      Paris
      Lyon
      Bordeaux
      Marseilles
▼ Germany
      Berlin
      Bonn
      Munchen
   Mysterious Country
```

Within a form, the hierarchical list object hlCities reuses that list with this object method:

```
        Case of
           : (Form event=On Load)
              hlCities:=Load list("Cities")
⇒             SET LIST PROPERTIES(hlCities;vlAppearance;vlIcon)
           : (Form event=On Unload)
              CLEAR LIST(hlCities;*)
        End case
```

In addition, the structure file of the database has been edited so it contains the following 'cicn' color icon resources:

5 'cicn' (Color Icon) Resources:

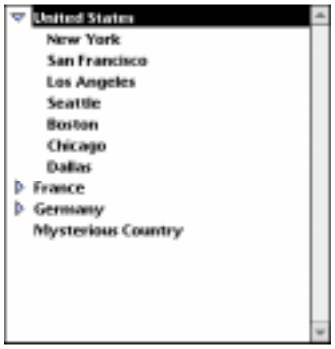


0	202	20000
0	202	20001
0	202	20002
0	322	20010
0	250	20011

1. With the following line:

```
⇒       SET LIST PROPERTIES(hlCities;ala Macintosh;Macintosh node)
```

The hierarchical list will look like this:

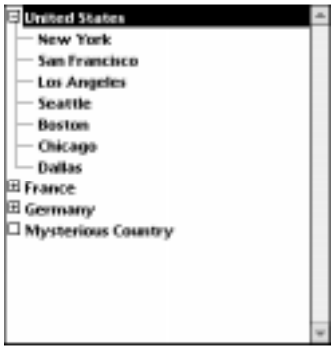


United States
New York
San Francisco
Los Angeles
Seattle
Boston
Chicago
Dallas
France
Germany
Mysterious Country

2. With the following line:

⇒ **SET LIST PROPERTIES**(hlCities;ala Windows;Windows node)

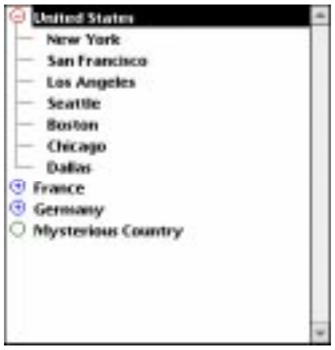
The hierarchical list will look like this:



3. With the following line:

⇒ **SET LIST PROPERTIES**(hlCities;ala Windows;20000)

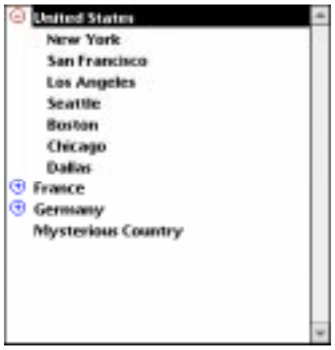
The hierarchical list will look like this:



4. With the following line:

⇒ **SET LIST PROPERTIES**(hlCities;ala Macintosh;20000)

The hierarchical list will look like this:

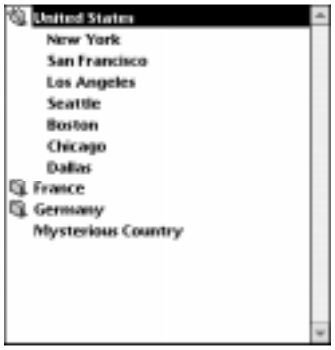


```
United States
  New York
  San Francisco
  Los Angeles
  Seattle
  Boston
  Chicago
  Dallas
France
Germany
Mysterious Country
```

5. With the following line:

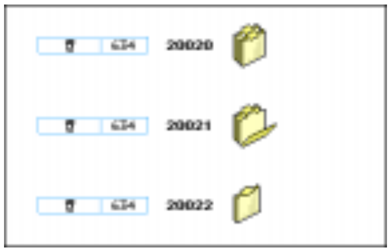
⇒ **SET LIST PROPERTIES**(hlCities;ala Macintosh;20010)

The hierarchical list will look like this:



```
United States
  New York
  San Francisco
  Los Angeles
  Seattle
  Boston
  Chicago
  Dallas
France
Germany
Mysterious Country
```

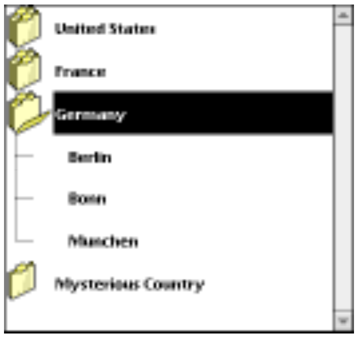
The 'cicn' color icon resources shown are then added to the structure file of the database:



6. With the following line:

⇒ **SET LIST PROPERTIES**(hlCities;<u>ala Windows</u>;20020;32)

The hierarchical list will look like this:



See Also

GET LIST ITEM PROPERTIES, GET LIST PROPERTIES, SET LIST ITEM PROPERTIES.

GET LIST PROPERTIES (list; appearance{; icon{; lineHeight}})

Parameter	Type		Description
list	ListRef	→	List reference number
appearance	Number	←	Graphical style of the list 1 Hierarchical list ala Macintosh 2 Hierarchical list ala Windows
icon	Number	←	'cicn' MacOS-based resource ID
lineHeight	Number	←	Minimal line height expressed in pixels

Description

The command GET LIST PROPERTIES returns information about the list whose reference number you pass in list.

The parameter appearance returns the graphical style of the list.

The parameter icon returns the resource IDs of the node icons displayed in the list.

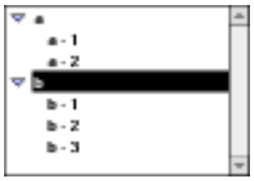
The parameter lineHeight returns the minimal line height.

These properties can be set using the command SET LIST PROPERTIES and/or in the Design environment List Editor, if the list was created there or saved using the command SAVE LIST.

For a complete description of the appearance, node icons, and minimal line height of a list, see the command SET LIST PROPERTIES.

Example

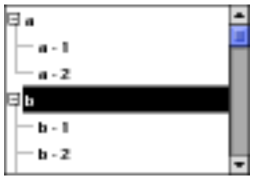
Given the list named hList, shown here in the User environment (in Macintosh appearance):



The object method for a button:

```
    ` bMacOrWin button Object Method
GET LIST PROPERTIES(hList;$vlAppearance;$vlIcon;$vlLH)
If ($vlAppearance=Ala Macintosh)
    $vlAppearance:=Ala Windows
    $vlIcon:=Windows node
    $vlLH:=20
Else
    $vlAppearance:=Ala Macintosh
    $vlIcon:=Macintosh node
    $vlLH:=0
End if
SET LIST PROPERTIES(hList;$vlAppearance;$vlIcon;$vlLH)
REDRAW LIST(hList) ` Do NOT forget to call REDRAW LIST otherwise the list won't be
updated
```

will alternately display the list as shown above and here (in Windows appearance):



See Also

SET LIST PROPERTIES.

SORT LIST

version 6.0

SORT LIST (list{; > or <})

Parameter	Type		Description
list	ListRef	→	List reference number
> or <		→	Sorting order: > to sort in ascending order, or < to sort in descending order

Description

The command SORT LIST sorts the list whose reference number is passed in list.

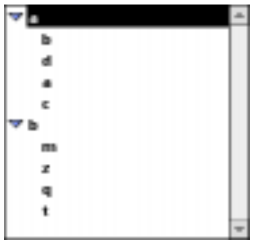
To sort in ascending order, pass >. To sort in descending order, pass <. If you omit the sorting order parameter, SORT LIST sorts in ascending order by default.

SORT LIST sorts all levels of the list; it first sorts the items of the list, then it sorts the items in each sublist (if any), and so on, through all the levels of the list. This is why you will usually apply SORT LIST to a list in a form. Sorting a sublist is of little interest because the order will be changed by a call to a higher level.

SORT LIST does not change the selected list items or the current expanded/collapsed state of the list and sublists. However, because the selected item can be moved by the sorting operation, Select list item may return a different position before and after the sort.

Example

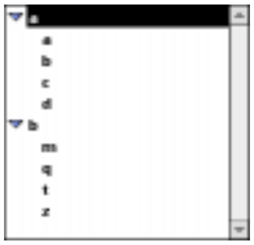
Given the list named hList, shown here in the User environment (in Macintosh appearance):



After the execution of this code:

```
           ` Sort the list and it sublists in ascending order
⇒      SORT LIST(hList;>)
           ` Do NOT forget to call REDRAW LIST otherwise the list won't be updated
       REDRAW LIST(hList)
```

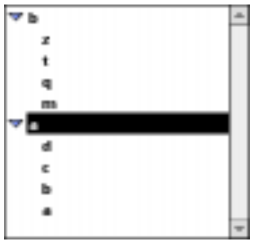
The list looks like:



After the execution of this code:

```
	` Sort the list and it sublists in ascending order
⇒	SORT LIST(hList;<)
	REDRAW LIST(hList) ` Do NOT forget to call REDRAW LIST otherwise the list won't be
updated
```

The list looks like:



See Also

Selected list item.

APPEND TO LIST

version 6.0

APPEND TO LIST (list; itemText; itemRef{; sublist{; expanded}})

Parameter	Type		Description
list	ListRef	→	List reference number
itemText	String	→	Text of the new list item (max. 31 characters)
itemRef	Number	→	Unique reference number for the new list item
sublist	ListRef	→	Optional sublist to attach to the new list item
expanded	Boolean	→	Indicates if the sublist will be expanded or collapsed

Description

The command APPEND TO LIST appends a new item to the hierarchical list whose list reference number you pass in list.

You pass the text of the item in itemText. You can pass a string or text expression of up to 31 characters. If you pass a longer value, it will be truncated.

You pass the unique reference number of the item in itemRef. Although we qualify this item reference number as unique, you can actually pass the value you want. See the Item Reference Numbers section below.

If you also want an item to have child items, pass a valid list reference to the child hierarchical list in sublist. To expand or collapse the child list, pass TRUE or FALSE in expanded.

The list reference you pass in sublist must refer to an existing list. The existing list may be empty, a one-level list, or a list with sublists. If you do not want to attach a child list to the new item, omit the parameter or pass 0. If you pass the sublist parameter and do not pass the expanded parameter, the sublist is not expanded, by default.

Tips

• To insert a new item in a list, use INSERT LIST ITEM. To change the text of an existing item or modify its child list as well as it expanded state, use SET LIST ITEM.
• To change the appearance of the new appended item use SET LIST ITEM PROPERTIES.

WARNING: If you append an item to a list currently displayed in a form or to a list that is attached to an item (through one or several levels) whose list is currently displayed in a form, you MUST call REDRAW LIST; 4D recalculates the list and displays it, reflecting your changes. The rule is simple: whatever the level of the list you act on, apply REDRAW LIST to the main list, which is list referenced by the object in the form.

Item Reference Numbers: What to do with them?

Each item of a hierarchical list has a Long Integer item reference number. This value is for your exclusive use: 4th Dimension only carries them. Here are some tips about what to do with them:

1. You do not need to uniquely identify each item (beginner level)

• First example: You programmatically build a Tab Control, for example, an address book. Since the Tab Control will return the number of the selected tab, you will probably not need more information. In this case, do not even bother about item reference numbers, pass 0 in the itemRef parameter. Note that for an address book Tab Control, you can predefine an A, B,..., Z list in the Design environment. However, you may want to create it programmatically in order to eliminate the letters for which there are no records (e.g., no records whose key field starts with Q).

• Second example: When working with a database, you progressively build a list of keywords. You can save the list at the end of each session, using the commands SAVE LIST or LIST TO BLOB, and reload it the beginning of each session, using Load list or BLOB to list. You display this list in a palette window. When you click on an item, you insert the clicked keyword in the current enterable area of the frontmost process. You can also use drag and drop. Anyway, what is important is that you will deal with the selected item (the one you clicked or dragged), because the commands Selected list item (click) and DRAG AND DROP PROPERTIES give you the position of the item you have to get. Using this position, you can obtain the text of the item using GET LIST ITEM. That's it. So you do not need to uniquely identify each item; you can pass 0 in the itemRef parameter.

2. You need to partially identify the list items (intermediate level)

You use item reference number for storing information required when you have to act on an item; this is explained in the next example. In this example, we use the item reference numbers for storing record numbers. However, we must be able to distinguish items corresponding the [Departments] records from those corresponding to the [Employees] records. Refer to the example for this command to see how this is done.

3. You need to uniquely identify the list items (advanced level)

You are programming an advanced handling of hierarchical lists, for which you absolutely need to uniquely identify each item at every level of the list. A simple way to do this is to maintain a private counter. Suppose you create a list hlList using New list. At this point, you initialize a counter vlhCounter to 0. Each time you call APPEND TO LIST or INSERT LIST ITEM, you increment this counter (vlhCounter:=vlhCounter+1), and you pass that counter as the item reference number. The trick is to not decrement the counter when you delete items—the counter can only grow. In doing so, you guarantee the uniqueness of item reference numbers. Since item reference numbers are Long Integer values, you can add or insert an item many times in a list that has been reinitialized. (Remember, however, if you work with thousands of items, you should use a table, not a list.)

Note: If use the Bitwise Operators you can also use item reference numbers for storing information that fit into a Long Integer value. It means: 2 Integer values, 4 byte values or 32 Booleans values.

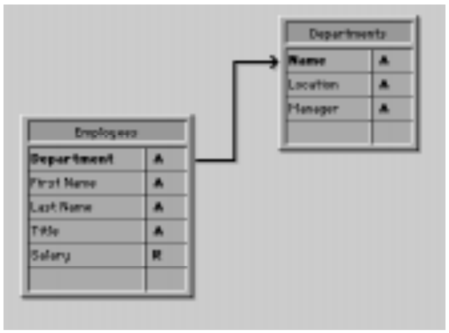
Why Do You Need Unique Reference Numbers?

In most cases, when using hierarchical lists for user interface purposes and when only dealing with the selected item (the one that was clicked or dragged), you will not need to use item reference numbers at all. Using Selected list item and GET LIST ITEM you have all you need to deal with the current selected item. In addition, commands such as INSERT LIST ITEM and DELETE LIST ITEM allow you to manipulate the list “relatively” to the selected item.

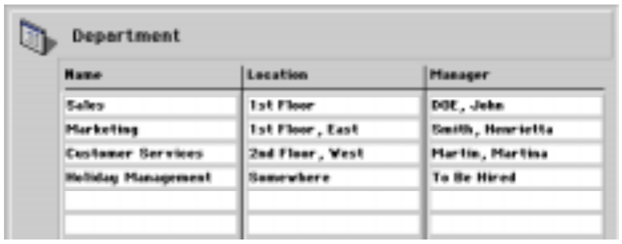
Basically, you need to deal with item reference numbers when you want programmatical direct access to any item of the list and not necessarily the one currently selected in the list.

Example

Here is a partial view of a database structure:

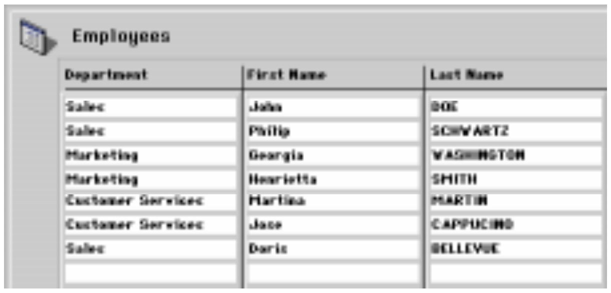


The [Departments] and [Employees] tables contain the following records:



Department

Name	Location	Manager
Sales	1st Floor	DOE, John
Marketing	1st Floor, East	Smith, Henrietta
Customer Services	2nd Floor, West	Martin, Martina
Holiday Management	Somewhere	To Be Hired



Employees

Department	First Name	Last Name
Sales	John	DOE
Sales	Philip	SCHWARTZ
Marketing	Georgia	WASHINGTON
Marketing	Henrietta	SMITH
Customer Services	Martina	MARTIN
Customer Services	Jose	CAPPUCINO
Sales	Doris	BELLEVUE

You want to display a hierarchical list, named hlList, that shows the Departments, and for each Department, a child list that shows the Employees working in that Department. The object method of hlList is:

```
      ` hlList Hierarchical List Object Method

   Case of

      : (Form event=On Load)
         C_LONGINT(hlList;$hSubList;$vlDepartment;$vlEmployee)
            ` Create a new empty hierarchical list
         hlList:=New list
            ` Select all the records from the [Departments] table
         ALL RECORDS([Departments])
            ` For each Department
         For ($vlDepartment;1;Records in selection([Departments]))
               ` Select the Employees from this Department
            RELATE MANY([Departments]Name)
               ` How many are they?
            $vlNbEmployees:=Records in selection([Employees])
               ` Is there at least one Employee in this Department?
            If ($vlNbEmployees>0)
                  ` Create a  child list for the Department item
               $hSubList:=New list
                  ` For each Employee
               For ($vlEmployee;1;Records in selection([Employees]))
                     ` Add the Employee item to the sublist
                     ` Note that the record number of the [Employees] record
                     ` is passed as item reference number
                  APPEND TO LIST($hSubList;[Employees]Last Name+", "+
                              [Employees]First Name;Record number([Employees]))
                     ` Go the next [Employees] record
                  NEXT RECORD([Employees])
               End for
            Else
```

```
                ` No Employees, No child list for the Department item
            $hSubList:=0
        End if
            ` Add the Department item to the main list
            ` Note that the record number of the [Departments] record
            ` is passed as item reference number. The bit #31
            ` of the item reference number is forced to one so we'll be able
            ` to distinguish Department and Employee items. See note further
            ` below on why we can use this bit as supplementary information about
            ` the item.
        APPEND TO LIST(hlList;[Departments]Name;
                0x80000000 | Record number([Departments]);$hSublist;$hSubList # 0)
            ` Set the Department item in Bold to emphasize the hierarchy of the list
        SET LIST ITEM PROPERTIES(hlList;0;False;Bold;0)
            ` Go to the next Department
        NEXT RECORD([Departments])
    End for
        ` Sort the whole list in ascending order
    SORT LIST(hlList;>)
        ` Display the list using the Windows style
        ` and force the minimal line height to 14 Pts
    SET LIST PROPERTIES(hlList;ala Windows;Windows node;14)

: (Form event=On Unload)
        ` The list is no longer needed, do not forget to get rid of it!
    CLEAR LIST(hlList;*)

: (Form event=On Double Clicked)
        ` A double-clicked occurred
        ` Get the position of the selected item
    $vlItemPos:=Selected list item(hlList)
        ` Just in case, check the position
    If ($vlItemPos # 0)
            ` Get the list item information
        GET LIST ITEM(hlList;$vlItemPos;$vlItemRef;$vsItemText;
                                                $vlItemSubList;$vbItemSubExpanded)
            ` Is the item a Department item?
        If ($vlItemRef ?? 31)
                ` If so, it is a double-click on a Department Item
            ALERT("You double-clicked on the Department item "+
                                                Char(34)+$vsItemText+Char(34)+".")
        Else
                ` If not, it is a double-click on an Employee item
                ` Using the parent item reference number find the [Departments] record
            GOTO RECORD([Departments];List item parent(hlList;$vlItemRef) ?- 31)
```

```
                  ` Tell where the Employee is working and to whom he or she is reporting
               ALERT("You double-clicked on the Employee item "+Char(34)
                           +$vsItemText+Char(34)+ " who is working in the Department "
                          +Char(34)+[Departments]Name+Char(34)+" whose manager is "
                                    +Char(34)+[Departments]Manager+Char(34)+".")
            End if
         End if

   End case

      ` Note: 4th Dimension can store up to 16 millions records per table
      ` (precisely 16,777,215). This value is 2^24 minus one. Record number
      ` fit on 24 bits. In our example, we use the bit #31 of the unused high byte for
      ` distinguishing Employees and Departments items.
```

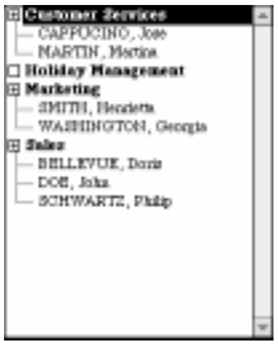
In this example, there is only one reason to distinguish [Departments] items and [Employees] items:

1. We store record numbers in the item reference numbers, therefore, we will probably end up with [Departments] items whose item reference numbers are the same as [Employees] items.

2. We use the command List parent item to retrieve the parent of the selected item. If we click on an [Employees] item whose associated record number is #10, if there is also a [Departments] item #10, the [Departments] item will be found first by List parent item when it browses the lists to locate the item with the item reference number we pass. The command will return the parent of the [Departments] item and not the parent of [Employees] item.

Therefore, we made the item reference numbers unique, not because we wanted unique numbers, but because we needed to distinguish [Departments] and [Employees] records.

In the User or Custom Menus environments, the list will look like this:



```
⊞ Customer Services
   ├─ CAPPUCINO, Jose
   └─ MARTIN, Martine
□ Holiday Management
⊞ Marketing
   ├─ SMITH, Henrietta
   └─ WASHINGTON, Georgia
⊞ Sales
   ├─ BELLEVUE, Doris
   ├─ DOE, John
   └─ SCHWARTZ, Philip
```

Note: This example is useful for user interface purposes if you deal with a reasonably <u>small</u> number of records. Remember that lists are held in memory—do not build user interfaces with hierarchical lists containing thousands of items.

version 6.0

INSERT LIST ITEM (list; beforeItemRef | *; itemText; itemRef{; sublist{; expanded}})

Parameter	Type		Description
list	ListRef	→	List reference number
beforeItemRef | *	Number | *	→	Item reference number or * for current selected list item
itemText	String	→	Text for the new list item (max. 31 characters)
itemRef	Number	→	Unique reference number for the new list item
sublist	ListRef	→	Optional sublist to attach to the new list item
expanded	Boolean	→	Indicates if the sublist will be expanded or collapsed

Description

The command INSERT LIST ITEM inserts a new item in the list whose reference number you pass in list.

If you pass * as second parameter, the item is inserted before the current selected item in the list. In this case, the newly inserted item will also become the selected item.

Otherwise, if you want to insert an item before a specific item, you pass the item reference number of that item. In this case, the newly inserted item is not automatically selected. If there is no item with that item reference number, the command does nothing.

You pass the text and the item reference number of the new item in itemText and itemRef.

Example

The following code inserts an item (with no attached sublist) just before the item currently selected in the list hList:

```
	vlUniqueRef:=vlUniqueRef+1
⇒	INSERT LIST ITEM(hList;*;"New Item";vlUniqueRef)
	REDRAW LIST(hList)
```

See Also

APPEND TO LIST.

SET LIST ITEM PROPERTIES (list; itemRef; enterable; styles; icon)

Parameter	Type		Description
list	ListRef	→	List reference number
itemRef	Number	→	Item reference number, or 0 for last item appended to the list
enterable	Boolean	→	TRUE = Enterable, FALSE = Non-enterable
styles	Number	→	Font style for the item
icon	Number	→	'cicn' MacOS-based resource ID, or 65536 + 'PICT' MacOS-based resource ID, or 131072 + Picture Reference Number

Description

The command SET LIST ITEM PROPERTIES modifies the item whose item reference number is passed in itemRef within the list whose reference number is passed in list.

If there is no item with the item reference number that is passed, the command does nothing. You can optionally pass 0 in itemRef to modify the last item added to the list using APPEND TO LIST.

If you work with item reference numbers, build a list in which items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, refer to the description of thecommand APPEND TO LIST.

Note: To change the text of the item or its sublist, use the command SET LIST ITEM.

To make an item enterable, pass TRUE in enterable; otherwise, pass FALSE.

Important: In order for an item to be enterable, it must belong to a list that is enterable. To make a whole list enterable, use the SET ENTERABLE command. To make an individual list item enterable, use SET LIST ITEM PROPERTIES. Changing the enterable property at the list level does not affect the enterable properties of the items. However, an item can be enterable only if its list is enterable.

You specify the font style of the item in the styles parameter. You pass a combination (one or a sum) of the following predefined constants:

Constant	Type	Value
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	4
Outline	Long Integer	8
Shadow	Long Integer	16
Condensed	Long Integer	32
Extended	Long Integer	64

Note: On Windows, only the styles Plain or a combination of Bold, Italic, and Underline are available.

To associate an icon to the item, pass one of the following numeric values:
• N, where N is the resource ID of MacOS-based ‘cicn’ resource
• Use PICT resource+N, where N is the the resource ID of a MacOS-based ‘PICT’ resource
• Use PicRef+N, where N is the reference number of a Picture from the Design environment Picture Library

Pass zero (0), if you do not want any graphic for the item.

Note: Use PICT resource and Use PicRef are predefined constants provided by 4D.

Example

See the example for the command APPEND TO LIST.

See Also

GET LIST ITEM PROPERTIES, SET LIST ITEM.

GET LIST ITEM PROPERTIES

GET LIST ITEM PROPERTIES (list; itemRef; enterable{; styles{; icon}})

Parameter	**Type**		**Description**
list	ListRef	→	List reference number
itemRef	Number	→	Item reference number
enterable	Boolean	←	TRUE = Enterable, FALSE = Non-enterable
styles	Number	←	Font style for the item
icon	Number	←	'cicn' MacOS-based resource ID, or 65536 + 'PICT' MacOS-based resource ID, or 131072 + Picture Reference Number

Description

The command GET LIST ITEM PROPERTIES returns the properties of the item whose reference number is passed in itemRef within the list whose list reference number is passed in list.

After the call:
• enterable returns TRUE if the item is enterable.
• styles returns the font style of the item.
• icon returns the icon or picture assigned to the item, 0 if none.

For details about these properties, see the description of the command SET LIST ITEM PROPERTIES.

If there is no item with the item reference number that is passed, the command leaves the parameters unchanged.

If you work with item reference numbers, build a list in which items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, refer to the description of the command APPEND TO LIST.

See Also

GET LIST ITEM, SET LIST ITEM, SET LIST ITEM PROPERTIES.

List item position

List item position (list; itemRef) → Number

Parameter	Type		Description
list	ListRef	→	List reference number
itemRef	Number	→	Item reference number
Function result	Number	←	Item position in expanded lists

Description

The command List item position returns the position of the item whose item reference number is passed in itemRef, within the list whose list reference number is passed in list.

The position is expressed relative to the top item of the main list, using the current expanded/collapsed state of the list and its sublist.

The result is therefore a number between 1 and the value returned by Count list items.

If the item is not visible because it is located in a collapsed list, List item position expands the appropriate list to make the item visible.

If the item does not exist, List item position returns 0.

See Also

Count list items, SELECT LIST ITEM BY REFERENCE.

List item parent

version 6.0

List item parent (list; itemRef) → Number

Parameter	Type		Description
list	ListRef	→	List reference number
itemRef	Number	→	Item reference rumber
Function result	Number	←	Item reference number of parent item 0 if none

Description

The command List item parent returns the item reference number of a parent item.

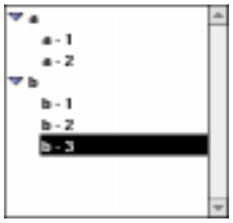
You pass a list reference number in list; you pass the item reference number of an item of the list in itemRef. In return, if the item reference number refers to an existing item in the list and if this item is in a sublist (and therefore has a parent item), you obtain the item reference number of a parent item.

If there is no item with the item reference number you passed or if the item has no parent, List item parent returns 0 (zero).

If you work with item reference numbers, build a list in which the items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, see the description of the command APPEND TO LIST.

Examples

Given the list named hList shown here in the User environment:



The item reference numbers are set as follows:

Item	Item Reference Number
a	100
a - 1	101
a - 2	102
b	200
b - 1	201
b - 2	202
b - 3	203

• In the following code, if the item “b - 3” is selected, the variable $vlParentItemRef gets 200, the item reference number of the item “b”:

```
       $vlItemPos:=Selected list item(hList)
       GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText)
⇒      $vlParentItemRef:=List item parent(hList;$vlItemRef) ` $vlParentItemRef gets 200
```

• If the item “a - 1” is selected, the variable $vlParentItemRef gets 100, the item reference number of the item “a”.

• If the item “a” or “b” is selected, the variable $vlParentItemRef gets 0, because these items have no parent item.

See Also

GET LIST ITEM, List item position, SELECT LIST ITEM BY REFERENCE, SET LIST ITEM.

DELETE LIST ITEM

version 6.0

DELETE LIST ITEM (list; itemRef | *{; *})

Parameter	Type		Description
list	ListRef	→	List reference number
itemRef | *	Number | *	→	Item reference number, or * for current selected list item
*		→	If specified, clear sublists (if any) from memory If omitted, sublists (if any) are not cleared

Description

The command DELETE LIST ITEM deletes an item from the list whose list reference number is passed in list.

If you pass * as second parameter, you delete the current selected item in the list.

Otherwise, you specify the item reference number of the item you want to delete. If there is no item with the item reference number you passed, the command does nothing.

If you work with item reference numbers, build a list in which the items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, see the description of the command APPEND TO LIST.

No matter which item you delete, you should specify the optional * parameter to let 4D automatically delete the sublist attached to the item, if any. If you do not specify the * parameter, it is a good idea to have previously obtained the list reference number of the (possible) sublist attached to the item, because eventually you will have to delete it, using the command CLEAR LIST.

Example

The following code deletes the current selected item of the list hList. If the item has an attached sublist, the sublist (as well as any sub-sublist) is cleared:

```
⇒   DELETE LIST ITEM(hList;*;*)
       ` Do NOT forget to call REDRAW LIST otherwise the list won't be updated
    REDRAW LIST(hList)
```

See Also

CLEAR LIST, GET LIST ITEM.

GET LIST ITEM

version 6.0

GET LIST ITEM (list; itemPos; itemRef; itemText{; sublist{; expanded}})

Parameter	Type		Description
list	ListRef	→	List reference number
itemPos	Number	→	Position of item in expanded lists
itemRef	Number	←	Item reference number
itemText	String	←	Text of the list item
sublist	ListRef	←	Sublist list reference number (if any)
expanded	Boolean	←	If a sublist is attached: TRUE = sublist is currently expanded FALSE = sublist is currently collapsed

Description

The command GET LIST ITEM returns information about the item whose position is passed in itemPos within the list whose reference number is passed in list.

The position must be expressed relatively, using the current expanded/collaped state of the list and its sublist. You pass a position value between 1 and the value returned by Count list items. If you pass a value outside this range, GET LIST ITEM returns your parameters unchanged.

After the call, you retrieve:
• The item reference number of the item in itemRef.
• The text of the item in itemText.

If you passed the optional parameters sublist and expanded:
• subList returns the list reference number of the sublist attached to the item. If the item has no sublist, subList returns zero (0).
• If the item has a sublist, expanded returns TRUE if the sublist is currently expanded, and FALSE if it is collapsed.

Example

hList is a list whose items have unique reference numbers. The following code programmatically toggles the expanded/collapsed state of the sublist, if any, attached to the current selected item:

```
      $vlItemPos:=Selected list item(hList)
      If ($vlItemPos>0)
⇒         GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText;$hSublist;$vbExpanded)
          If (Is a list($hSublist))
              SET LIST ITEM(hList;$vlItemRef;$vsItemText;$vlItemRef;
                                                                  $hSublist;Not($vbExpanded))
              REDRAW LIST(hList)
          End if
      End if
```

See Also

GET LIST ITEM PROPERTIES, List item parent, List item position, Selected list item, SET LIST ITEM, SET LIST ITEM PROPERTIES.

SET LIST ITEM

version 6.0

SET LIST ITEM (list; itemRef; newItemText; newItemRef{; sublist{; expanded}})

Parameter	Type		Description
list	ListRef	→	List reference number
itemRef	Number	→	Item reference number or 0 for last appended to the list
newItemText	String	→	New item text
newItemRef	Number	→	New item reference number
sublist	ListRef	→	New sublist attached to item, or 0 for no sublist (detaching current one, if any), or -1 for no change
expanded	Boolean	→	Indicates if the sublist will be expanded or collapsed

Description

The command SET LIST ITEM modifies the item whose item reference number is passed in itemRef within the list whose reference number is passed in list.

If there is no item with the item reference number you passed, the command does nothing. You can optionally pass 0 in itemRef to modify the last item added to the list using APPEND TO LIST.

If you work with item reference numbers, build a list in which the items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, see the description of the command APPEND TO LIST.

You pass the new text for the item in newItemText. To change the item reference number, pass the new value in newItemRef; otherwise, pass the same value as itemRef.

To attach a list to the item, pass the list reference number in subList. In this case, you also specify if the newly sublist is expanded by passing TRUE in expanded; otherwise, pass FALSE.

To detach a sublist already attached to the item, pass 0 (zero) in sublist. In this case, it is a good idea to have previously obtained the reference number of that list using GET LIST ITEM, so you can later delete the sublist using CLEAR LIST, if you no longer need it.

If you do not want to change the sublist property of the item, pass -1 in sublist.

Examples

1. hList is a list whose items have unique reference numbers. The following object method for a button adds a child item to the current selected list item.

```
$vlItemPos:=Selected list item(hList)
If ($vlItemPos>0)
    GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText;$hSublist;$vbExpanded)
    $vbNewSubList:=Not(Is a list($hSublist))
    If ($vbNewSubList)
        $hSublist:=New list
    End if
    vlUniqueRef:=vlUniqueRef+1
    APPEND TO LIST($hSubList;"New Item";vlUniqueRef)
    If ($vbNewSubList)
        SET LIST ITEM(hList;$vlItemRef;$vsItemText;$vlItemRef;$hSublist;True)
    End if
    SELECT LIST ITEM BY REFERENCE(hList;vlUniqueRef)
    REDRAW LIST(hList)
End if
```

2. See example for the command GET LIST ITEM.
3. See example for the command APPEND TO LIST.

See Also

GET LIST ITEM, GET LIST ITEM PROPERTIES, SET LIST ITEM PROPERTIES.

Selected list item

version 6.0

Selected list item (list) → Long

Parameter	Type		Description
list	ListRef	→	List reference number
Function result	Long	←	Position of current selected list item in expanded list

Description

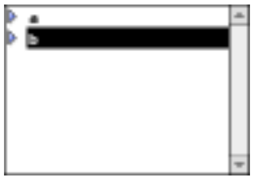
The command Selected list item returns the position of the selected item in the list whose reference number you pass in list.

You apply this command to a list displayed in a form to detect which item the user has selected.

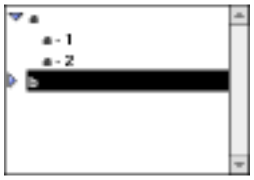
If the list has sublists, you apply the command to the main list (the one actually defined in the form), not one of its sublists. The position is expressed relative to the top item of the main list, using the current expanded/collapsed state of the list and its sublist.

Examples

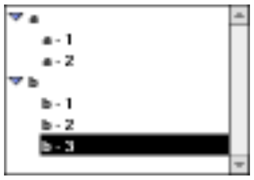
Here a list named hList, shown in User environment:



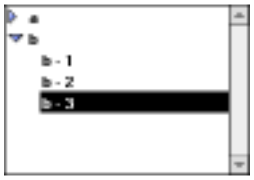
⇒ $vlItemPos:=**Selected list item**(hList) ` at this point $vlItemPos gets 2



⇒ $vlItemPos:=**Selected list item**(hList) ` at this point $vlItemPos gets 4



⇒ $vlItemPos:=**Selected list item**(hList) ` at this point $vlItemPos gets 7



⇒ $vlItemPos:=**Selected list item**(hList) ` at this point $vlItemPos gets 5

See Also

SELECT LIST ITEM, SELECT LIST ITEM BY REFERENCE.

SELECT LIST ITEM

version 6.0

SELECT LIST ITEM (list; itemPos)

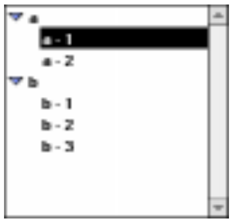
Parameter	Type		Description
list	ListRef	→	List reference number
itemPos	Number	→	Position of item in expanded list

Description

The command SELECT LIST ITEM selects the item whose position is passed in itemPos within the list whose reference number is passed in list. The parameter itemPos is a position expressed using the current expanded/collapsed state of the list and its sublists. You pass a position value between 1 and the value returned by Count list items. If you pass a value outside this range, the first item is selected by default.

Examples

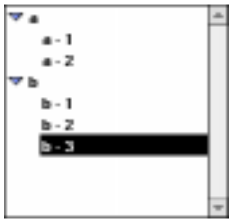
Given the list named hList, shown here in the User environment:



After the execution of this code:

```
⇒	SELECT LIST ITEM(hList;Count list items(hList))
		` Do NOT forget to call REDRAW LIST otherwise the list won't be updated
	REDRAW LIST(hList)
```

The last visible list item is selected:



See Also

SELECT LIST ITEM BY REFERENCE, Selected list item.

SELECT LIST ITEM BY REFERENCE

version 6.0

SELECT LIST ITEM BY REFERENCE (list; itemRef)

Parameter	Type		Description
list	ListRef	→	List reference number
itemRef	Number	→	Item reference number

Description

The command SELECT LIST ITEM BY REFERENCE selects the item whose item reference number is passed in itemRef within the list whose reference number is passed in list.

If there is no item with the item reference number you passed, the command does nothing.

If the item is not currently visible (i.e., it is located in a collapsed sublist), the command expands the required sublist(s) so that the new selected item becomes visible.

If you work with item reference numbers, builds a list in which the items have unique reference numbers, otherwise you will not be able to distinguish the items. For more information, see the description of the command APPEND TO LIST.

Example

hList is a list whose items have unique reference numbers. The following object method for a button selects the parent item (if any) of the current selected item:

```
	  ` Get position of selected item
	$vlItemPos:=Selected list item(hList)
	  ` Get item ref. num. of selected item
	GET LIST ITEM(hList;$vlItemPos;$vlItemRef;$vsItemText)
	  ` Get item ref. num. of parent item (if any)
	$vlParentItemRef:=List item parent(hList;$vlItemRef)
	If ($vlParentItemRef>0)
		  ` Select the parent item
⇒		SELECT LIST ITEM BY REFERENCE(hList;List item parent(hList;$vlItemRef))
		  ` Do NOT forget to call REDRAW LIST otherwise the list won't be updated
		REDRAW LIST(hList)
	End if
```

See Also

SELECT LIST ITEM, Selected list item.

20 Import and Export

IMPORT TEXT

version 3

IMPORT TEXT ({table; }document)

Parameter	Type		Description
table	Table	→	Table into which to import data, or Default table, if omitted
document	String	→	Text document from which to import data

Description

The command IMPORT TEXT reads data from document, a Windows or Macintosh text document, into the table table by creating new records for that table.

The import operation is performed through the current input form. The import operation reads fields and variables based on the layering of objects in the input form. For this reason, you should be very careful about the front-to-back order of text objects (fields and variables) in the form. The first object into which data will be imported should be in the back of the form, and so on. If the number of fields or variables in the form does not match the number of fields being imported, the extra ones are ignored. An input form used for importing cannot contain any buttons. Subform objects are ignored.

Note: One way to ensure that the data is imported into the correct objects is to select the object into which the first field should be imported and move it to the front. Continue to move fields and variables to the front in order, making sure that you have one field or variable for each field being imported.

An On Validate event is sent to the form method for each record that is imported. If you use variables in the import form, use this event to copy data from variables to fields, .

The document parameter can include a path that contains volume and folder names. If you pass an empty string, the standard Open File dialog box is displayed. If the user cancels this dialog, the import operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during import. The user can cancel the operation by clicking a button labeled Stop. Records that have already been imported will not be removed if the user presses the Stop button. If the import is successfully completed, the OK system variable is set to 1. If an error occurs or the operation was interrupted, the OK variable is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The import operation is made using the default ASCII map for the platform on which it is executed, unless you change the ASCII map (using the command USE ASCII MAP) prior to the import. An ASCII map can be used to convert the data coming from other platforms that have a different ASCII table.

Using IMPORT TEXT, the default field delimiter is the tab character (ASCII 9). The default record delimiter is the carriage return character (ASCII 13). You can change these defaults by assigning values to the two delimiter system variables: FldDelimit and RecDelimit. The user can change the
defaults in the User environment's Import Data dialog box. Text fields may contain carriage returns, therefore, be careful when using a carriage return as a delimiter if you are importing text fields.

Example

The following example imports data from a text document. The method first sets the input form so that the data will be imported through the correct form, changes the 4D delimiter variables, then performs the import:

```
	INPUT FORM([People]; "Import")
	FldDelimit:=27 ` Set field delimiter to Escape character
	RecDelimit:=10 ` Set record delimiter to Line Feed character
⇒	IMPORT TEXT([People];"NewPeople") ` Import from “NewPeople” document
```

See Also

EXPORT TEXT, IMPORT DIF, IMPORT SYLK, USE ASCII MAP.

System Variables and Sets

OK is set to 1 if the import is successfully completed; otherwise, it is set to 0.

EXPORT TEXT

version 3

EXPORT TEXT (table; document)

Parameter	Type		Description
table	Table	→	Table from which to export data, or Default table, if omitted
document	String	→	Text document to receive the data

Description

The command EXPORT TEXT writes data from the records of the current selection of table in the current process. The data is written to document, a Windows or Macintosh text document on the disk.

The export operation is performed through the current output form. The export operation writes fields and variables based on the entry order of the output form. For this reason, you should use an output form that contains only the fields or enterable objects that you wish to export. Do not place buttons or other extraneous objects on the export form. Subform objects are ignored.

An On Load event is sent to the form method for each record that is exported. Use this event to set the variables you may use in the export form.

The document parameter can name a new or existing document. If document is given the same name as an existing document, the existing document is overwritten. The document can include a path that contains volume and folder names. If you pass an empty string, the standard Save File dialog box is displayed. If the user cancels this dialog, the export operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a Stop button. If the export is successfully completed, the OK system variable is set to 1. If the operation is canceled or an error occurs, the OK system variable is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The export operation is made using the default ASCII map for the platform on which it is executed, unless you change the ASCII map (using the command USE ASCII MAP) prior to the export. An ASCII map can be used to convert the data for use on other platforms that have a different ASCII table.

Using EXPORT TEXT, the default field delimiter is the tab character (ASCII 9). The default record delimiter is the carriage return character (ASCII 13). You can change these defaults by assigning values to the two delimiter system variables: FldDelimit and RecDelimit. The user can change the
defaults in the User environment Export Data dialog box. Text fields may contain carriage returns, therefore, be careful when using a carriage return as a delimiter if you are exporting text fields.

Example

The following example exports data to a text document. The method first sets the output form so that the data will be exported through the correct form, changes the 4D delimiter variables, then performs the export:

```
        OUTPUT FORM([People];"Export")
        FldDelimit:=27 ` Set field delimiter to Escape character
        RecDelimit:=10 ` Set record delimiter to Line Feed character
⇒       EXPORT TEXT([People];"NewPeople") ` Export to the "NewPeople" document
```

See Also

EXPORT DIF, EXPORT SYLK, IMPORT TEXT, USE ASCII MAP.

System Variables and Sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

IMPORT SYLK

version 3

IMPORT SYLK ({table; }document)

Parameter	Type		Description
table	Table	→	Table into which to import data, or Default table, if omitted
document	String	→	SYLK document from which to import data

Description

The command IMPORT SYLK reads data from document, a Windows or Macintosh SYLK document, into the table table by creating new records for that table.

The import operation is performed through the current input form. The import operation reads fields and variables based on the layering of objects in the input form. For this reason, you should be very careful about the front-to-back order of text objects (fields and variables) in the form. The first object into which data will be imported should be in the back of the form, and so on. If the number of fields or variables in the form does not match the number of fields being imported, the extra ones are ignored. An input form used for importing cannot contain any buttons. Subform objects are ignored.

Note: One way to ensure that the data is imported into the correct objects is to select the object into which the first field should be imported and move it to the front. Continue to move the fields and variables to the front, in order, making sure that you have one field or variable for each field being imported.

An On Validate event is sent to the form method for each record that is imported. If you use variables in the import form, use this event to copy data from variables to fields, .

The document parameter can include a path that contains volume and folder names. If you pass an empty string, the standard Open File dialog box is displayed. If the user cancels this dialog, the import operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during the import. The user can cancel the operation by clicking a Stop button. Records that have already been imported will not be removed if the user presses the Stop button. If the import is successfully completed, the OK system variable is set to 1. If an error occurs or the operation was interrupted, the OK variable is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The import operation is made using the default ASCII map for the platform on which it is executed, unless you change the ASCII map (using the command USE ASCII MAP) prior to the import. An ASCII map can be used to convert the data coming from platforms that have a different ASCII table.

Example

The following example imports data from a SYLK document. The method first sets the input form so the data will be imported through the correct form, then performs the import:

```
	INPUT FORM([People]; "Import")
⇒	IMPORT SYLK([People];"NewPeople") ` Import from “NewPeople” document
```

See Also

EXPORT SYLK, IMPORT DIF, IMPORT TEXT, USE ASCII MAP.

System Variables and Sets

OK is set to 1 if the import is successfully complete; otherwise, it is set to 0.

EXPORT SYLK

version 3

EXPORT SYLK ({table; }document)

Parameter	Type		Description
table	Table	→	Table from which to export data, or Default table, if omitted
document	String	→	SYLK document to receive the data

Description

The command EXPORT SYLK writes data from the records of the current selection of table in the current process. The data is written to document, a Windows or Macintosh Sylk document on the disk.

The export operation is performed through the current output form. The export operation writes fields and variables based on the entry order of the output form. For this reason, you should use an output form that contains only the fields or enterable objects that you wish to export. Do not place buttons or other extraneous objects on the export form. Subform objects are ignored.

An On Load event is sent to the form method for each record that is exported. Use this event to set the variables you may use in the export form.

The document parameter can name a new or existing document. If document is given the same name as an existing document, the existing document is overwritten. The document can include a path that contains volume and folder names. If you pass an empty string, the standard Save File dialog box is displayed. If the user cancels this dialog, the export operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a Stop button. If the export is successfully completed, the OK system variable is set to 1. If the operation is canceled or an error occurs, the OK system variable is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The export operation is made using the default ASCII map for the platform on which it is executed, unless you change the ASCII map (using the command USE ASCII MAP) prior to the export. An ASCII map can be used to convert the data for use on platforms that have a different ASCII table.

Example

The following example exports data to a SYLK document. The method first sets the output form so that the data will be exported through the correct form, then performs the export:

```
	OUTPUT FORM([People];"Export")
⇒	EXPORT SYLK([People];"NewPeople") ` Export to the "NewPeople" document
```

See Also

EXPORT DIF, EXPORT TEXT, IMPORT SYLK, USE ASCII MAP.

System Variables and Sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

IMPORT DIF

version 3

IMPORT DIF ({table; }document)

Parameter	Type		Description
table	Table	→	Table into which to import data, or Default table, if omitted
document	String	→	DIF document from which to import data

Description

The command IMPORT DIF reads data from document, a Windows or Macintosh DIF document, into the table table by creating new records for that table.

The import operation is performed through the current input form. The import operation reads fields and variables based on the layering of objects in the input form. For this reason, you should be very careful about the front-to-back order of text objects (fields and variables) in the form. The first object into which data will be imported should be in the back of the form, and so on. If the number of fields or variables in the form does not match the number of fields being imported, the extra ones are ignored. An input form used for importing cannot contain any buttons. Subform objects are ignored.

Note: One way to ensure that the data is imported into the correct objects is to select the object into which the first field should be imported and move it to the front. Continue to move the fields and variables to the front, in order, making sure that you have one field or variable for each field being imported.

An On Validate event is sent to the form method for each record that is imported. Use this event to copy data from variables to fields, if you use variables in the import form.

The document parameter can include a path that contains volume and folder names. If you pass an empty string, the standard Open File dialog box is displayed. If the user cancels this dialog, the import operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during import. The user can cancel the operation by clicking a Stop button. Records that have already been imported will not be removed if the user presses the Stop button. If the import is successfully completed, the OK system variable is set to 1. If an error occurs or the operation was interrupted, the OK variable is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The import operation is made using the default ASCII map for the platform on which it is executed, unless you change the ASCII map (using the command USE ASCII MAP) prior to the import. An ASCII map can be used to convert the data coming from platforms that have a different ASCII table.

Example

The following example imports data from a DIF document. The method first sets the input form so that the data will be imported through the correct form, then performs the import:

```
	INPUT FORM([People]; "Import")
⇒	IMPORT DIF([People];"NewPeople") ` Import from “NewPeople” document
```

See Also

EXPORT DIF, IMPORT SYLK, IMPORT TEXT, USE ASCII MAP.

System Variables and Sets

OK is set to 1 if the import is successfully completed; otherwise, it is set to 0.

EXPORT DIF

version 3

EXPORT DIF ({table; }document)

Parameter	Type		Description
table	Table	→	Table from which to export data,or Default table, if omitted
document	String	→	DIF document to receive the data

Description

The command EXPORT DIF writes data from the records of the current selection of table in the current process. The data is written to document, a Windows or Macintosh DIF document on the disk.

The export operation is performed through the current output form. The export operation writes fields and variables based on the entry order of the output form. For this reason, you should use an output form that contains only the fields or enterable objects that you wish to export. Do not place buttons or other extraneous objects on the export form. Subform objects are ignored.

An On Load event is sent to the form method for each record that is exported. Use this event to set the variables you may use in the export form.

The document parameter can name a new or existing document. If document is given the same name as an existing document, the existing document is overwritten. The document can include a path that contains volume and folder names. If you pass an empty string, the standard Save File dialog box is displayed. If the user cancels this dialog, the export operation is canceled, and the OK system variable is set to 0.

A progress thermometer is displayed during export. The user can cancel the operation by clicking a Stop button. If the export is successfully completed, the OK system variable is set to 1. If the operation is canceled or an error occurs, the OK system variable is set to 0. The thermometer can be hidden with the MESSAGES OFF command.

The export operation is made using the default ASCII map for the platform on which it is executed, unless you change the ASCII map (using the command USE ASCII MAP) prior to the export. An ASCII map can be used to convert the data for use on platforms that have a different ASCII table.

Example

The following example exports data to a DIF document. The method first sets the output form so that the data will be exported through the correct form, then performs the export:

```
	OUTPUT FORM([People];"Export")
⇒	EXPORT DIF([People];"NewPeople") ` Export to the "NewPeople" document
```

See Also

EXPORT SYLK, EXPORT TEXT, IMPORT DIF, USE ASCII MAP.

System Variables and Sets

OK is set to 1 if the export is successfully completed; otherwise, it is set to 0.

21 Interruptions

ON EVENT CALL

ON EVENT CALL (eventMethod{; processName})

Parameter	Type		Description
eventMethod	String	→	Event method to be invoked, or Empty string to stop intercepting events
processName	String	→	Process name

Description

The command ON EVENT CALL installs the method, whose name you pass in eventMethod, as the method for catching (trapping) events. This method is called the **event-handling method** or **event-catching method**..

Tip: This command requires advanced programming knowledge. Usually, you do not need to use ON EVENT CALL for working with events. While using forms, 4th Dimension handles the events and sends them to the appropriate forms and objects.

Tip: Version 6 introduces new commands, such as GET MOUSE, Shift down, etc., for getting information about events. These commands can be called from within object methods to get the information you need about an event involving an object. Using them spares you the writing of an algorithm based on the ON EVENT CALL scheme.

The scope of this command is the current working session. By default, the method is run in a separate local process. You can have only one event-handling method at a time. To stop catching events with a method, call ON EVENT CALL again and pass the empty string in eventMethod.

As the event-handling method is run in a separate process, it is constantly active, even if no 4th Dimension method is running. After installation, 4th Dimension calls the event-handling method each time an event occurs. An event can be a mouse click or a keystroke.

The optional processName parameter names the process created by the ON EVENT CALL command. If processName is prefixed with a dollar sign ($), a local process is started, which is usually what you want. If you omit the processName parameter, 4D creates, by default, a local process named $Event Manager.

WARNING: Be very careful in what you do within an event-handling method. Do NOT call commands that generate events, otherwise it will be extremely difficult to get out of the event-handling method execution. The key combination Ctrl+Shift+Backspace (on Windows) or Command-Shift-Option-Control-Backspace (on Macintosh) converts the Event Manager process into a normal process. This means that the method will no longer be automatically passed all the events that occur. You may want to use this technique to recover an event-handing gone wrong (i.e., one that has bugs triggering events).

In the event-handling method, you can read the following system variables—MouseDown, KeyCode, Modifiers, MouseX, MouseY, and MouseProc. Note that these variables are process variables. Their scope is therefore the event-handling process. Copy them into interprocess variables if you want their values available in another process.

• The MouseDown system variable is set to 1 if the event is a mouse click, and to 0 if it is not.
• The KeyCode system variable is set to the ASCII code for a keystroke. This variables may return an ASCII code or a function key code. These codes are listed in the sections ASCII Codes (and its subsections) and Function Key Codes. 4D provides predefined constants for the major ASCII Codes and for Function Key Codes. In the Explorer window, look for the themes of these constants.
• The Modifiers system variable contains the modifier value. The Modifiers system variable indicates whether any of the following modifier keys were down when the event occurred:

Platform	**Modifiers**
Windows	Shift key, Caps Lock, Alt key, Ctrl key, Right mouse button
Macintosh	Shift key, Caps Lock, Option key, Command key, Control key

Notes
- The Windows ALT key is equivalent to the Macintosh Option key
- The Windows Ctrl key is equivalent to the Macintosh Command key
- The Macintosh Control key has no equivalent on Windows. However, a right mouse button click on Windows, is equivalent to a Control-Click on Macintosh.

The modifier keys do not generate an event; another key or the mouse button must also be pressed. The Modifiers variable is a 4-byte Long Integer variable that should be seen as an array of 32 bits. 4D provides predefined constants expressing bit positions or bit masks for testing the bit corresponding to each modifier key. For example, to detect if the Shift key was pressed for the event, you can write:

```
	If (Modifiers ?? Shift key bit ) ` If the Shift key was down
```
or:
```
	If ((Modifiers & Shift key mask)#0)` If the Shift key was down
```

• The system variables MouseX and MouseY contain the horizontal and vertical positions of the mouse click, expressed in the local coordinate system of the window where the click occurred. The upper left corner of the window is position 0,0. These are meaningful only when there is a mouse click.

• The MouseProc system variable contains the process reference number of the process in which the event occurred, whatever its nature—mouse click or keystroke.

Important: The system variables MouseDown, KeyCode, Modifiers, MouseX, MouseY, and MouseProc contain significant values only within an event-hanlding method installed with ON EVENT CALL.

Example

This example will cancel printing if the user presses Ctrl-period. First, the event-handling method is installed. Then a message is displayed, announcing that the user can cancel printing. If the interprocess variable <>vbWeStop is set to True in the event-handling method, the user is alerted to the number of records that have already been printed. Then the event-handling method is deinstalled:

```
      PAGE SETUP
      If (OK=1)
         <>vbWeStop:=False
⇒        ON EVENT CALL("EVENT HANDLER") ` Installs the event-handling method
         ALL RECORDS([People])
         MESSAGE("To interrupt printing press Ctrl-Period")
         $vlNbRecords:=Records in selection([People])
         For ($vlRecord;1;$vlNbRecords)
            If (<>vbWeStop)
               ALERT("Printing cancelled at record "+String($vlRecord)
                                                      +" of "+String($vlNbRecords))
               $vlRecord:=$vlNbRecords+1
            Else
               PRINT FORM([People];"Special Report")
            End if
         End for
         PAGE BREAK
⇒        ON EVENT CALL("") ` Deinstalls the event-handling method
      End if
```

If Ctrl-period has been pressed, the event-handling method sets <>vbWeStop to True:

```
    ` EVENT HANDLER project method
If ((Modifiers ?? Command key bit) & (KeyCode = Period))
    CONFIRM("Are you sure?")
    If (OK=1)
        <>vbWeStop:=True
        FILTER EVENT ` Do NOT forget this call otherwise 4D will also get this event
    End if
End if
```

Note that this example uses ON EVENT CALL, because it performs a special printing report using the commands PAGE SETUP, PRINT FORM and PAGE BREAK with a For loop.

If you print a report using PRINT SELECTION, you do NOT need to handle events that let the user interrupt the printing; PRINT SELECTION does that for you.

See Also

FILTER EVENT, GET MOUSE, Shift down.

FILTER EVENT

FILTER EVENT

Parameter	Type	Description
This command does not require any parameters		

Description

You call the FILTER EVENT command from within an event-handling project method installed using the ON EVENT CALL command.

If an event-handling method calls FILTER EVENT, the current event is not passed to 4D.

This command allows you to remove the current event (i.e., click, keystroke) from the event queue, so 4D will not perform any additional treatment to the one you made in the event-handling project method.

WARNING: Avoid creating an event-handling method that only calls the FILTER EVENT command, because all the events are going to be ignored by 4D. In case you have an event-handling method with only the FILTER EVENT command, type Ctrl+Shift+Backspace (on Windows) or Command-Option-Shift-Control-Backspace (on Macintosh). This converts the On Event Call process into a normal process that does not get any events at all.

Example

See example for the command ON EVENT CALL.

See Also

ON EVENT CALL.

ON ERR CALL

version 3

ON ERR CALL (errorMethod)

Parameter	Type		Description
errorMethod	String	→	Error method to be invoked, or Empty string to stop trapping errors

Description

The command ON ERR CALL installs the project method, whose name you pass in errorMethod, as the method for catching (trapping) errors. This project method is called the **error-handling method** or **error-catching method**.

The scope of this command is the current process. You can have only one error-handling method per process at a time, but you can have different error-handling methods for several processes.

To stop the trapping of errors, call ON ERR CALL again and pass the empty string in errorMethod.

Once an error-handling project is installed, 4th Dimension calls the method each time an error occurs.

You can identify errors by reading the Error system variable, which contains the code number of the error. Error codes are listed in the theme Error codes. For more information, see the section Syntax Errors or Database Engine Errors. The Error variable value is significant only within the error-handling method; if you need the error code within the method that provoked the error, copy the Error variable to your own process variable.

The error-handling method should manage the error in an appropriate way or present an error message to the user. Errors can be generated by:
• The 4th Dimension database engine; for example, when saving a record tries to duplicate a unique index key.
• The 4th Dimension environment; for example, when you do not have enough memory for allocating an array.
• The operating system on which the database is runs; for example, disk full or I/O errors.

The ABORT command can be used to terminate processing. If you don’t call ABORT in the error-handling method, 4th Dimension returns to the interrupted method and continues to execute the method. Use the ABORT command when an error cannot be recovered.

If an error occurs in the error-handling method itself, 4th Dimension takes over error handling. Therefore, you should make sure that the error-handling method cannot generate an error. Also, you cannot use ON ERR CALL inside the error-handling method.

When an ON ERR CALL error-handling method is installed, it is not possible to trace a method by using Alt+Click (on Windows) or Option-Click (on Macintosh). This is because Alt+Click and Option-Click) generate an error (error code 1006) that immediately activates the ON ERR CALL error-handling method. However, you can test this error code by calling TRACE.

Examples

1. The following project method tries to create a document whose name is received as parameter. If the document cannot be created, the project metod returns 0 (zero) or the error code:

```
     ` Create doc project method
     ` Create doc ( String ; Pointer ) -> LongInt
     ` Create doc ( DocName ; ->DocRef ) -> Error code result

  gError:=0
⇒ ON ERR CALL("IO ERROR HANDLER")
  $2->:=Create document($1)
⇒ ON ERR CALL("")
  $0:=gError
```

The IO ERROR HANDLER project method is listed here:

```
     ` IO ERROR HANDLER project method
  gError:=Error ` just copy the error code to the process variable gError
```

Note the use of the gError process variable to get the error code result within the current executing method. Once these methods are present in your database, you can write:

```
     ` ...
  C_TIME(vhDocRef)
  $vlErrCode:=Create doc($vsDocumentName;->vhDocRef)
  If ($vlErrCode=0)
        `...
     CLOSE DOCUMENT($vlErrCode)
  Else
     ALERT ("The document could not be created, I/O error "+String($vlErrCode))
  End if
```

2. See example in the section Arrays and Memory.

3. While implementing a complex set of operations, you may end up with various subroutines that require different error-handling methods. You can have only one error-handling method per process at a time, so you have two choices:
- Keep track of the current one each time you call ON ERR CALL, or
- Use a process array variable (in this case, asErrorMethod) to “pile up” the error-handling methods and a project method (in this case, ON ERROR CALL) to install and deinstall the error-handling methods.

You must initialize the array at the very beginning of the process execution:

```
    ` Do NOT forget to initialize the array at the beginning
    ` of the process method (the project method that runs the process)
ARRAY STRING(63;asErrorMethod;0)
```

Here is the custom ON ERROR CALL method:

```
    ` ON ERROR CALL project method
    ` ON ERROR CALL { ( String ) }
    ` ON ERROR CALL { ( Method Name ) }

C_STRING(63;$1;$ErrorMethod)
C_LONGINT($vlElem)

If (Count parameters>0)
   $ErrorMethod:=$1
Else
   $ErrorMethod:=""
End if

If ($ErrorMethod#"")
   C_LONGINT(gError)
   gError:=0
   $vlElem:=1+Size of array(asErrorMethod)
   INSERT ELEMENT(asErrorMethod;$vlElem)
   asErrorMethod{$vlElem}:=$1
   ON ERR CALL($1)
Else
   ON ERR CALL("")
   $vlElem:=Size of array(asErrorMethod)
   If ($vlElem>0)
      DELETE ELEMENT(asErrorMethod;$vlElem)
      If ($vlElem>1)
         ON ERR CALL(asErrorMethod{$vlElem-1})
      End if
   End if
End if
```

Then, you can call it this way:

```
	gError:=0
	ON ERROR CALL("IO ERRORS") ` Installs the IO ERRORS error-handling method
	   ` ...
	ON ERROR CALL("ALL ERRORS") ` Installs the ALL ERRORS error-handling method
	   ` ...
	ON ERROR CALL   ` Deinstalls the ALL ERRORS error-handling method and reinstalls IO
ERRORS
	   ` ...
	ON ERROR CALL   ` Deinstalls the IO ERRORS error-handling method
	   ` ...
```

4. The following error-handling method ignores the user interruptions:

```
	   ` SHOW ONLY ERRORS project method
	If (Error#1006)
	   ALERT ("The error "+String(Error)+" occurred.")
	End if
```

See Also

ABORT.

ABORT

version 3

Note: You will rarely call this command.

ABORT

Parameter	**Type**	**Description**

This command does not require any parameters

Description

The command ABORT is to be used from within an error-handling project method installed using the command ON ERR CALL.

If you do not have an error-handling project method, when an error occurs (for example, a database engine error) 4D displays its standard error dialog box and then interrupts the execution of your code. If the code being executed is:
• An object method, form method (or a project method called by a form or object method), the control returns to the form currently being displayed.
• A method called from a menu, the control returns to the menu bar or to the form currently being displayed.
• The master method of a process, the process then ends.
• A method called directly or indirectly by an import or export operation, the operation is stopped. The same is true for sequential queries or order by operations.
• And so on...

If you use an error-handling project method to catch errors, 4D neither displays its standard error dialog box nor interrupts the execution of your code. Instead, 4D calls your error-handling project method (that you can see as an exception handler), and resumes the execution to the next line of code in the method that triggered the error.

There are errors you can treat programmatically; for example, during an import operation, if you catch a database engine duplicated value error, you can “cover” the error and pursue the import. However, there are errors that you cannot process and errors that you should not “cover.” In these cases, you need to stop the execution by calling ABORT from within the error-handling project method.

Historical Note

Although the ABORT command is intended to be used only from within a error-handling project method, some members of the 4D community also use it to interrupt execution in other project methods. The fact that it works is only a side effect. We do not recommend the use of this command in methods other than error-handling methods.

22 Language

Count parameters

Language

version 3

Count parameters → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of parameters actually passed

Description

The command Count parameters returns the number of parameters passed to a project method.

WARNING: Count parameters is meaningful only in a project method that has been called by another method (project method or other). If the project method calling Count parameters is associated with a menu, Count parameters returns 0.

Examples

1. 4th Dimension project methods accept optional parameters, starting from the right. For example, you can call the method MyMethod(a;b;c;d) in the following ways:

```
MyMethod ( a ; b ; c ; d )  ` All parameters are passed
MyMethod ( a ; b ; c )  ` The last parameter is not passed
MyMethod ( a ; b )  ` The last two parameters are not passed
MyMethod ( a )  ` Only the first parameter is passed
MyMethod  ` No Parameter is passed at all
```

Using Count parameters from within MyMethod, you can detect the actual number of parameters and perform different operations depending on what you have received. The following example displays a text message and can insert the text into a 4D Write area or send the text into a document on disk:

```
     ` APPEND TEXT Project Method
     ` APPEND TEXT ( Text { ; Long { ; Time } } )
     ` APPEND TEXT ( Text { ; 4D Write Area { ; DocRef } } )

  C_TEXT ($1)
  C_TIME ($2)
  C_LONGINT ($3)

  MESSAGE ($1)
⇒ If (Count parameters>=3)
     SEND PACKET ($3;$1)
  Else
```

```
⇒         If (Count parameters>=2)
              WR INSERT TEXT ($2;$1)
          End if
      End if
```

After this project method has been added to your application, you can write:

```
      APPEND TEXT (vtSomeText)  ` Will only display the text message
      APPEND TEXT (vtSomeText;$wrArea)  ` Will display the text message and append it to
$wrArea
      APPEND TEXT (vtSomeText;0;$vhDocRef)  ` Will display the text message and write it to
$vhDocRef
```

2. 4th Dimension project methods accept a variable number of parameters of the same type, starting from the right. To declare these parameters, you use a compiler directive to which you pass ${N} as a variable, where N specifies the first parameter. Using Count parameters you can address those parameters with a For loop and the parameter indirection syntax. This example is a function that returns the greatest number received as parameter:

```
          ` Max of Project Method
          ` Max of ( Real { ; Real2... ; RealN } ) -> Real
          ` Max of ( Value { ; Value2... ; ValueN } ) -> Greatest value

      C_REAL ($0;${1})  ` All parameters will be of type REAL as well as the function result
      $0:=${1}
⇒     For ($vlParam;2;Count parameters)
          If (${$vlParam}>$0)
              $0:=${$vlParam}
          End if
      End for
```

After this project method has been added to your application, you can write:

```
      vrResult:=Max of (Records in set("Operation A");Records in set("Operation B"))
```

or:

```
      vrResult:=Max of (r1;r2;r3;r4;r5;r6)
```

See Also

Compiler commands, C_BLOB, C_BOOLEAN, C_DATE, C_GRAPH, C_INTEGER, C_LONGINT, C_PICTURE, C_POINTER, C_REAL, C_STRING, C_TEXT, C_TIME.

Type

version 6.0 (Modified)

Type (fieldVar) → Number

Parameter	Type		Description
fieldVar	Field | Variable	→	Field or Variable to be tested
Function result	Number	←	Data type number

Description

The command Type returns a numeric value that denotes the type of the field or variable you pass as fieldVar.

4th Dimension provides the following predefined constants:

Constant	Type	Value
Is Alpha Field	Long Integer	0
Is String Var	Long Integer	24
Is Text	Long Integer	2
Is Real	Long Integer	1
Is Integer	Long Integer	8
Is LongInt	Long Integer	9
Is Date	Long Integer	4
Is Time	Long Integer	11
Is Boolean	Long Integer	6
Is Picture	Long Integer	3
Is Subtable	Long Integer	7
Is BLOB	Long Integer	30
Is Undefined	Long Integer	5
Is Pointer	Long Integer	23
String array	Long Integer	21
Text array	Long Integer	18
Real array	Long Integer	14
Integer array	Long Integer	15
LongInt array	Long Integer	16
Date array	Long Integer	17
Boolean array	Long Integer	22
Picture array	Long Integer	19
Pointer array	Long Integer	20
Array 2D	Long Integer	13

Compatibility Note: In previous versions of 4D, Type returned 3 (Is Picture) when applied to a Graph variable declared using the command C_GRAPH. Starting with version 6, Type returns 9 (Is LongInt) when applied to a Graph variable.

You can apply Type to fields, interprocess variables, process variables, local variables, and dereferenced pointers referring to these types of objects.

Version 6 Note: Starting with version 6, you can apply Type to parameters ($1,$2..., ${...}), or to project method or function results ($0).

Examples

1. See example for the APPEND TO CLIPBOARD command.

2. See example for DRAG AND DROP PROPERTIES command.

3. The following project method empties some or all of the fields for the current record of the table whose a pointer is passed as parameter. It does this without deleting or changing the current record:

```
      ` EMPTY RECORD Project Method
      ` EMPTY RECORD ( Pointer {; Long } )
      ` EMPTY RECORD ( -> [Table] { ; Type Flags } )

C_POINTER ($1)
C_LONGINT ($2;$vlTypeFlags)

If (Count parameters>=2)
    $vlTypeFlags:=$2
Else
    $vlTypeFlags:=0xFFFFFFFF
End if
For ($vlField;1;Count fields($1))
    $vpField:=Field(Table($1);$vlField)
    $vlFieldType:=Type($vpField->)
    If ( $vlTypeFlags ?? $vlFieldType )
        Case of
            : (($vlFieldType=Is Alpha Field)|($vlFieldType=Is Text))
                $vpField->:=""
            : (($vlFieldType=Is Real)|($vlFieldType=Is Integer)|($vlFieldType=Is LongInt))
                $vpField->:=0
            : ($vlFieldType=Is Date)
                $vpField->:=!00/00/00!
            : ($vlFieldType=Is Time)
                $vpField->:=?00:00:00?
            : ($vlFieldType=Is Boolean)
                $vpField->:=False
            : ($vlFieldType=Is Picture)
                C_PICTURE($vgEmptyPicture)
                $vpField->:=$vgEmptyPicture
            : ($vlFieldType=Is Subtable)
                Repeat
                    ALL SUBRECORDS($vpField->)
                    DELETE SUBRECORD($vpField->)
                Until(Records in subselection($vpField->)=0)
            : ($vlFieldType=Is BLOB)
                SET BLOB SIZE($vpField->;0)
        End case
    End if
End for
```

After this project method is implemented in your database, you can write:

```
    ` Empty the whole current record of the table [Things To Do]
EMPTY RECORD (->[Things To Do])

    ` Empty Text, BLOB and Picture fields for the current record
    ` of the table [Things To Do]
EMPTY RECORD (->[Things To Do]; 0 ?+ Is Text ?+ Is BLOB ?+ Is Picture )

    ` Empty the whole current record of the table [Things To Do]
    ` except Alphanumeric fields
EMPTY RECORD (->[Things To Do]; -1 ?- Is Alpha Field )
```

See Also

Is a variable, Undefined.

Self

Self → Pointer

Parameter	Type		Description
This command does not require any parameters			
Function result	Pointer	←	Pointer to form object (if any) whose method is currently being executed. Otherwise Nil (->[]) if outside of context

Description

The command Self returns a pointer to the object whose object method is currently being executed.

Self is used to reference a variable within its own object method. It returns a valid pointer only when it is called from within an object method. It cannot be used in a project method, even when called from an object method. If Self is called out of context, it returns a Nil pointer (->[]).

Tip: Self is useful when several objects on a form need to perform the same task, yet operate on themselves.

Example

See the example for the RESOLVE POINTER command.

See Also

RESOLVE POINTER.

RESOLVE POINTER

RESOLVE POINTER (pointer; varName; tableNum; fieldNum)

Parameter	Type		Description
pointer	Pointer	→	Pointer for which to retrieve the referenced object
varName	String	←	Name of referenced variable or empty string
tableNum	Number	←	Number of referenced table or array element or 0 or -1
fieldNum	Number	←	Number of referenced field or 0

Description

The command RESOLVE POINTER retrieves the information of the object referenced by the pointer expression pointer and returns it into the parameters varName, tableNum, and fieldNum.

Depending on the nature of the referenced object, RESOLVE POINTER returns the following values:

Referenced object	Parameters		
	varName	*tableNum*	*fieldNum*
None (NIL pointer)	"" (empty string)	0	0
Variable	Name of the variable	-1	0
Array	Name of the array	-1	0
Array element	Name of the array	Element number	0
Table	"" (empty string)	Table number	0
Field	"" (empty string)	Table number	Field number

Note: If the value you pass in pointer is not a pointer expression, a syntax error occurs.

Examples

1. Within a form, you create a group of 100 enterable variables called v1, v2... v100. To do so, you perform the following steps:

a. Create one enterable variable that you name v.

b. Set the properties of the object.

c. Attach the following method to that object:

```
DoSomething (Self)  ` DoSomething being a project method in your database
```

d. At this point, you can either duplicate the variable as many times as you need, or use the Objects on Grid feature in the Form Editor.

e. Within the DoSomething method, if you need to know the index of the variable for which the method is called, you write:

```
⇒      RESOLVE POINTER($1;$vsVarName;$vlTableNum;$vlFieldNum)
       $vlVarNum:=Num(Substring($vsVarName;2))
```

Note that by constructing your form in this way, you write the methods for the 100 variables only once; you do not need to write DoSomething (1), DoSomething (2)...,DoSomething (100).

2. For debugging purposes, you need to verify that the second parameter ($2) to a method is a pointer to a table. At the beginning of this method, you write:

```
          ` ...
       If (<>DebugOn)
⇒         RESOLVE POINTER($2;$vsVarName;$vlTableNum;$vlFieldNum)
          If (Not(($vlTableNum>0)&($vlFieldNum=0)&($vsVarName="")))
          ` WARNING: The pointer is not a reference to a table
             TRACE
          End
       End if
       ` ...
```

3. See example for the DRAG AND DROP PROPERTIES command.

See Also

DRAG AND DROP PROPERTIES, Field, Get pointer, Is a variable, Nil, Table.

Nil

Nil (aPointer) → Boolean

Parameter	Type		Description
aPointer	Pointer	→	Pointer to be tested
Function result	Boolean	←	TRUE = Nil pointer (->[]) FALSE = Valid pointer to an existing object

Description

The command Nil returns True if the pointer you pass in aPointer is Nil (->[]). It returns False in all other cases (pointer to field, table or variable).

Starting with version 6, instead of using Nil, it will be more convenient to use RESOLVE POINTER, which tells you about the nature of the referenced object, no matter what the object is (including Nil pointers).

See Also

Is a variable, RESOLVE POINTER.

Is a variable

Language

version 3

Is a variable (aPointer) → Boolean

Parameter	Type		Description
aPointer	Pointer	→	Pointer to be tested
Function result	Boolean	←	TRUE = Pointer points to a variable FALSE = Pointer does not point to a variable

Description

The command Is a variable returns True if the pointer you pass in aPointer references a defined variable. It returns False in all other cases (pointer to field or table, Nil pointer, and so on).

Starting with version 6, instead of using Is a variable, it will be more convenient to use RESOLVE POINTER, which tells you about the nature of the referenced object, no matter what the object is (including the case of Nil pointers).

See Also

Nil, RESOLVE POINTER.

Get pointer

Get pointer (varName) → Pointer

Parameter	Type		Description
varName	String	→	Name of a process variable
Function result	Pointer	←	Pointer to process variable

Description

The command Get pointer returns a pointer to the variable whose name you pass in varName.

To get a pointer to a field, use Field. To get a pointer to a table, use Table.

Example

In a form, you build a 5 x 10 grid of enterable variables named v1, v2... v50. To initialize all of these variables, you write:

```
          ` ...
       For ($vlVar;1;50)
⇒          $vpVar:=Get pointer("v"+String($vlVar))
           $vpVar->:=""
       End for
```

See Also

Field, Table.

EXECUTE

Language

version 3

Note: You will rarely need to use this command.

EXECUTE (statement)

Parameter	**Type**		**Description**
statement	String	→	Code to be executed

Description

EXECUTE executes statement as a line of code. The statement string must be one line. If statement is an empty string, EXECUTE does nothing.

The rule of thumb is that if the statement can be executed as a one line method, then it will execute properly.

In a compiled database, the line of code is not compiled. This means that statement will be executed, but it will not have been checked by 4D Compiler at compilation time.

Use EXECUTE sparingly, as it slows down execution speed.

The statement can be in the following:
• a Call to a project method
• a Call to a 4D command
• an Assignment

The statement can include process variables and interprocess variables. The statement cannot contain control of flow statements, because it must be in one line of code.

Example

See examples for the Command Name command.

See Also

Command name.

Command name

Command name (command) → String

Parameter	Type		Description
command	Number	→	Command number
Function result	String	←	Localized command name

Description

The command Command name returns the literal name of the command whose command number you pass in command.

4th Dimension integrates a dynamic translation of the keywords, constants, and command names used in your methods. For example, if you use the English version of 4D, you write:

```
DEFAULT TABLE ([MyTable])
ALL RECORDS ([MyTable])
```

This same code, reopened with the French version of 4D, will read:

```
TABLE PAR DEFAUT ([MyTable])
TOUT SELECTIONNER ([MyTable])
```

However, 4th Dimension also includes a unique feature, the EXECUTE command, which allows you to build code on the fly and then execute this code, even though the database is compiled.

The example code, written with EXECUTE statements in English, looks like:

```
EXECUTE ( "DEFAULT TABLE([MyTable])")
EXECUTE ( "ALL RECORDS([MyTable])")
```

This same code, reopened with the French version of 4D, will then read:

```
EXECUTER ( "DEFAULT TABLE([MyTable])")
EXECUTER ( "ALL RECORDS([MyTable])")
```

4D automatically translates EXECUTE (English) to EXECUTER (French), but cannot translate the text statement you passed to the command.

If you use the EXECUTE command in your application, you can use Command name to eliminate international localization issues for statements you execute in this way, and thus make your statements independent of language.

The example code becomes:

```
⇒      EXECUTE (Command name (46)+"([MyTable])")
⇒      EXECUTE (Command name (47)+"([MyTable])")
```

With a French version of 4D, this code will read:

```
⇒      EXECUTER (Nom commande (46)+"([MyTable])")
⇒      EXECUTER (Nom commande (47)+"([MyTable])")
```

Note: Command names and numbers are listed in Commands by name and Commands by number.

Examples

1. For all the tables of your database, you have a form called “INPUT FORM” used for standard data entry in each table. Then, you want to add a generic project method that will set this form as the current input form for the table whose pointer or name you pass. You write:

```
           ` STANDARD INPUT FORM project method
           ` STANDARD INPUT FORM ( Pointer {; String })
           ` STANDARD INPUT FORM ( ->Table {; TableName })
       C_POINTER ($1)
       C_STRING (31;$2)

       If (Count parameters>=2)
⇒         EXECUTE (Command name (55)+"(["+$2+"];"INPUT FORM")")
       Else
          If (Count parameters>=1)
             INPUT FORM ($1->;"INPUT FORM")
          End if
       End if
```

After this project method has been added to your database, you write:

```
       STANTARD INPUT FORM (->[Employees])
       STANTARD INPUT FORM ("Employees")
```

Note: Usually, it is better to use pointers when writing generic routines. First, the code will run compiled if the database is compiled. Second, 4D Insider will retrieve the references to the object whose pointer you pass. Third, as in the previous example, your code can cease to work correctly if you rename the table. However, in certain cases, using EXECUTE will solve the problem.

2. In a form, you want a drop-down list populated with the basic summary report commands. In the object method for that drop-down list, you write:

```
        Case of
           : (Form event =On Before)
              ARRAY TEXT (asCommand;4)
⇒             asCommand{1}:=Command name (1) ` Sum
⇒             asCommand{2}:=Command name (2) ` Average
⇒             asCommand{3}:=Command name (4) ` Min
⇒             asCommand{4}:=Command name (3) ` Max
           ` ...
        End case
```

In the English version of 4D, the drop-down list will read: Sum, Average, Min, and Max. In the French version, the drop-down list will read: Somme, Moyenne, Min, and Max.

See Also

Commands by Name, Commands by Number, EXECUTE.

Commands by Name

This table lists the 4D commands by name, and their numbers. Command numbers must be used with the command Command Name.

A

ABORT	156
Abs	99
ACCEPT	269
ACCUMULATE	303
ACI folder	485
Activated	346
ADD DATA SEGMENT	361
ADD RECORD	56
ADD SUBRECORD	202
Add to date	393
ADD TO SET	119
After	31
ALERT	41
ALL RECORDS	47
ALL SUBRECORDS	109
Append document	265
APPEND MENU ITEM	411
APPEND TO CLIPBOARD	403
APPEND TO LIST	376
Application file	491
Application type	494
Application version	493
APPLY TO SELECTION	70
APPLY TO SUBSELECTION	73
Arctan	20
ARRAY BOOLEAN	223
ARRAY DATE	224
ARRAY INTEGER	220
ARRAY LONGINT	221
ARRAY PICTURE	279
ARRAY POINTER	280
ARRAY REAL	219
ARRAY STRING	218
ARRAY TEXT	222
ARRAY TO LIST	287
ARRAY TO SELECTION	261
ARRAY TO STRING LIST	512
Ascii	91

Commands by Number

Language

version 6.0

This table lists the 4D commands by number, with their names. Command numbers must be used with the command Command Name.

Note: Unlisted numbers are currently not used.

1	Sum
2	Average
3	Max
4	Min
5	PRINT FORM
6	PAGE BREAK
7	Records in subselection
8	Int
9	Dec
10	String
11	Num
12	Substring
13	Uppercase
14	Lowercase
15	Position
16	Length
17	Sin
18	Cos
19	Tan
20	Arctan
21	Exp
22	Log
23	Day of
24	Month of
25	Year of
26	Std deviation
27	Variance
28	Sum squares
29	Before
30	During
31	After
32	Modified
33	Current date
34	Not
35	Old
36	End selection
37	End subselection
38	REJECT

39	PRINT LABEL
40	DIALOG
41	ALERT
42	RELATE ONE
43	SAVE RELATED ONE
44	OLD RELATED ONE
45	SAVE OLD RELATED ONE
46	DEFAULT TABLE
47	ALL RECORDS
48	QUERY BY FORMULA
49	ORDER BY
50	FIRST RECORD
51	NEXT RECORD
52	LOAD RECORD
53	SAVE RECORD
54	OUTPUT FORM
55	INPUT FORM
56	ADD RECORD
57	MODIFY RECORD
58	DELETE RECORD
59	DISPLAY SELECTION
60	PRINT SELECTION
61	FIRST SUBRECORD
62	NEXT SUBRECORD
63	EXECUTE
64	SEARCH BY INDEX
65	CREATE RELATED ONE
66	DELETE SELECTION
67	MENU BAR
68	CREATE RECORD
70	APPLY TO SELECTION
71	PRINT RECORD
72	CREATE SUBRECORD
73	APPLY TO SUBSELECTION
74	LOAD VARIABLES
75	SAVE VARIABLES
76	Records in selection
77	SET CHANNEL
78	SEND RECORD
79	RECEIVE RECORD
80	SEND VARIABLE
81	RECEIVE VARIABLE
82	Undefined
83	Records in table
84	EXPORT DIF
85	EXPORT SYLK
86	IMPORT DIF
87	IMPORT SYLK

88	MESSAGE
89	CLEAR VARIABLE
90	Char
91	Ascii
93	INVERT BACKGROUND
94	Round
95	Trunc
96	DELETE SUBRECORD
97	Subtotal
98	Mod
99	Abs
100	Random
101	Level
102	Date
103	SEND PACKET
104	RECEIVE PACKET
105	DISPLAY RECORD
106	PRINT SETTINGS
107	ORDER SUBRECORDS BY
108	QUERY SUBRECORDS
109	ALL SUBRECORDS
110	PREVIOUS RECORD
111	PREVIOUS SUBRECORD
112	In header
113	In break
114	Day number
116	CREATE SET
117	CLEAR SET
118	USE SET
119	ADD TO SET
120	UNION
121	INTERSECTION
122	DIFFERENCE
140	CREATE EMPTY SET
143	Semaphore
144	CLEAR SEMAPHORE
145	READ ONLY
146	READ WRITE
147	Locked
148	GRAPH TABLE
149	ENABLE MENU ITEM
150	DISABLE MENU ITEM
151	BEEP
152	Menu selected
153	Open window
154	CLOSE WINDOW
155	ON ERR CALL
156	ABORT

157	TRACE
158	NO TRACE
159	DELETE DOCUMENT
160	ERASE WINDOW
161	GOTO XY
162	CONFIRM
163	Request
164	FONT
165	FONT SIZE
166	FONT STYLE
167	EXPORT TEXT
168	IMPORT TEXT
169	GRAPH
170	SORT BY INDEX
171	ON SERIAL PORT CALL
172	RECEIVE BUFFER
174	REDRAW
175	MESSAGES OFF
176	PUSH RECORD
177	POP RECORD
178	Current time
179	Time
180	Time string
181	MESSAGES ON
182	Current user
184	SAVE SET
185	LOAD SET
186	CHANGE PASSWORD
187	Screen width
188	Screen height
189	ONE RECORD SELECT
190	ON EVENT CALL
191	In footer
192	ENABLE BUTTON
193	DISABLE BUTTON
194	BUTTON TEXT
195	Records in set
197	REPORT
198	Before selection
199	Before subselection
200	LAST RECORD
201	LAST SUBRECORD
202	ADD SUBRECORD
203	MODIFY SUBRECORD
204	MODIFY SELECTION
205	USE ASCII MAP
206	GOTO AREA
207	QUERY SELECTION BY FORMULA

208	SET MENU ITEM MARK
209	GET HIGHLIGHT
210	HIGHLIGHT TEXT
212	UNLOAD RECORD
213	SET WINDOW TITLE
214	True
215	False
218	ARRAY STRING
219	ARRAY REAL
220	ARRAY INTEGER
221	ARRAY LONGINT
222	ARRAY TEXT
223	ARRAY BOOLEAN
224	ARRAY DATE
225	DUPLICATE RECORD
226	COPY ARRAY
227	INSERT ELEMENT
228	DELETE ELEMENT
229	SORT ARRAY
230	Find in array
231	Insert string
232	Delete string
233	Replace string
234	Change string
235	SET FILTER
236	SET FORMAT
237	SET CHOICE LIST
238	SET ENTERABLE
239	START TRANSACTION
240	VALIDATE TRANSACTION
241	CANCEL TRANSACTION
242	GOTO RECORD
243	Record number
244	Sequence number
245	GOTO SELECTED RECORD
246	Selected record number
247	GOTO PAGE
248	NEXT PAGE
249	PREVIOUS PAGE
250	FIRST PAGE
251	LAST PAGE
252	Table
253	Field
254	Count tables
255	Count fields
256	Table name
257	Field name
258	GET FIELD PROPERTIES

259	Count parameters
260	SELECTION TO ARRAY
261	ARRAY TO SELECTION
262	RELATE MANY
263	OLD RELATED MANY
264	Open document
265	Append document
266	Create document
267	CLOSE DOCUMENT
268	SET TIMEOUT
269	ACCEPT
270	CANCEL
271	SET COLOR
273	Is in set
274	Size of array
275	Printing page
276	Current form page
277	QUERY
278	Last object
279	ARRAY PICTURE
280	ARRAY POINTER
281	EDIT ACCESS
282	C_INTEGER
283	C_LONGINT
284	C_TEXT
285	C_REAL
286	C_PICTURE
287	ARRAY TO LIST
288	LIST TO ARRAY
289	CHANGE ACCESS
290	PLAY
291	QUIT 4D
292	QUERY BY EXAMPLE
293	C_STRING
294	Is a variable
295	Type
297	FLUSH BUFFERS
298	GRAPH SETTINGS
299	PAGE SETUP
300	ORDER BY FORMULA
301	C_POINTER
302	BREAK LEVEL
303	ACCUMULATE
304	Get pointer
305	C_BOOLEAN
306	C_TIME
307	C_DATE
308	Self

309	Open external window
310	AUTOMATIC RELATIONS
311	IDLE
314	Modified record
315	Nil
316	SET ABOUT
317	New process
319	PAUSE PROCESS
320	RESUME PROCESS
321	FILTER EVENT
322	Current process
323	DELAY PROCESS
324	HIDE PROCESS
325	SHOW PROCESS
326	BRING TO FRONT
327	Frontmost process
328	Outside call
329	CALL PROCESS
330	Process state
331	COPY NAMED SELECTION
332	USE NAMED SELECTION
333	CLEAR NAMED SELECTION
334	CUT NAMED SELECTION
335	Count tasks
336	PROCESS PROPERTIES
338	User in group
339	DISTINCT VALUES
340	RELATE MANY SELECTION
341	QUERY SELECTION
342	Count users
343	Count user processes
344	SET INDEX
345	SELECT LOG FILE
346	Activated
347	Deactivated
348	SET MENU ITEM
349	RELATE ONE SELECTION
350	SCAN INDEX
351	REDUCE SELECTION
352	C_GRAPH
353	LOCKED ATTRIBUTES
355	COMPRESS PICTURE
356	Picture size
357	LOAD COMPRESS PICTURE FROM FILE
358	SAVE PICTURE TO FILE
359	COMPRESS PICTURE FILE
361	ADD DATA SEGMENT
362	Read only state

363	Current default table
364	SET PRINT PREVIEW
365	PLATFORM PROPERTIES
366	MAP FILE TYPES
367	SET PLATFORM INTERFACE
368	SUBSELECTION TO ARRAY
369	Database event
370	SET PROCESS VARIABLE
371	GET PROCESS VARIABLE
372	Process number
373	Execute on server
375	New list
376	APPEND TO LIST
377	CLEAR LIST
378	GET LIST ITEM
379	Selected list item
380	Count list items
381	SELECT LIST ITEM
382	REDRAW LIST
383	Load list
384	SAVE LIST
385	SET LIST ITEM
386	SET LIST ITEM PROPERTIES
387	SET LIST PROPERTIES
388	Form event
389	FILTER KEYSTROKE
390	Keystroke
391	SORT LIST
392	SET DEFAULT CENTURY
393	Add to date
394	RESOLVE POINTER
395	SET QUERY LIMIT
396	SET QUERY DESTINATION
397	In transaction
398	Trigger level
399	TRIGGER PROPERTIES
400	Test clipboard
401	GET CLIPBOARD
402	CLEAR CLIPBOARD
403	APPEND TO CLIPBOARD
404	Count menus
405	Count menu items
411	APPEND MENU ITEM
412	INSERT MENU ITEM
413	DELETE MENU ITEM
422	Get menu item
423	SET MENU ITEM KEY
424	Get menu item key

425	SET MENU ITEM STYLE
426	Get menu item style
428	Get menu item mark
430	Get menu title
431	SHOW MENU BAR
432	HIDE MENU BAR
433	SHOW TOOL BAR
434	HIDE TOOL BAR
437	Count screens
438	SCREEN COORDINATES
439	SCREEN DEPTH
440	Menu bar height
441	Menu bar screen
442	WINDOW LIST
443	GET WINDOW RECT
444	SET WINDOW RECT
445	Window kind
446	Window process
447	Frontmost window
448	Next window
449	Find window
450	Get window title
452	DRAG WINDOW
456	REDRAW WINDOW
457	PICTURE PROPERTIES
458	Tickcount
459	Milliseconds
460	FONT LIST
461	Font number
462	Font name
463	Mac to Win
464	Win to Mac
465	POST KEY
466	POST CLICK
467	POST EVENT
468	GET MOUSE
469	SET CURSOR
470	Get platform interface
471	VOLUME LIST
472	VOLUME ATTRIBUTES
473	FOLDER LIST
474	DOCUMENT LIST
475	CREATE DIRECTORY
476	Test path name
477	GET DOCUMENT PROPERTIES
478	SET DOCUMENT PROPERTIES
479	Get document size
480	SET DOCUMENT SIZE

481	Get document position
482	SET DOCUMENT POSITION
483	Current machine
484	Current machine owner
485	ACI folder
486	Temporary folder
487	System folder
488	Gestalt
489	Structure file
490	Data file
491	Application file
492	Compiled application
493	Application version
494	Application type
495	Version type
496	Create resource file
497	Open resource file
498	CLOSE RESOURCE FILE
499	RESOURCE TYPE LIST
500	RESOURCE LIST
501	DELETE RESOURCE
502	GET PICTURE RESOURCE
503	SET PICTURE RESOURCE
504	Get text resource
505	SET TEXT RESOURCE
506	Get string resource
507	SET STRING RESOURCE
508	GET RESOURCE
509	SET RESOURCE
510	Get indexed string
511	STRING LIST TO ARRAY
512	ARRAY TO STRING LIST
513	Get resource name
514	SET RESOURCE NAME
515	Get resource properties
516	SET RESOURCE PROPERTIES
517	GET ICON RESOURCE
519	Mac to ISO
520	ISO to Mac
521	SET PICTURE TO CLIPBOARD
522	GET PICTURE FROM CLIPBOARD
523	SET TEXT TO CLIPBOARD
524	Get text from clipboard
525	DOCUMENT TO BLOB
526	BLOB TO DOCUMENT
527	DATA SEGMENT LIST
528	Document type
529	Document creator

530	SET DOCUMENT TYPE
531	SET DOCUMENT CREATOR
532	VARIABLE TO BLOB
533	BLOB TO VARIABLE
534	COMPRESS BLOB
535	EXPAND BLOB
536	BLOB PROPERTIES
537	SET SCREEN DEPTH
538	Command name
539	Square root
540	MOVE DOCUMENT
541	COPY DOCUMENT
542	Pop up menu
543	Shift down
544	Macintosh control down
545	Macintosh option down
546	Macintosh command down
547	Caps lock down
548	INTEGER TO BLOB
549	BLOB to integer
550	LONGINT TO BLOB
551	BLOB to longint
552	REAL TO BLOB
553	BLOB to real
554	TEXT TO BLOB
555	BLOB to text
556	LIST TO BLOB
557	BLOB to list
558	COPY BLOB
559	INSERT IN BLOB
560	DELETE FROM BLOB
561	REMOVE FROM SET
562	Windows Ctrl down
563	Windows Alt down
564	PICTURE LIBRARY LIST
565	GET PICTURE FROM LIBRARY
566	SET PICTURE TO LIBRARY
567	REMOVE PICTURE FROM LIBRARY
600	COPY SET
601	SET TABLE TITLES
602	SET FIELD TITLES
603	SET VISIBLE
604	C_BLOB
605	BLOB size
606	SET BLOB SIZE
607	DRAG AND DROP PROPERTIES
608	Drop position
609	GET USER LIST

610	GET GROUP LIST
611	GET USER PROPERTIES
612	SET USER PROPERTIES
613	GET GROUP PROPERTIES
614	SET GROUP PROPERTIES
615	DELETE USER
616	Is user deleted
617	START WEB SERVER
618	STOP WEB SERVER
619	SEND HTML FILE
620	SET WEB DISPLAY LIMITS
621	Is a list
622	SET WEB TIMEOUT
623	SET REAL COMPARISON LEVEL
624	DELETE LIST ITEM
625	INSERT LIST ITEM
626	Copy list
627	Current form table
628	SET RGB COLOR
629	List item position
630	SELECT LIST ITEM BY REFERENCE
631	GET LIST ITEM PROPERTIES
632	GET LIST PROPERTIES
633	List item parent
634	SET HTML ROOT
635	VARIABLE TO VARIABLE

TRACE

Language

version 3

TRACE

Parameter	Type	Description
This command does not require any parameters		

Description

You use TRACE to trace methods during development of a database.

The TRACE command turns on the 4th Dimension debugger for the current process. The debugger window is displayed before the next line of code is executed, and continues to be displayed for each line of code that is executed. You can also turn on the debugger by pressing the Alt key (Windows) or the Option key (Macintosh) and the mouse button while code is executing.

In compiled databases, the TRACE command is ignored.

4D Server: If you call TRACE from a project method executed within the context of a Stored Procedure, the debugger window appears on the Server machine.

Tips

1. Do not place TRACE calls when using a form whose On Activate and On Deactivate events have been enabled. Each time the debugger window appears, these events will be invoked; you will then loop infinitely between these events and the debugger window. If you end up in this situation, you can get out of it by switching to the Design environment.

To do so, click in an event window or the debugger window. Then proceed as follows:

• If the call to TRACE is in a project method or an object method (methods reloaded at execution), delete it. This will stop the infinite loop.

• If the call to TRACE is in the form method, it will not be reloaded until you exit the form, which is impossible because you are stuck in a loop. So, either reopen the database or abort the process in question. If the process cannot be aborted, then you have to reopen the database.

2. If you call the TRACE command from within a form or an object method executed during the update of the form at the screen, you will also end up in an infinite repetition of updates and debugger window apparitions. At this point, press Alt + Shift (Windows) or Option-Shift (Macintosh). This will disable the update events for the current window and consequently stop to call TRACE via the form or object methods. Then, you can switch to the Design environment and remove the call to TRACE.

Note that these two tips also apply to the same situations generated by the presence of permanent break points in your code. Regarding Tip 1, no matter where the break point is located, you can remove it and thereby get out of debugger window apparitions without reopening the database.

Example

The following code expects the process variable BUILD_LANG to be equal to “US” or “FR”. If this is not the case, it calls the project method DEBUG:

```
    ` ...
Case of
    : (BUILD_LANG="US")
        vsBHCmdName:=[Commands]CM US Name
    : (BUILD_LANG="FR")
        vsBHCmdName:=[Commands]CM FR Name
Else
    DEBUG ("Unexpected BUILD_LANG value")
End case
```

The DEBUG project method is listed here:

```
    ` DEBUG Project Method
    ` DEBUG (Text)
    ` DEBUG (Optional Debug Information)

C_TEXT ($1)

If (<>vbDebugOn) ` Interprocess variable set in the On Startup Method
    If (Compiled Application)
        If (Count parameters>=1)
            ALERT ($1+Char(13)+"Call Designer at x911")
        End if
    Else
⇒           TRACE
    End if
End if
```

See Also

NO TRACE.

NO TRACE

Language

version 3

NO TRACE

Parameter	Type	Description
This command does not require any parameters		

Description

You use NO TRACE to trace methods during development of a database.

NO TRACE turns off the debugger engaged by TRACE, by an error, or by the user. Using NO TRACE has the same effect as clicking the No Trace button in the debugger.

In compiled databases, the NO TRACE command is ignored.

See Also

TRACE.

23 Math

Abs

Math

version 3

Abs (number) → Number

Parameter	Type		Description
number	Number	→	Number for which to return the absolute value
Function result	Number	←	Absolute value of number

Description

Abs returns the absolute (unsigned, positive) value of number. If number is negative, it is returned as positive. If number is positive, it is returned unchanged.

Example

The following example returns the absolute value of –10.3, which is 10.3:

```
⇒	vlVector:=Abs(–10.3)
```

Int

Int (number) → Number

Parameter	Type		Description
number	Number	→	Number whose integer portion is returned
Function result	Number	←	Integer portion of number

Description

Int returns the integer portion of number. Int truncates a negative number away from zero.

Examples

The following example illustrates how Int works for both positive and negative numbers. Note that the decimal portion of the number is removed:

```
⇒      vlResult:=Int (123.4)  ` vlResult gets 123
⇒      vlResult:=Int(–123.4)  ` vlResult gets –124
```

See Also

Dec.

Dec

version 3

Dec (number) → Number

Parameter	Type		Description
number	Number	→	Number whose decimal portion is returned
Function result	Number	←	Decimal part of number

Description

Dec returns the decimal (fractional) portion of number. The value returned is always positive or zero.

Examples

The following example takes a monetary value expressed as a real number, and extracts the dollar part and the cents part. If vrAmount is 7.31, then vlDollars is set to 7 and vlCents is set to 31:

```
    vlDollars:=Int (vrAmount)  ` Get the dollars
⇒   vlCents:=Dec(vrAmount) * 100  ` Get the fractional part
```

See Also

Int.

Round

Round (round; places) → Number

Parameter	Type		Description
round	Number	→	Number to be rounded
places	Number	→	Number of decimal places used for rounding
Function result	Number	←	Number rounded to the number of decimal places specified by Places

Description

Round returns number rounded to the number of decimal places specified by places.

If places is positive, number is rounded to places decimal places. If places is negative, number is rounded on the left of the decimal point.

If the digit following places is 5 though 9, Round rounds toward positive infinity for a positive number, and toward negative infinity for a negative number. If the digit following places is 0 through 4, Round rounds toward zero.

Examples

The following example illustrates how Round works with different arguments. Each line assigns a number to the vlResult variable. The comments describe the results:

```
⇒	vlResult:=Round (16.857; 2)  ` vlResult gets 16.86
⇒	vlResult:=Round (32345.67; –3)  ` vlResult gets 32000
⇒	vlResult:=Round (29.8725; 3)  ` vlResult gets 29.873
⇒	vlResult:=Round (–1.5; 0)  ` vlResult gets –2
```

See Also

Trunc.

Trunc

Trunc (number; places) → Number

Parameter	Type		Description
number	Number	→	Number to be truncated
places	Number	→	Number of decimal places used for truncating
Function result	Number	←	Number with its decimal part truncated to the number of decimal places specified by Places

Description

Trunc returns number with its decimal part truncated to the number of decimal places specified by places. Trunc always truncates toward negative infinity.

If places is positive, number is truncated to places decimal places. If places is negative, number is truncated on the left of the decimal point.

Examples

The following example illustrates how Trunc works with different arguments. Each line assigns a number to the vlResult variable. The comments describe the results:

```
⇒	vlResult := Trunc (216.897; 1)  ` vlResult gets 216.8
⇒	vlResult := Trunc (216.897; –1)  ` vlResult gets 210
⇒	vlResult := Trunc (–216.897; 1)  ` vlResult gets –216.9
⇒	vlResult := Trunc (–216.897; –1)  ` vlResult gets –220
```

See Also

Round.

Random

Random → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Random number

Description

Random returns a random integer value between 0 and 32,767 (inclusive).

To define a range of integers, use this formula:

```
        (Random%(End–Start+1))+Start
```

The value start is the first number in the range, and the value end is the last.

Example

The following example assigns a random integer between 10 and 30 to the vlResult variable:

```
⇒       vlResult:=(Random%21)+10
```

Mod

Math

version 3

Mod (number1; number2) → Number

Parameter	Type		Description
number1	Number	→	Number to divide
number2	Number	→	Number to divide by
Function result	Number	←	Returns the remainder

Description

The command Mod returns the remainder of the Integer division of number1 by number2.

Note: Mod accepts Integer, Long Integer, and Real expressions. However, if number1 or number2 are real numbers, the numbers are first rounded and then Mod is calculated.

You can also use the % operator to calculate the remainder (see Numeric Operators).

WARNING: The % operator returns valid results with Integer and Long Integer expressions. To calculate the modulo of real values, you must use the Mod command.

Examples

The following example illustrates how the Mod function works with different arguments. Each line assigns a number to the vlResult variable. The comments describe the results:

```
⇒	vlResult:=Mod(3;2)  ` vlResult gets 1
⇒	vlResult:=Mod(4;2)  ` vlResult gets 0
⇒	vlResult:=Mod(3.5;2)  ` vlResult gets 0
```

See Also

Numeric Operators.

Square root

version 6.0

Square root (number) → Number

Parameter	Type		Description
number	Number	→	Number whose square root is calculated
Function result	Number	←	Square root of the number

Description

Square root returns the square root of number.

Examples

1. The line:

```
⇒      $vrSquareRootOfTwo := Square root (2)
```

assigns the value 1.414213562373 to the variable $vrSquareRootOfTwo.

2. The following method returns the hypotenuse of the right triangle whose two legs are passed as parameters:

```
          ` Hypotenuse method
          ` Hypotenuse ( real ; real ) -> real
          ` Hypotenuse ( legA ; legB ) -> Hypotenuse
       C_REAL($0;$1;$2)
⇒      $0 := Square root(($1^2)+($2^2))
```

For instance, *Hypotenuse* (4;3) returns 5.

See Also

Numeric Operators.

Log

version 3

Log (number) → Number

Parameter	Type		Description
number	Number	→	Number for which to return the log
Function result	Number	←	Log of number

Description

Log returns the natural (Napierian) log of number. Log is the inverse function of Exp.

Note: 4D provides the predefined constant e number (2.71828...).

Example

The following line displays 1:

```
⇒	ALERT(String(Log(Exp(1)))
```

See Also

Exp.

Exp

Exp (number) → Number

Parameter	Type		Description
number	Number	→	Number to evaluate
Function result	Number	←	Natural log base by the power of number

Description

Exp raises the natural log base (e = 2.71828...) by the power of number. Exp is the inverse function of Log.

Note: 4D provides the predefined constant e number (2.71828...).

Example

The following example assigns the exponential of 1 to vrE (the log of vrE is 1):

```
⇒    vrE := Exp (1)  ` vrE gets 2.17828...
```

See Also

Log.

Sin

Sin (number) → Number

Parameter	Type		Description
number	Number	→	Number, in radians, whose sine is returned
Function result	Number	←	Sine of number

Description

Sin returns the sine of number, where number is expressed in radians.

Note: 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

See Also

Arctan, Cos, Tan.

Cos

Cos (number) → Number

Parameter	Type		Description
number	Number	→	Number, in radians, whose cosine is returned
Function result	Number	←	Cosine of number

Description

Cos returns the cosine of number, where number is expressed in radians.

Note: 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

See Also

Arctan, Sin, Tan.

Tan

Tan (number) → Number

Parameter	Type		Description
number	Number	→	Number, in radians, whose tangent is returned
Function result	Number	←	Tangent of number

Description

Tan returns the tangent of number, where number is expressed in radians.

Note: 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

See Also

Arctan, Cos, Sin.

Arctan

version 3

Arctan (number) → Number

Parameter	Type		Description
number	Number	→	Tangent for which to calculate the angle
Function result	Number	←	Angle in radians

Description

Arctan returns the angle, expressed in radians, of the tangent number.

Note: 4D provides the predefined constants Pi, Degree, and Radian. Pi returns the Pi number (3.14159...), Degree returns one degree expressed in radians (0.01745...), and Radian returns one radian expressed in degrees (57.29577...).

Examples

The following example shows the value of Pi:

```
⇒	ALERT("Pi is equal to: "+String(Arctan(1)*4))
```

See Also

Cos, Sin, Tan.

SET REAL COMPARISON LEVEL

version 6.0

SET REAL COMPARISON LEVEL (epsilon)

Parameter	Type		Description
epsilon	Number	→	Epsilon value for real equality comparisons

Description

The command SET REAL COMPARISON LEVEL sets the epsilon value used by 4th Dimension to compare real values and expressions for equality.

A computer always performs approximative real computations; therefore, testing real numbers for equality should take this approximation into account. 4th Dimension does this when comparing real numbers, by testing whether or not the difference between the two numbers exceeds a certain value. This value is called the **epsilon** and works this way:

Given two real numbers a and b, if Abs(a-b) is greater than the epsilon, the numbers are considered not equal; otherwise, the numbers are considered equal.

By default, 4th Dimension, sets the epsilon value to 10 power minus 6 (10^-6). Examples:

• 0.00001=0.00002 returns False, because the difference 0.00001 is greater than 10^-6.
• 0.000001=0.000002 returns True, because the difference 0.000001 is not greater than 10^-6.
• 0.000001=0.000003 returns False, because the difference 0.000002 is greater than 10^-6.

Using SET REAL COMPARISON LEVEL, you can increase or decrease the epsilon value as you require.

WARNING: Typically, you will not need to use this command to change the default epsilon value.

IMPORTANT: Changing the epsilon only affects real comparison for equality. It has no effect on other real computations nor on the display of real values.

See Also

Comparison Operators.

Preliminary Note

If you do not deal with cross-platform development, you can skip this section.

On computers, floating point arithmetic is more a technology than a mathematical science. For example, you learned in school that one-third (1/3) can be written as an infinite number of threes after the decimal point. A computer, on the other hand, does not know this and must calculate the expression. In the same way, you know conceptually that three times one third is equal to one; a computer calculates the expression to get the result. Depending on the type of computer you use, one-third is calculated as a limited number of threes after the decimal point. This number is called the “precision” of the machine.

On 68K-based Macintosh, the precision number is 19; this means that 1/3 is calculated with 19 significant digits. On Windows and Power Macintosh, this number is 15; so 1/3 is displayed with 15 significant digits. If you display the expression 1/3 in the Debugger window of 4th Dimension, you will get 0.3333333333333333333 on 68K-based Macintosh and something like 0.3333333333333333148 on Windows or Power Macintosh. Note that the last three digits are different because the precision on Windows and Power Macintosh is less than on the 68K-based Macintosh. Yet, if you display the expression (1/3)*3, the result is 1 on both machines.

If your floating point arithmetic computations deal with the number of square feet in your backyard, you will say “Fine with me!” because you do not care about the digits after the decimal point. On the other hand, if you are filling out an IRS form, you may, in certain circumstances, care about the accuracy of your computer. However, remember that 19 or 15 digits after the decimal point are quite sufficient even if you manage billions of dollars of revenue.

Why does the value 1/3 seem different on 68K Macintosh and onWindows or Power Macintosh?

On 68K-based Macintosh, the operating system stores real numbers on 10 bytes (80 bits), while on Windows and Power Macintosh, it stores them on 8 bytes (64 bits). This is why real numbers have up to 19 significant digits on 68K-based Macintosh and up to 15 significant digits on Windows and Power Macintosh.

So, why does the expression (1/3)*3 return 1 on both machines?

A computer can only make approximate computations. Therefore, while comparing or computing numbers, a computer does not treat real numbers as mathematical objects but as approximate values. In our example, 0.3333... multiplied by 3 gives 0.9999...; the difference between 0.9999... and 1 is so small that the machine considers the result equal to 1, and consequently returns 1. For details on this subject, see the discussion for the command SET REAL COMPARISON LEVEL.

There is dual behavior of real numbers, so we must make the distinction between:
• How they are calculated and compared
• How they are displayed on the screen or printer

Originally, 4th Dimension handled real numbers using the standard 10-byte data type provided by the operating system of the 68K-based Macintosh. Consequently, real values stored in the data file on disk are saved using this format. In order to maintain compatibility between the 68K, Power Macintosh, and Windows versions of 4th Dimension, the 4th Dimension data files still hold the real values using the 10-byte data type. Because floating point arithmetic is performed on Windows or Power Macintosh using the 8 byte format, 4th Dimension converts the values from 10 bytes to 8 bytes, and vice versa. Therefore, if you load a record containing real values, which have been saved on 68K-based Macintosh, onto Windows or Power Macintosh, it is possible to lose some precision (from 19 to 15 significant digits). Yet, if you load a record containing real values, which have been saved on Windows or Power Macintosh, on a 68K-based Macintosh, there will be no loss of precision. Basically, if you use a database on 68K or Power Macintosh and Windows, count on floating point arithmetic with 15 significant digits, not 19.

Using Customizer Plus, you can set the number of digits to be skipped when simplifying the display of real numbers on 68K or Power Macintosh and Windows. The default settings are: no digits on 68K and five digits on Power Macintosh and Windows.

24 Menus

Managing Menus

version 6.0 (Modified)

Terminology

The documentation of Menus commands uses the terms **menu command** and **menu item** interchangeably when describing a line in a menu.

Menu Bars

Menu bars are identified by number, rather than by name. The first menu bar is Menu Bar #1. It is also the default menu bar. To open an application with a menu bar other than Menu Bar #1, you must use the MENU BAR command in the On Startup database method.

Each menu command can have one project method attached to it. If you do not assign a method to a menu command, choosing that menu command causes 4th Dimension to exit the Custom Menus environment and go to the User environment. If the user is working with the 4th Dimension Custom Menus version or does not have access to the User environment, this means quitting to the Desktop.

Every menu bar comes equipped with three menus—the File, Edit and Help menus (Windows) or the Apple, File and Edit menus (Macintosh).

• The File menu has only one menu command—Quit. Quit has no method associated with it; that is how it signals 4th Dimension to exit the Custom Menus environment. You can rename the File menu, add menu commands to it or keep it as is. If it is renamed, it will no longer appear to the left of the Edit menu. It is recommended that you always keep Quit as the last menu command in the File menu.

• The Edit menu contains the standard editing menu commands. The Edit menu is not displayed in the Menu editor and cannot be modified.

• On Windows, the Help menu contains About 4th Dimension and any Windows Help files that may be available for the application. The Help menu is not displayed in the Menu editor and cannot be modified, except for the About menu command, which can be modified using the SET ABOUT command.

• On Macintosh, the Apple menu contains About 4th Dimension and any applications located in the Apple Menu Items system folder. The Apple menu is not displayed in the Menu editor and cannot be modified, except for the About menu command, which can be modified using the SET ABOUT command.

Menu Numbers and Menu Command Numbers

Like menu bars, menus are numbered. The Edit and Help or Apple menus are not included in the count, because they cannot be modified. Instead, File is menu 1. Thereafter, menus are numbered sequentially from left to right (2, 3, 4, and so on). Menu numbering is important when you are working, for example, with the Menu selected function. When a menu is associated with a form, the menu numbering scheme is different. The first appended menu begins with the number 2049. To refer to an appended menu, add 2048 to the normal menu number.

The menu commands within each menu are numbered sequentially from the top of the menu to the bottom. The topmost menu command is item 1.

Associated Menu Bars

You can associate a menu bar with a form by using the Menu Bar... menu command from the Form menu in the Form editor. Such a menu bar is called a “form menu bar” in this document.

The menus on a form menu bar are appended to the current menu bar when the form is displayed. The menus are appended for input forms in both the User and Custom Menus environments and for output forms in the Custom Menus environment.

Form menu bars are specified by a menu bar number. If the number of the menu bar displayed with the current form is the same as the number of the menu bar appended to the form, the menu bar is not appended.

If you specify a negative number for a form menu bar, 4th Dimension uses the absolute value of the menu bar. For example, if you specify –3 as the menu bar, Menu Bar #3 is used. When a form menu bar is specified with a negative number, the menu commands for all the menus in the menu bar (splash screen and form) will execute the methods that are attached to them.

If you do not specify a negative number for a form menu bar, choosing a menu command from that appended menu bar will not execute its method. Instead, an On Menu Selected event will be sent to the form method, and you can use Menu selected to test for the selected menu.

Modifying Menu Items Programmatically

4th Dimension provides the following commands for adding, deleting, inserting or modifying menu items in a menu of the menu bar currently displayed or installed in a process:

• ENABLE MENU ITEM
• DISABLE MENU ITEM
• SET MENU ITEM
• SET MENU ITEM STYLE
• SET MENU ITEM MARK
• SET MENU ITEM
• APPEND MENU ITEM
• INSERT MENU ITEM
• DELETE MENU ITEM

The scope of each of these commands is the current use of the menu bar. As soon as you call MENU BAR again, all the menus and menu items will return to their original state as defined in the Design environment Menu Bar Editor.

Each of these commands expects a menu and a menu item number.

As explained, menus are numbered 1 to N from left to right. For example, File is usually the first menu. On Windows, the Edit and Help menus are excluded from this numbering. On Macintosh, the Apple and Edit menus are excluded.

Note that the command Count menus does not take these menus into account. For example, if you have a menu bar composed of the File, Customers and Invoices menus, Count menus will return 3, ignoring the system menus maintained by 4D.

Menu items are numbered 1 to N from top to bottom, including separator lines.

Menus that are inserted in the menu bar by means of a menu bar associated with a form, and therefore appended to the current menu bar, are numbered from left to right starting with the number 2049 (2048 + 1 to N).

The command Menu selected returns menu and menu item numbers using that convention.

Warning: These commands cannot access the system menus maintained by 4D (Edit and Help on Windows, Apple and Edit on Macintosh).

Connected Menus: Menus can be connected to menu bars. If a connected menu is modified using one of these commands, every other instance of the menu will reflect these changes. For more information about connecting menus, refer to the *4th Dimension Design Reference* Manual.

MENU BAR

MENU BAR (menuBar{; process}{; *})

Parameter	Type		Description
menuBar	Number	→	Number of the menu bar
process	Number	→	Process reference number
*		→	Save menu bar state

Description

MENU BAR replaces the current menu bar with the menu bar specified by menuBar for the current process only.

The optional process parameter changes the menu bar of the specified process to menuBar. The optional * parameter allows you to save the state of the menu bar. If this parameter is omitted, MENU BAR reinitializes the menu bar when the command is executed.

For example, suppose that MENU BAR(1) is executed. Next, several menu commands are disabled using the DISABLE MENU ITEM command.

If MENU BAR(1) is executed a second time, either from the same process or from a different process, all menu commands will revert to their initial enabled state.

If MENU BAR(1;*) is executed, the menu bar will retain the same state as before, and the menu commands that were disabled will remain disabled.

Note: If you do not use the optional process parameter, * can be the second parameter. In other words, MENU BAR(1;2;*) and MENU BAR(1; *) are both valid statements.

When a user enters the Custom Menus environment, the first menu bar is displayed (Menu Bar #1). You can change this menu bar when opening a database by specifying the desired menu bar in the On Startup database method or in the startup method for an individual user.

Examples

1. The following example changes the current menu bar to menu bar #3 and resets the states of the menu commands to their original states:

```
⇒	MENU BAR (3)
```

2. The following example changes the current menu bar to menu bar #3 and saves the states of the menu commands. Menu commands that were previously disabled will appear disabled.

```
⇒	MENU BAR (3;*)
```

3. The following example sets the current menu bar to menu bar #3 while records are being modified. After the records have been modified, the menu bar is reset to menu bar #2, with the menu state saved:

```
⇒	MENU BAR(3)
	ALL RECORDS([Customers])
	MODIFY SELECTION([Customers])
⇒	MENU BAR(2;*)
```

See Also

Managing Menus.

HIDE MENU BAR

version 6.0

HIDE MENU BAR

Parameter	Type	Description
This command does not require any parameters		

Description

The command HIDE MENU BAR makes the menu bar invisible.

If the menu bar was already hidden, the command does nothing.

Example

The following method displays a record in full-screen display (Macintosh) until you click the mouse button:

```
       HIDE TOOL BAR
⇒      HIDE MENU BAR
       Open window(-1;-1;1+Screen width;1+Screen height;Alternate dialog box)
       INPUT FORM([Paintings];"Full Screen 800")
       DISPLAY RECORD([Paintings])
       Repeat
          GET MOUSE($vlX;$vlY;$vlButton)
       Until($vlButton#0)
       CLOSE WINDOW
       SHOW MENU BAR
       SHOW TOOL BAR
```

Note: On Windows, the window will be limited to the bounds of the application window.

See Also

HIDE TOOL BAR, SHOW MENU BAR, SHOW TOOL BAR.

SHOW MENU BAR

SHOW MENU BAR

Parameter	Type	Description

This command does not require any parameters

Description

The command SHOW MENU BAR makes the menu bar visible.

If the menu bar was already visible, the command does nothing.

Example

See example for the command HIDE MENU BAR.

See Also

HIDE MENU BAR, HIDE TOOL BAR, SHOW TOOL BAR.

SET ABOUT

version 3

SET ABOUT (itemText; method)

Parameter	Type		Description
itemText	String	→	New About menu item text
method	String	→	Method to execute when menu item is chosen

Description

The command SET ABOUT changes the About 4th Dimension menu command in the Help (Windows) or Apple (Macintosh) menu to ItemText.

After the call, when a user selects this menu command, method will be called. The method can open a dialog box to give version information about your database.

The 4th Dimension icon, 4th Dimension version number, 4D Compiler version number, and a single-line copyright notice will be appended across the top of the dialog box.

Note: Your application compilation version number is displayed, too, if you have activated the Automatic Version option in your 4D Compiler project.

Examples

1. The following example replaces the About 4th Dimension menu command with the About Scheduler menu command. The ABOUT method displays a custom About box:

```
⇒	SET ABOUT(“About Scheduler…”; “ABOUT”)
```

2. The following example resets the About 4th Dimension menu command back to the original About box:

```
⇒	SET ABOUT("About 4th Dimension®";"")
```

Menu selected

version 3

Menu selected → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Menu command selected Menu number in high word Menu item number in low word

Description

Menu selected is used only when forms are displayed. It detects which menu command has been chosen from a menu.

Tip: Whenever possible, use methods associated with menu commands in an associated menu bar (with a negative menu bar number) instead of using Menu selected. Associated menu bars are easier to manage, since it is not necessary to test for their selection. However, if you use the commands APPEND MENU ITEM or INSERT MENU ITEM, you have to use Menu selected because the menu items added by these commands do not have associated methods.

Menu selected returns the menu-selected number, a long integer. To find the menu number, divide Menu selected by 65,536 and convert the result to an integer. To find the menu command number, calculate the modulo of Menu selected with the modulus 65,536. Use the following formulas to calculate the menu number and menu command number:

⇒ Menu := **Menu selected** \ 65536

⇒ menu command := **Menu selected** % 65536

Starting with version 6, you can also extract these values using the bitwise operators as follows:

⇒ Menu := (**Menu selected** & 0xFFFF0000) >> 16

⇒ menu command := **Menu selected** & 0xFFFF

If no menu commands is selected, Menu selected returns 0.

Example

The following form method uses Menu selected to supply the menu and menu command arguments to SET MENU ITEM MARK:

```
        Case of
            : (Form event=On Menu Selected)
⇒               If (Menu selected # 0)
⇒                   SET MENU ITEM MARK (Menu selected\65536;Menu
selected%65536;Char(18))
                End if
        End case
```

See Also

Managing Menus.

Count menus

version 6.0

Count menus {(process)} → Number

Parameter	Type		Description
process	Number	→	Process reference number
Function result	Number	←	Number of menus in the current menu bar

Description

The command Count menus returns the number of menus present in the menu bar.

If you omit the process parameter, Count menus applies to the menu bar for the current process. Otherwise, Count menus applies to the menu bar for the process whose reference number is passed in process.

See Also

Count menu items.

Count menu items

version 6.0

Count menu items (menu{; process}) → Number

Parameter	Type		Description
menu	Number	→	Menu number
process	Number	→	Process reference number
Function result	Number	←	Number of menu items in the menu

Description

The command Count menu items returns the number of menu items present in the menu whose number is passed in menu.

If you omit the process parameter, Count menu items applies to the menu bar for the current process. Otherwise, Count menu items applies to the menu bar for the process whose reference number is passed in process.

See Also

Count menus.

Get menu title

Get menu title (menu{; process}) → String

Parameter	Type		Description
menu	Number	→	Menu number
process	Number	→	Process reference number
Function result	String	←	Title of the menu

Description

The command Get menu title returns the title of the menu whose number is passed in menu.

If you omit the process parameter, Get menu title applies to the menu bar for the current process. Otherwise, Get menu title applies to the menu bar for the process whose reference number is passed in process.

See Also

Count menus.

Get menu item

version 6.0

Get menu item (menu; menuItem{; process}) → String

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Process reference number
Function result	String	←	Text of the menu item

Description

The command Get menu item returns the text of the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, Get menu item applies to the menu bar for the current process. Otherwise, Get menu item applies to the menu bar for the process whose reference number is passed in process.

See Also

SET MENU ITEM.

SET MENU ITEM

version 3

SET MENU ITEM (menu; menuItem; itemText{; process})

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
itemText	String	→	New text for the menu item
process	Number	→	Process reference number

Description

The command SET MENU ITEM changes the text of the menu item whose menu and item numbers are passed in menu and menuItem, to the text passed in itemText.

If you omit the process parameter, SET MENU ITEM applies to the menu bar for the current process. Otherwise, SET MENU ITEM applies to the menu bar for the process whose reference number is passed in process.

See Also

Get menu item.

Get menu item style

Get menu item style (menu; menuItem{; process}) → Number

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Process reference number
Function result	Number	←	Current menu item style

Description

The command Get menu item style returns the font style of the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, Get menu item style applies to the menu bar for the current process. Otherwise, Get menu item style applies to the menu bar for the process whose reference number is passed in process.

Get menu item style returns a combination (one or a sum) of the following predefined constants:

Constant	Type	Value
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	4
Outline	Long Integer	8
Shadow	Long Integer	16
Condensed	Long Integer	32
Extended	Long Integer	64

Note: On Windows, only the styles Plain or a combination of Bold, Italic, and Underline are available.

Example

To test if a menu item is displayed in bold, you write:

```
⇒	If ((Get menu item style($vlMenu;$vlItem) & Bold)#0)
	   `...
	End if
```

See Also

SET MENU ITEM STYLE.

SET MENU ITEM STYLE

version 6.0

SET MENU ITEM STYLE (menu; menuItem; itemStyle{; process})

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
itemStyle	Number	→	New menu item style
process	Number	→	Process reference number

Description

The command SET MENU ITEM STYLE changes the font style of the menu item whose menu and item numbers are passed in menu and menuItem according to the font style passed in itemStyle.

If you omit the process parameter, SET MENU ITEM STYLE applies to the menu bar for the current process. Otherwise, SET MENU ITEM STYLE applies to the menu bar for the process whose reference number is passed in process.

You specify the font style of the item in the itemStyle parameter. You pass a combination (one or a sum) of the following predefined constants:

Constant	Type	Value
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	4
Outline	Long Integer	8
Shadow	Long Integer	16
Condensed	Long Integer	32
Extended	Long Integer	64

Note: On Windows, only the styles Plain or a combination of Bold, Italic, and Underline are available.

See Also

Get menu item style.

Get menu item mark

version 6.0

Get menu item mark (menu; menuItem{; process}) → String

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Process reference number
Function result	String	←	Current menu item mark

Description

The command Get menu item mark returns the check mark of the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, Get menu item mark applies to the menu bar for the current process. Otherwise, Get menu item mark applies to the menu bar for the process whose reference number is passed in process.

If the menu item has no mark, Get menu item mark returns an empty string.

Note: See discussion of check marks on Macintosh and Windows in the description of the command SET MENU ITEM MARK.

Example

The following example toggles the check mark of a menu item:

```
⇒      SET ITEM MARK($vlMenu;$vlItem;
                         Char(18)*Num(Get menu item mark($vlMenu;$vlItem)=""))
```

See Also

SET MENU ITEM MARK.

SET MENU ITEM MARK (menu; item; mark{; process})

Parameter	Type		Description
menu	Number	→	Menu number
item	Number	→	Item number
mark	String	→	New menu item mark
process	Number	→	Process reference number

Description

The command SET MENU ITEM MARK changes the check mark of the menu item whose menu and item numbers are passed in menu and menuItem to the first character of the string passed in mark.

If you omit the process parameter, SET MENU ITEM MARK applies to the menu bar for the current process. Otherwise, SET MENU ITEM MARK applies to the menu bar for the process whose reference number is passed in process.

If you pass an empty string, any mark is removed from the menu item. Otherwise:
• On Macintosh, the first character of the string becomes the mark of the menu item. Usually, you will pass Char (18), which is the check mark character for Macintosh menus.
• On Windows, the menu item is assigned the standard check mark.

Example

See example for the command Get item mark.

See Also

Get menu item mark.

Get menu item key

version 6.0

Get menu item key (menu; menuItem{; process}) → Number

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Process reference number
Function result	Number	←	ASCII code of menu item key

Description

The command Get menu item key returns the ASCII code of the Ctrl (Windows) or Command (Macintosh) shortcut for the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, Get menu item key applies to the menu bar for the current process. Otherwise, Get menu item key applies to the menu bar for the process whose reference number is passed in process.

If the menu item has no shortcut, Get menu item key returns 0 (zero).

See Also

SET MENU ITEM KEY.

SET MENU ITEM KEY

SET MENU ITEM KEY (menu; menuItem; itemKey{; process})

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
itemKey	Number	→	ASCII code of new menu item key
process	Number	→	Process reference number

Description

The command SET MENU ITEM KEY changes the Ctrl (Windows) or Command (Macintosh) shortcut for the the menu item whose menu and item numbers are passed in menu and menuItem, to the character whose ASCII code is passed in itemKey.

If you omit the process parameter, SET MENU ITEM KEY applies to the menu bar for the current process. Otherwise, SET MENU ITEM KEY applies to the menu bar for the process whose reference number is passed in process.

If you pass 0 (zero) in itemKey, any shortcut is removed from the menu item.

Note: For consistency in the user interface, use uppercase characters, digits or symbols that are available on the keyboard, without using any modifier key other than the Ctrl (Windows) or Command (Macintosh) key.

See Also

Get menu item key.

DISABLE MENU ITEM

DISABLE MENU ITEM (menu; menuItem{; process})

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Proces reference number

Description

The command DISABLE MENU ITEM disables the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, DISABLE MENU ITEM applies to the menu bar for the current process. Otherwise, DISABLE MENU ITEM applies to the menu bar for the process whose reference number is passed in process.

Tip: To enable/disable all items of a menu at once, pass 0 (zero) in menuItem.

See Also

ENABLE MENU ITEM.

ENABLE MENU ITEM

ENABLE MENU ITEM (menu; menuItem{; process})

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Proces reference number

Description

The command ENABLE MENU ITEM enables the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, ENABLE MENU ITEM applies to the menu bar for the current process. Otherwise, ENABLE MENU ITEM applies to the menu bar for the process whose reference number is passed in process.

Tip: To enable/disable all items of a menu at once, pass 0 (zero) in menuItem.

See Also

DISABLE MENU ITEM.

APPEND MENU ITEM

version 6.0

APPEND MENU ITEM (menu; itemText{; process})

Parameter	Type		Description
menu	Number	→	Menu number
itemText	String	→	Text for the new menu items
process	Number	→	Process reference number

Description

The command APPEND MENU ITEM appends new menu items to the menu whose number is passed in menu.

If you omit the process parameter, APPEND MENU ITEM applies to the menu bar for the current process. Otherwise, APPEND MENU ITEM applies to the menu bar for the process whose reference number is passed in process.

APPEND MENU ITEM allows you to append one or several menu items in one call.

You define the items to be appended with the parameter itemText as follows:

• Separate each item from the next one with a semi-colon (;). For example, "ItemText1;ItemText2;ItemText3".
• To disable an item: Place an opening parenthesis (() in the item text.
• To specify a separation line: Pass "(-" as item text.
• To specify a font style for a line: In the item text, place a less than sign (<) followed by one of these characters:

<B	Bold
<I	Italic
<U	Underline
<O	Outline (Macintosh only)
<S	Shadow (Macintosh only)

• To add a check mark to an item: In the item text, place a question mark (?) followed by the character you want as a check mark. On Macintosh, the character is displayed; on Windows, a check mark is displayed no matter what character you passed.
• To add an icon to an item: In the item text, place a circumflex accent (^) followed by a character whose ASCII code minus 48 is the resource ID of a MacOS-based icon resource.
• To add a shortcut to an item: In the item text, place a slash (/) followed by the shortcut character for the item.

Note: Use menus that have a reasonable number of items. If you want to display more than 50 items, think about a using scrollable area in a form instead of a menu.

Note: APPEND MENU ITEM accepts up to 32,000 characters, while INSERT MENU ITEM accepts up to 255 characters.

Important: The new items do not have any associated method. Therefore, they must be managed from within a form method using the Menu selected command.

Example

The following example appends the names of the available fonts to the Font menu, which in this example is the sixth menu of the current menu bar:

```
       ` In the On Startup database method
       ` The font list is loaded and menu item text is built
   FONT LIST(<>asAvailableFont)
   <>atFontMenuItems:=""
   For ($vlFont;1;Size of array(<>asAvailableFont))
       <>atFontMenuItems:=<>atFontMenuItems+";"+<>asAvailableFont{$vlFont}
   End for
```

Then, in any form or project method, you can write:

```
⇒  APPEND MENU ITEM(6;<>atFontMenuItems)
```

See Also

DELETE MENU ITEM, INSERT MENU ITEM.

INSERT MENU ITEM

version 6.0

INSERT MENU ITEM (menu; beforeItem; itemText{; process})

Parameter	Type		Description
menu	Number	→	Menu number
beforeItem	Number	→	Menu item number
itemText	Number	→	Text for the menu item to be inserted
process	Number	→	Process reference number

Description

The command INSERT MENU ITEM inserts new menu items into the menu whose number is passed in menu before the existing menu item whose number is passed in beforeItem.

If you omit the process parameter, INSERT MENU ITEM applies to the menu bar for the current process. Otherwise, INSERT MENU ITEM applies to the menu bar for the process whose reference number is passed in process.

INSERT MENU ITEM allows to you insert one or several menu items in one call.

INSERT MENU ITEM works like APPEND MENU ITEM, except for the following differences:
• INSERT MENU ITEM enables you to insert items anywhere in the menu, while APPEND MENU ITEM always adds them at the end of the menu.
• With INSERT MENU ITEM, the item definition passed in itemText is limited to 255 characters, while with APPEND MENU ITEM, itemText is limited to 32,000 characters.

See the description of the command APPEND MENU ITEM for details about the the item definition passed in itemText.

Important: The new items do not have any associated method. Therefore, they must be managed from within a form method using the Menu selected command.

See Also

APPEND MENU ITEM.

DELETE MENU ITEM

version 6.0

DELETE MENU ITEM (menu; menuItem{; process})

Parameter	Type		Description
menu	Number	→	Menu number
menuItem	Number	→	Menu item number
process	Number	→	Process reference number

Description

The command DELETE MENU ITEM deletes the menu item whose menu and item numbers are passed in menu and menuItem.

If you omit the process parameter, DELETE MENU ITEM applies to the menu bar for the current process. Otherwise, DELETE MENU ITEM applies to the menu bar for the process whose reference number is passed in process.

Note: For consistency in the user interface, do not keep a menu with no items.

See Also

APPEND MENU ITEM, INSERT MENU ITEM.

25 Messages

MESSAGES OFF

MESSAGES OFF

Parameter	Type	Description
This command does not require any parameters		

Description

The commands MESSAGES ON and MESSAGES OFF turn on and off the progress meters displayed by 4th Dimension while executing time-consuming operations. By default, messages are on.

The following table shows User environment operations that display the progress meter:

Apply Formula	Quick Report	Order by
Export Data	Import Data	Graph
Query by Form	Query by Formula	Query Editor

The following table lists the commands that display the progress meter:

APPLY TO SELECTION	IMPORT SYLK	QUERY
DISTINCT VALUES	IMPORT TEXT	QUERY BY FORMULA
EXPORT DIF	RELATE MANY SELECTION	QUERY BY FORM
EXPORT SYLK	RELATE ONE SELECTION	QUERY SELECTION
EXPORT TEXT	REDUCE SELECTION	QUERY SELECTION BY FORMULA
GRAPH TABLE	REPORT	ORDER BY FORMULA
IMPORT DIF	SCAN INDEX	ORDER BY

Example

The following example turns off the progress meter before doing a sort, and then turns it back on after completing the sort:

```
⇒	MESSAGES OFF
	ORDER BY ([Addresses];[Addresses]ZIP;>;[Addresses]Name2;>)
	MESSAGES ON
```

MESSAGES ON

MESSAGES ON

Parameter	Type	Description
This command does not require any parameters		

Description

See the description of the MESSAGES OFF command.

ALERT

version 6.0 (Modified)

ALERT (message{; ok button title})

Parameter	Type		Description
message	String	→	Message to display in the alert dialog box
ok button title	String	→	OK button title

Description

The ALERT command displays an alert dialog box composed of a note icon, a message, and an OK button.

You pass the message to be displayed in the parameter message. This message can be up to 255 characters long. However, if the message does not fit into the message area, it can appear truncated, depending on its length and the width of the characters.

By default, the title of the OK button is “OK.” To change the title of the OK button, pass the new custom title into the optional parameter ok button title. If necessary, the OK button width is resized toward the left, according to the width of the custom title you pass.

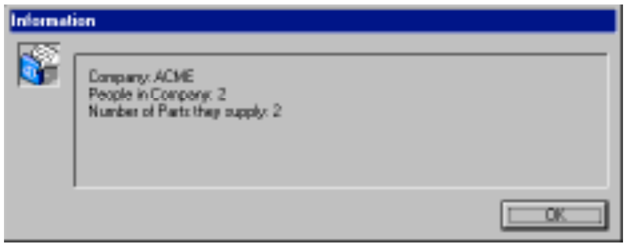
Tip: Do not call the ALERT command from the section of a form or object method that handles the On Activate or On Deactivate form events; this will cause an endless loop.

Examples

1. The following example displays an alert showing information about a company. Note that the displayed string contains carriage returns, which cause the string to wrap to the next line:

```
⇒    ALERT("Company: "+[Companies]Name+Char(13)+"People in company: "+
          String(Records in selection([People]))+Char(13)+"Number of parts they supply: "+
                                        String (Records in selection([Parts])))
```

This line of code displays the following alert box (on Windows):



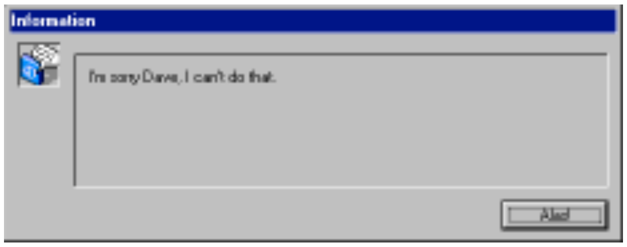
Information

Company: ACME
People in Company: 2
Number of Parts they supply: 2

2. The line:

⇒ **ALERT**("I'm sorry Dave, I can't do that.";"Alas!")

will display the alert dialog box (on Windows) shown here:

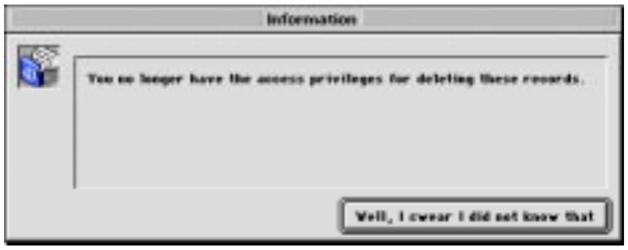


I'm sorry Dave, I can't do that.

3. The line:

⇒ **ALERT**("You no longer have the access privileges for deleting these records.";"Well, I swear I did not know that")

will display the alert dialog box (On Macintosh) shown here:



You no longer have the access privileges for deleting these records.

See Also

CONFIRM, Request.

CONFIRM

version 6.0 (Modified)

CONFIRM (message{; OK button title{; cancel button title}})

Parameter	Type		Description
message	String	→	Message to display in the confirmation dialog box
OK button title	String	→	OK button title
cancel button title	String	→	Cancel button title

Description

The CONFIRM command displays a confirm dialog box composed of a note icon, a message, an OK button, and a Cancel Button.

You pass the message to be displayed in the message parameter. This message can be up to 255 characters long. If the message does not fit into the message area, it can appear truncated, depending on its length and the width of the characters.

By default, the title of the OK button is “OK” and that of the Cancel button is “Cancel.” To change the titles of these buttons, pass the new custom titles into the optional parameters ok button title and cancel button title. If necessary, the width of the buttons is resized toward the left, according to the width of the custom titles you pass.

The OK button is the default button. If the user clicks the OK button or presses Enter to accept the dialog box, the OK system variable is set to 1. If the user clicks the Cancel button to cancel the dialog box, the OK system variable is set to 0.

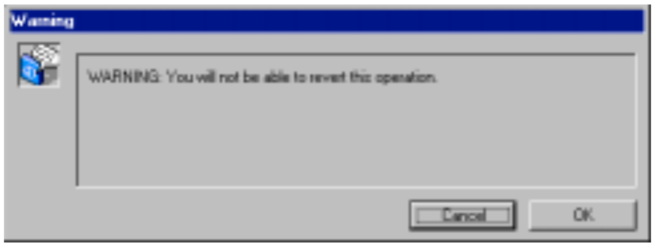
Tip: Do not call the CONFIRM command from the section of a form or object method that handles the On Activate or On Deactivate form events; this will cause an endless loop.

Examples

1. The line:

```
⇒   CONFIRM("WARNING: You will not be able to revert this operation.")
    If (OK=1)
       ALL RECORDS([Old Stuff])
       DELETE SELECTION([Old Stuff])
    Else
       ALERT ("Operation canceled.")
    End if
```

will display the confirm dialog box (on Windows) shown here:

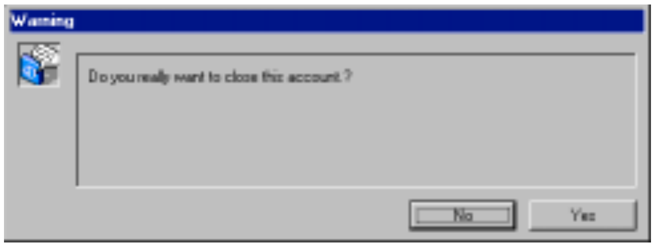

Warning

WARNING: You will not be able to revert this operation.

2. The line:

⇒ **CONFIRM**("Do you really want to close this account?";"Yes";"No")

will display the confirm dialog box (on Windows) shown here:

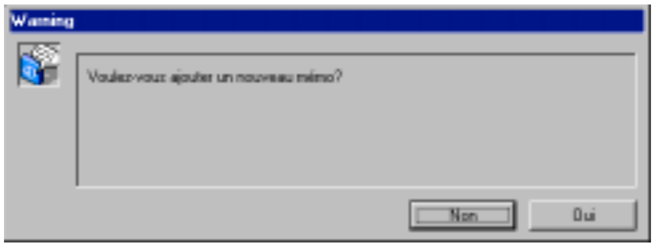

Warning

Do you really want to close this account ?

3. You are writing a 4D application for the international market. You wrote a project method that returns the correct localized text from its English version. You have also populated an array named <>asLocalizedUIMessages,where you store the most common words. In doing so, the line:

⇒ **CONFIRM**(*INTL Text* ("Do you want to add a new Memo?");
<>asLocalizedUIMessages{kLoc_YES};<>asLocalizedUIMessages{kLoc_NO})

could display the French confirm dialog box (on Windows) shown here:

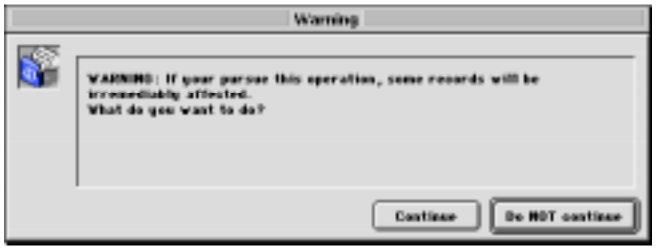

Warning

Voulez-vous ajouter un nouveau mémo?

4. The line:

```
⇒	CONFIRM("WARNING: If your pursue this operation, some records will be "+
				"irremediably affected."+Char(13)+"What do you want to do?";
						"Do NOT continue";"Continue")
```

will display the confirm dialog box (on Macintosh) shown here:



WARNING: If your pursue this operation, some records will be irremediably affected.
What do you want to do?

See Also
ALERT, Request.

Request

Request (message{; default response{; OK button title{; Cancel button title}}}) → String

Parameter	Type		Description
message	String	→	Message to display in the request dialog box
default response	String	→	Default data for the enterable text area
OK button title	String	→	OK button title
Cancel button title	String	→	Cancel button title
Function result	String	←	Value entered by user

Description

The command Request displays a request dialog box composed of a message, a text input area, an OK button, and a Cancel Button.

You pass the message to be displayed in the message parameter. This message can be up to 255 characters long. If the message does not fit into the message area, it can appear truncated, depending on its length and the width of the characters.

By default, the title of the OK button is “OK” and that of the Cancel button is “Cancel.” To change the titles of these buttons, pass the new custom titles into the optional parameters ok button title and cancel button title. If necessary, the width of the buttons is resized toward the left, according to the width of the custom titles you pass.

The OK button is the default button. If you click the OK button or press Enter to accept the dialog box, the OK system variable is set to 1. If you click the Cancel button to cancel the dialog box, the OK system variable is set to 0.

The user can enter text into the text input area. To specify a default value, pass the default text in the default response parameter. If the user clicks OK, Request returns the text. If the user clicks Cancel, Request returns an empty string (""). If the response should be a numeric or a date value, convert the string returned by Request to the proper type with the Num or Date functions.

Tip: Do not call the Request command from the section of a form or object method that handles the On Activate or On Deactivate form event; this will cause an endless loop.

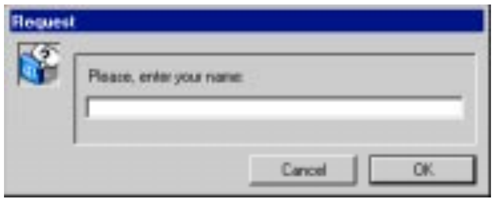
Tip: If you need to get several pieces of information from the user, design a form and present it with DIALOG, rather than presenting a succession of Request dialog boxes.

Examples

1. The line:

```
⇒      $vsPrompt := Request ("Please enter your name:")
```

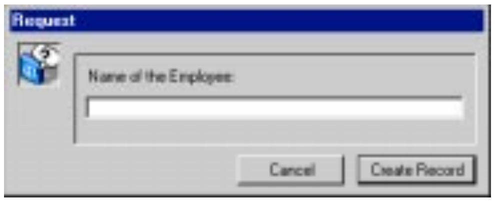
will display the request dialog box (on Windows) shown here:



2. The line:

```
⇒      vsPrompt := Request ("Name of the Employee:";"";"Create Record";"Cancel")
       If (OK=1)
           ADD RECORD ([Employees])
               ` Note: vsPrompt is then copied into the field [Employees]Last name
               ` during the On Load event in the form method
       End if
```

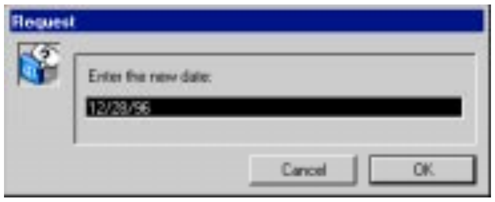
will display the request dialog box (on Windows) shown here:



3. The line:

```
⇒      $vdPrompt := Date (Request ("Enter the new date:";String (Current date)))
```

will display the request dialog box (on Windows) shown here:



See Also

ALERT, CONFIRM.

MESSAGE

MESSAGE (message)

Parameter	Type		Description
message	String	→	Message to display

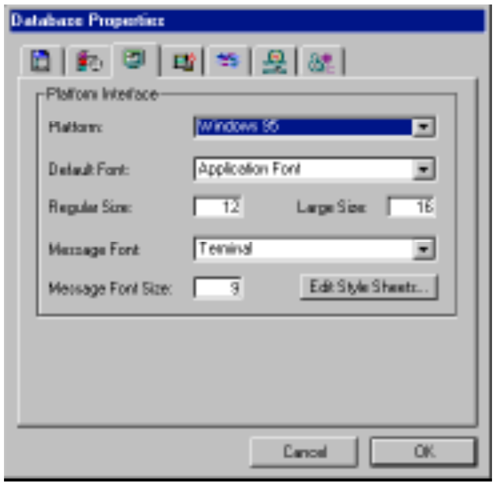
Description

The MESSAGE command is usually used to inform the user of some activity. It displays message on the screen in a special message window that opens and closes each time you call MESSAGE, unless you work with a window you previously opened using Open window (see the following details). The message is temporary and is erased as soon as a form is displayed or the method stops executing. If another MESSAGE is executed, the old message is erased.

If a window is opened with Open window, all subsequent calls to MESSAGE display the messages in that window. The window behaves like a terminal:

• Successive messages do not erase previous messages when displayed in the window. Instead, they are concatenated onto existing messages.
• If a message is wider than the window, 4th Dimension automatically performs text wrap.
• If a message has more lines than the window, 4th Dimension automatically scrolls the message window.
• To control line breaks, include carriage returns — Char(13) — into your message.
• To display the text at a particular place in the window, call GOTO XY.
• To erase the contents of the window, call ERASE WINDOW .
• The window is only an output window and does not redraw when other windows overlap it.

4th Dimension uses the Message Font and Message Font Size properties to display messages. You can change these settings in the Database Properties dialog box:


Database Properties

Platform Interface

Platform: Windows 95

Default Font: Application Font

Regular Size: 12 Large Size: 16

Message Font: Terminal

Message Font Size: 9

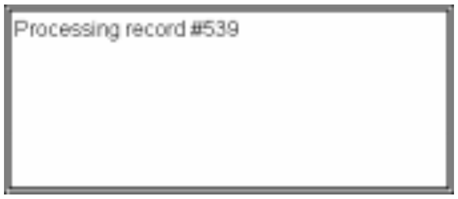
You can choose the font and the font size (within limits) at your convenience. However, if you combine the use of MESSAGE and GOTO XY, it is a good idea to choose a fixed width font, such as Terminal on Windows or Monaco on Macintosh.

Examples

1. The following example processes a selection of records and calls MESSAGE to inform the user about the progress of the operation:

```
    For($vlRecord;1;Records in selection([anyTable]))
⇒       MESSAGE ("Processing record #"+String($vlRecord))
           ` Do Something with the record
        NEXT RECORD([anyTable])
    End for
```

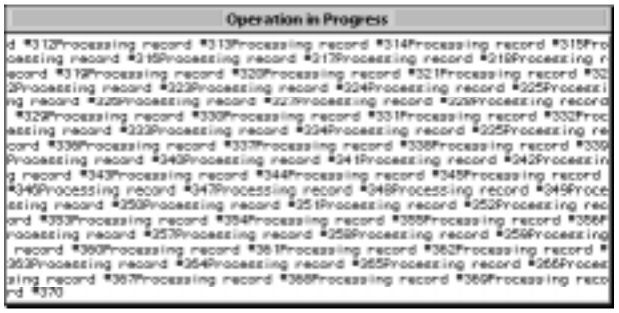
The following window appears and disappears at each MESSAGE call:


Processing record #539

2. In order to avoid this "blinking" window, you can display the messages in a window opened using Open window, as in this example:

```
        Open window(50;50;500;250;5;"Operation in Progress")
        For($vlRecord;1;Records in selection([anyTable]))
⇒           MESSAGE ("Processing record #"+String($vlRecord))
                ` Do Something with the record
            NEXT RECORD([anyTable])
        End for
        CLOSE WINDOW
```

This provides the following result (shown here on Macintosh):

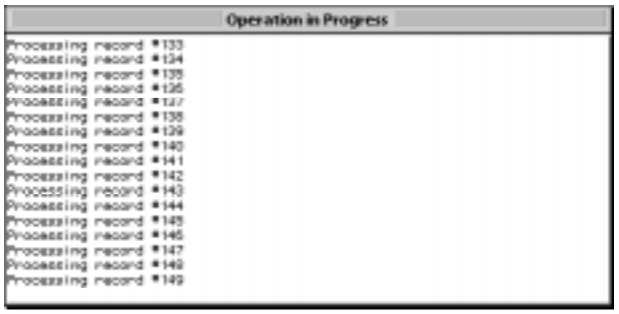


```
d #312Processing record #313Processing record #314Processing record #315Pro
cessing record #316Processing record #317Processing record #318Processing r
ecord #319Processing record #320Processing record #321Processing record #32
2Processing record #323Processing record #324Processing record #325Processi
ng record #326Processing record #327Processing record #328Processing record
 #329Processing record #330Processing record #331Processing record #332Proc
essing record #333Processing record #334Processing record #335Processing re
cord #336Processing record #337Processing record #338Processing record #339
Processing record #340Processing record #341Processing record #342Processin
g record #343Processing record #344Processing record #345Processing record
#346Processing record #347Processing record #348Processing record #349Proce
ssing record #350Processing record #351Processing record #352Processing rec
ord #353Processing record #354Processing record #355Processing record #356P
rocessing record #357Processing record #358Processing record #359Processing
 record #360Processing record #361Processing record #362Processing record #
363Processing record #364Processing record #365Processing record #366Proces
sing record #367Processing record #368Processing record #369Processing reco
rd #370
```

3. Adding a carriage return makes a better presentation:

```
        Open window(50;50;500;250;5;"Operation in Progress")
        For($vlRecord;1;Records in selection([anyTable]))
⇒           MESSAGE ("Processing record #"+String($vlRecord)+Char(13))
                ` Do Something with the record
            NEXT RECORD([anyTable])
        End for
        CLOSE WINDOW
```

This provides the following result (shown here on Macintosh):

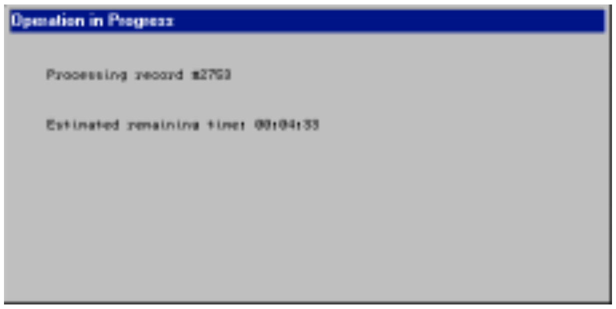


```
Processing record #133
Processing record #134
Processing record #135
Processing record #136
Processing record #137
Processing record #138
Processing record #139
Processing record #140
Processing record #141
Processing record #142
Processing record #143
Processing record #144
Processing record #145
Processing record #146
Processing record #147
Processing record #148
Processing record #149
```

4. Using GOTO XY and writing some additional lines:

```
        Open window(50;50;500;250;5;"Operation in Progress")
        $vlNbRecords:=Records in selection([anyTable])
        $vhStartTime:=Current time
        For($vlRecord;1;$vlNbRecords)
            GOTO XY(5;2)
⇒           MESSAGE ("Processing record #"+String($vlRecord)+Char(13))
                ` Do Something with the record
            NEXT RECORD([anyTable])
            GOTO XY(5;5)
            $vlRemaining:=(($vlNbRecords/$vlRecord)-1)*(Current time-$vhStartTime)
⇒           MESSAGE ("Estimated remaining time: "+Time string($vlRemaining))
        End for
        CLOSE WINDOW
```

This provides the following result (shown here on Windows):



```
Processing record #2763

Estimated remaining time: 00:04:33
```

See Also
CLOSE WINDOW, ERASE WINDOW, GOTO XY, Open window.

GOTO XY

GOTO XY (x; y)

Parameter	Type		Description
x	Number	→	x (horizontal) position of cursor
y	Number	→	y (vertical) position of cursor

Description

The GOTO XY command is used in conjunction with the MESSAGE command when you display messages in a window opened using Open window.

GOTO XY positions the character cursor (an invisible cursor) to set the location of the next message in the window.

The upper-left corner is position 0,0. The cursor is automatically placed at 0,0 when a window is opened and after ERASE WINDOW is executed.

After GOTO XY positions the cursor, you can use MESSAGE to display characters in the window.

Tip: Using a fixed-width (monospaced) font, such as Terminal on Windows and Monaco on Macintosh, for the message, gives the best display results with GOTO XY and MESSAGE. See the description of the MESSAGE command for more information.

Examples

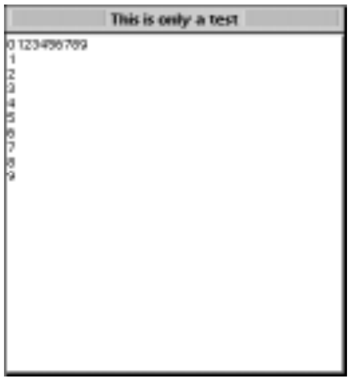
1. See example for the command MESSAGE.

2. See example for the command Milliseconds.

3. The following example:

```
Open window(50;50;300;300;5;"This is only a test")
For ($vlRow;0;9)
   GOTO XY($vlRow;0)
   MESSAGE(String($vlRow))
End for
For ($vlLine;0;9)
   GOTO XY(0;$vlLine)
   MESSAGE(String($vlLine))
End for
$vhStartTime:=Current time
Repeat
Until ((Current time-$vhStartTime)>†00:00:30†)
```

displays the following window (on Macintosh) for 30 seconds:



```
0 123456789
1
2
3
4
5
6
7
8
9
```

See Also

MESSAGE.

26 Named Selections

version 3

Named selections provide an easy way to manipulate several selections simultaneously. A named selection is an ordered list of records for a table in a process. This ordered list can be given a name and kept in memory. Named selections offer a simple means to preserve in memory the order of the selection and the current record of the selection.

The following commands enable you to work with named selections:

• COPY NAMED SELECTION
• CUT NAMED SELECTION
• USE NAMED SELECTION
• CLEAR NAMED SELECTION

Named selections are created with the COPY NAMED SELECTION and CUT NAMED SELECTION commands. Named selections are generally used to work on one or more selections and to save and later restore an ordered selection. There can be many named selections for each table in a process. To reuse a named selection as the current selection, call USE NAMED SELECTION. When you are done with a named selection, use CLEAR NAMED SELECTION.

Named selections can be process or interprocess in scope.

A named selection is an interprocess named selection if its name is preceded by the symbols (<>) — a “less than” sign followed by a “greater than” sign.

Note: This syntax can be used on both Windows and Macintosh. In addition, on Macintosh only, you can use the diamond (Option-Shift-V on US keyboard).

The scope of an interprocess named selection is identical to the scope of an interprocess variable. An interprocess named selection can be accessed from any process.

A named selection whose name is not prefixed with the symbols (<>) is process in scope and is available only within the process in which it was created.

With 4D Client and 4D Server, an interprocess named selection is available only to the processes of the client that created it. An interprocess named selection is not available to other client machines.

Warning: Creating a named selection requires access to the selection of the table. Since selections are kept on the server and a local process does not have access to server data, do not use named selections within local processes.

Named Selections and Sets

The differences between sets and named selections are:

• A named selection is an ordered list of records; a set is not.
• Sets are very memory efficient, because they require only one bit for each record in the file. Named selections require 4 bytes for each record in the selection.
• Unlike sets, named selections cannot be saved to disk.
• Sets have the standard Intersection, Union and Difference operations; named selections cannot be combined with other named selections.

The similarities between named selections and sets are:

• Like a set, a named selection exists in memory.
• A named selection and a set store references to a record. If records are modified or deleted, the named selection or the set can become invalid.
• Like a set, a named selection "remembers" the current record as of the time the named selection was created.

COPY NAMED SELECTION

version 3

COPY NAMED SELECTION ({table; }name)

Parameter	Type		Description
table	Table	→	Table from which to copy selection, or Default table, if omitted
name	String	→	Name of the named selection to create

Description

COPY NAMED SELECTION copies the current selection of table to the named selection name. The default table for the process is used if the optional table parameter is not specified. The parameter name contains a copy of the selection. The current selection and the current record of Table for the process are not changed.

A named selection does not actually contain the records, but only an ordered list of references to records. Each reference to a record takes 4 bytes in memory. This means that when a selection is copied using the COPY NAMED SELECTION command, the amount of memory required is 4 bytes multiplied by the number of records in the selection. Since named selections reside in memory, you should have enough memory for the named selection as well as the current selection of the table in the process.

Use the CLEAR NAMED SELECTION command to free the memory used by name.

Example

The following example allows you to check if there are other overdue invoices in the [People] table. The selection is sorted and then saved. We search for all records where invoices are due. Then we reuse the selection and clear the named selection in memory. Clearing the named selection in memory is optional, in case the database designer wants to keep the sorted selection for future use:

```
	ALL RECORDS([People])
		`Allow the user to sort the selection
	ORDER BY([People])
		` Save the sorted selection as a named selection
⇒	COPY NAMED SELECTION([People];"UserSort")
		` Search for records where invoices are due
	QUERY([People];[People]InvoiceDue=True)
		` If records are found
```

```
If (Records in selection([People])>0)
        ` Alert the user
    ALERT("Yes, there are overdue invoices on table.")
End if
    ` Reuse the sorted named selection
USE NAMED SELECTION("UserSort")
    ` Remove the selection from memory
CLEAR NAMED SELECTION("UserSort")
```

See Also

CLEAR NAMED SELECTION, CUT NAMED SELECTION, USE NAMED SELECTION.

CUT NAMED SELECTION

CUT NAMED SELECTION ({table; }name)

Parameter	Type		Description
table	Table	→	Table from which to cut selection, or Default table, if omitted
name	String	→	Name of the named selection to create

Description

CUT NAMED SELECTION creates a named selection name and moves the current selection of table to it. This command differs from COPY NAMED SELECTION in that it does not copy the current selection, but moves the current selection of table itself.

After the command has been executed, the current selection of table in the current process becomes empty. Therefore, CUT NAMED SELECTION should not be used while a record is being modified.

CUT NAMED SELECTION is more memory efficient than COPY NAMED SELECTION. With COPY NAMED SELECTION, 4 bytes times the number of selected records is duplicated in memory. With CUT NAMED SELECTION, only the reference to the list is moved.

Example

The following method empties the current selection of a table [Customers]:

```
⇒   CUT NAMED SELECTION([Customers]; "ToBeCleared")
    CLEAR NAMED SELECTION("ToBeCleared")
```

See Also

CLEAR NAMED SELECTION, COPY NAMED SELECTION, USE NAMED SELECTION.

USE NAMED SELECTION

USE NAMED SELECTION (name)

Parameter	**Type**		**Description**
name	String	→	Name of named selection to be used

Description

USE NAMED SELECTION uses the named selection name as the current selection for the table to which it belongs.

When you create a named selection, the current record is "remembered" by the named selection. USE NAMED SELECTION retrieves the position of this record and makes the record the new current record; this command loads the current record. If the current record was modified after name was created, the record should be saved before USE NAMED SELECTION is executed, in order to avoid losing the modified information.

• If COPY NAMED SELECTION was used to create name, the named selection name is copied to the current selection of the table to which name belongs. The named selection name exists in memory until it is cleared. Use the CLEAR NAMED SELECTION command to clear the named selection and free the memory used by name.

• If CUT NAMED SELECTION was used to create name, the current selection is set to name and name no longer exists in memory.

Remember that a named selection is a representation of a selection of records at the moment that the named selection is created. If the records represented by the named selection change, the named selection may no longer be accurate. Therefore, a named selection represents a group of records that does not change frequently. A number of things can invalidate a named selection: modifying a record of the named selection, deleting a record of the named selection, or changing the criterion that determined the named selection.

Also note that during a transaction, temporary record addresses are used. If a named selection is created during a transaction, it may contain addresses that will no longer be valid when the transaction is validated or cancelled, because the records will receive their final and actual address after the transaction is validated.

See Also

COPY NAMED SELECTION, CUT NAMED SELECTION, USE NAMED SELECTION.

CLEAR NAMED SELECTION

version 3

CLEAR NAMED SELECTION (name)

Parameter	**Type**		**Description**
name	String	→	Name of named selection to be cleared

Description

CLEAR NAMED SELECTION clears name from memory and frees the memory used by name. CLEAR NAMED SELECTION does not affect tables, selections, or records. Since named selections use memory, it is good practice to clear named selections when they are no longer needed.

If name was created using the CUT NAMED SELECTION command and then manipulated using the USE NAMED SELECTION command, name no longer exists in memory. In this case, the CLEAR NAMED SELECTION command does not need to be used.

See Also

COPY NAMED SELECTION, CUT NAMED SELECTION, USE NAMED SELECTION.

27 Object Properties

Object Properties

Object Properties commands

The Object Properties commands are:

• FONT
• FONT SIZE
• FONT STYLE
• ENABLE BUTTON
• DISABLE BUTTON
• BUTTON TEXT
• SET CHOICE LIST
• SET ENTERABLE
• SET VISIBLE
• SET FORMAT
• SET FILTER
• SET COLOR
• SET RGB COLOR

The Object Properties commands act on the properties of objects present in forms. They enable you to change the appearance and behavior of the objects while using the forms in the User or Custom menus environment.

Important: The scope of these commands is the form currently being used; changes disappear when you exit the form.

Accessing Objects using their Object Names or their Data Source Names

The Object Properties commands share the same generic syntax described here:

COMMAND NAME({*;} object { ; additional parameters specific to each command)

If you specify the optional * parameter, you indicate an object name (a string) in object.

You can use the @ character within that name if you want to address several objects of the form in one call. The following table shows examples of object names you can specify to this command.

Object Names	**Objects affected by the call**
mainGroupBox	Only the object mainGroupBox.
main@	The objects whose name starts with “main”.
@GroupBox	The objects whose name ends with “GroupBox”.
@Group@	The objects whose name contains “Group”.
main@Btn	The objects whose name starts with “main” and ends with “Btn”.
@	All the objects present in the form.

If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string.

Note: This second syntax is compatible with the previous version of 4th Dimension.

See Also

BUTTON TEXT, DISABLE BUTTON, ENABLE BUTTON, FONT, FONT SIZE, FONT STYLE, SET CHOICE LIST, SET ENTERABLE, SET FILTER, SET FORMAT, SET RGB COLOR, SET VISIBLE.

FONT

version 6.0 (Modified)

FONT ({*; }object; font)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
font	String | Number	→	Font name or Font number

Description

FONT sets the form objects specified by object to be displayed using the font whose name or number you pass in font.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

Examples

1. The following example sets the font for a button named bOK:

```
⇒	FONT (bOK; "Arial")
```

2. The following example sets the font for all the form objects whose name contains "info":

```
⇒	FONT (*;"@info@"; "Times")
```

See Also

FONT SIZE, FONT STYLE.

FONT SIZE

FONT SIZE ({*; }object; size)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
size	Number	→	Font size in points

Description

FONT SIZE sets the form objects specified by object to be displayed using the font size you pass in size.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

The size is any integer between 1 and 255. If the exact font size does not exist, characters are scaled.

The area for the object, as defined in the form, must be large enough to display the data in the new size. Otherwise, the text may be truncated or not displayed at all.

Examples

1. The following example sets the font size for a variable named vtInfo:

```
⇒    FONT SIZE (vtInfo; 14)
```

2. The following example sets the font size for all the form objects whose name starts with "hl":

```
⇒    FONT SIZE (*;"hl@"; 14)
```

See Also

FONT, FONT STYLE.

FONT STYLE

version 6.0 (Modified)

FONT STYLE ({*; }object; styles)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
styles	Number	→	Font style

Description

FONT STYLE sets the form objects specified by object to be displayed using the font style you pass in styles.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

You pass in styles a sum of the constants describing your font style selection. The following are the predefined constants provided by 4D:

Constant	Type	Value
Plain	Long Integer	0
Bold	Long Integer	1
Italic	Long Integer	2
Underline	Long Integer	4
Outline	Long Integer	8
Shadow	Long Integer	16
Condensed	Long Integer	32
Extended	Long Integer	64

Note: On Windows, only the Plain, Bold, Italic and Underline styles are available.

Examples

1. This example sets the font style for a button named bAddNew. The font style is set to bold italic:

```
⇒      FONT STYLE (bAddNew; Bold + Italic)
```

2. This example sets the font style to Plain for all form objects with names starting with "vt":

```
⇒      FONT STYLE (*;"vt@"; Plain)
```

See Also

FONT, FONT SIZE.

ENABLE BUTTON

version 6.0 (Modified)

ENABLE BUTTON ({*; }object)

Parameter	Type		Description
*		→	If specified, object is an Object Name (String) If omitted, object is a Variable
object	Form Object	→	Object Name (if * is specified), or Variable (if * is omitted)

Description

The command ENABLE BUTTON enables the form objects specified by object.

An enabled button or object reacts to mouse clicks and shortcuts.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

This command (despite what its name suggests) can be applied to the following types of object:

• Button, Default Button, 3D Button, Invisible Button, Highlight Button
• Radio Button, 3D Radio Button, Radio Picture
• Check Box, 3D Check Box
• Pop-up menu, Drop-down List, Combo Box, Menu/Drop-down list
• Thermometer, Ruler

Note: It is not practical to use this command with an object that is assigned an automatic action, because 4D changes the state of the control when needed.

Examples

1. This example enables the button bValidate:

```
⇒	ENABLE BUTTON(bValidate)
```

2. This example enables all form objects that have names containing “btn”:

```
⇒	ENABLE BUTTON(*;"@btn@")
```

3. See example for the command BUTTON TEXT.

See Also

BUTTON TEXT, DISABLE BUTTON.

DISABLE BUTTON

version 6.0 (Modified)

DISABLE BUTTON ({*; }object)

Parameter	Type		Description
*		→	If specified, object is an Object Name (String) If omitted, object is a Variable
object	Form Object	→	Object Name (if * is specified), or Variable (if * is omitted)

Description

The command DISABLE BUTTON disables the form objects specified by object.

A disabled button or object does not react to mouse clicks and shortcuts, and is displayed dimmed or grayed out.

Note: Disabling a button or an object does not prevent you from changing its value programmatically.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

This command (despite what its name suggests) can be applied to the following types of object:

• Button, Default Button, 3D Button, Invisible Button, Highlight Button
• Radio Button, 3D Radio Button, Radio Picture
• Check Box, 3D Check Box
• Pop-up menu, Drop-down List, Combo Box, Menu/Drop-down list
• Thermometer, Ruler

Note: It is not practical to use this command with an object that is assigned an automatic action, because 4D changes the state of the control when needed.

Examples

1. This example disables the button bValidate:

```
⇒	DISABLE BUTTON(bValidate)
```

2. This example disables all form objects that have names containing “btn”:

```
⇒	DISABLE BUTTON(*;"@btn@")
```

3. See example for the command BUTTON TEXT.

See Also

BUTTON TEXT, ENABLE BUTTON.

BUTTON TEXT

version 6.0 (Modified)

BUTTON TEXT ({*; }object; buttonText)

Parameter	Type		Description
*		→	If specified, object is an Object Name (String) If omitted, object is a Variable
object	Form Object	→	Object Name (if * is specified), or Variable (if * is omitted)
buttonText	String	→	New title for the button

Description

The command BUTTON TEXT changes the title of the buttons specified by object to the value you pass in buttonText.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

BUTTON TEXT affects only buttons that display text: buttons, check boxes, and radio buttons.

Usually, you will apply this command to one button at a time. The button area must be large enough to accommodate the text; if it is not, the text is truncated. Do not use carriage returns in buttonText.

Example

The following example is the object method of a search button located in the footer area of an output form displayed using MODIFIED SELECTION. The method searches a table; depending on the search results, it enables or disables a button labeled bDelete and changes its title:

```
        QUERY ([People]; [People]Name = vName)
        Case of
            : (Records in selection ([People]) = 0) ` No people found
⇒               BUTTON TEXT (bDelete;" Delete")
                DISABLE BUTTON (bDelete)
            : (Records in selection ([People]) = 1) ` One person found
⇒               BUTTON TEXT (bDelete;"Delete Person")
                ENABLE BUTTON (bDelete)
            : (Records in selection([People]) > 1) ` Many people found
⇒               BUTTON TEXT (bDelete;"Delete People")
                ENABLE BUTTON (bDelete)
        End case
```

See Also

DISABLE BUTTON, ENABLE BUTTON.

SET FORMAT

version 6.0 (Modified)

SET FORMAT ({*; }object; displayFormat)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
displayFormat	String	→	New display format for the object

Description

SET FORMAT sets the display format for the objects specified by object to the format you pass in displayFormat.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

SET FORMAT can be used for both input forms and output forms (displayed or printed) and can be applied to fields, enterable/non-enterable variables, and numeric scrollable areas.

You use a display format suitable to the type of data present in the object:

• To format Boolean fields, pass two values, separated by a semicolon (;).

• To format Date fields or variables, pass Char(n) in displayFormat, where n is one of the following predefined constants provided by 4D:

Constant	Type	Value
Short	Long Integer	1
Abbreviated	Long Integer	2
Long	Long Integer	3
MM DD YYYY	Long Integer	4
Month Date Year	Long Integer	5
Abbr Month Date	Long Integer	6
MM DD YYYY Forced	Long Integer	7

• To format Time fields or variables, pass Char(n) in displayFormat, where n is one of the following predefined constants provided by 4D:

Constant	**Type**	**Value**
HH MM SS	Long Integer	1
HH MM	Long Integer	2
Hour Min Sec	Long Integer	3
Hour Min	Long Integer	4
HH MM AM PM	Long Integer	5

• To format Picture fields or variables, pass Char(n) in displayFormat, where n is one of the following predefined constants provided by 4D:

Constant	**Type**	**Value**
Truncated Centered	Long Integer	1
Scaled to Fit	Long Integer	2
On Background	Long Integer	3
Truncated non Centered	Long Integer	4
Scaled to fit proportional	Long Integer	5
Scaled to fit prop centered	Long Integer	6

For more information about alphanumeric and numeric display formats, see the *4th Dimension Design Reference* manual.

Note: In displayFormat, to use display formats you may have predefined in the Database Properties dialog box, prefix the name of the format with a vertical bar (|).

Examples

1. The following line of code formats the [Employee]Date Hired field to Month Date Year.

```
⇒	SET FORMAT ([Employee]Date Hired; Char(Month Date Year))
```

2. The following example changes the format for a [Company]ZIP Code field according to the length of the value stored in the field:

```
	If (Length ([Company]ZIP Code) = 9)
⇒		SET FORMAT ([Company]ZIP Code; "#####–####")
	Else
⇒		SET FORMAT ([Company]ZIP Code; "#####")
	End if
```

3. The following example sets the format of a Boolean field to display Married and Unmarried, instead of the default Yes and No:

```
⇒	SET FORMAT ([Employee]Marital Status;"Married;Unmarried")
```

See Also

SET FILTER.

SET FILTER

SET FILTER ({*; }object; entryFilter)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
entryFilter	String	→	New data entry filter for the enterable area

Description

SET FILTER sets the entry filter for the objects specified by object to the filter you pass in entryFilter.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

SET FILTER can be used for input and dialog forms and can be applied to fields and enterable variables that accept an entry filter in Design environment.

Passing an empty string in entryFilter removes the current entry filter for the objects.

Note: This command cannot be used with fields located in a subform's list form.

Note: In entryFilter, to use entry filters you may have predefined in the Database Properties dialog box, prefix the name of the filter with a vertical bar (|).

Examples

1. The following example sets the entry filter for a postal code field. If the address is in the U.S., the filter is set to ZIP codes. Otherwise, it is set to allow any entry:

```
      If ([Companies]Country = "US")  ` Set the filter to a ZIP code format
⇒         SET FILTER ([Companies]ZIP Code; "&9#####")
      Else  ` Set the filter to accept alpha and numeric and uppercase the alpha
⇒         SET FILTER ([Companies]ZIP Code; "~@")
      End if
```

2. The following example allows only the letters “a,” “b,” “c,” or “g” to be entered in two places in the field Field:

```
⇒	SET FILTER([Table]Field ;"&"+Char(Double quote)+ "a;b;c;g"+
						Char(Double quote)+"##")
```

Note: This example sets the entry filter to &"a;b;c;g"##.

See Also

SET FORMAT.

SET CHOICE LIST

version 6.0 (Modified)

SET CHOICE LIST ({*; }object; list)

Parameter	Type		Description
*		→	If specified, object is an Object Name (String) If omitted, object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
list	String	→	Name of the list to use as Choice list (as defined in Design environment)

Description

The command SET CHOICE LIST sets the choice list for the objects specified by object to the hierarchical list (defined in the Design environment List Editor) whose name you pass in list.

This command can be applied in an input or dialog form, to fields and enterable variables whose value can be entered as text. The list is displayed during data entry when the user selects the text area.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

Note: This command cannot be used with fields located in a subform's list form.

Example

The following example sets a choice list for a shipping field. If the shipping is overnight, then the choice list is set to shippers who can ship overnight. Otherwise, it is set to the standard shippers:

```
    If ([Shipments]Overnight)
⇒      SET CHOICE LIST([Shipments]Shipper; "Fast Shippers")
    Else
⇒      SET CHOICE LIST([Shipments]Shipper; "Normal Shippers")
    End if
```

SET ENTERABLE

SET ENTERABLE ({*; }entryArea; enterable)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
entryArea	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
enterable	Boolean	→	True for enterable; False for non-enterable

Description

The command SET ENTERABLE makes the form objects specified by object either enterable or non-enterable.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

Using this command is equivalent to selecting Enterable or Non-enterable for a field or variable in the Form Editor's Object Properties window. This command works in subforms only if it is in the form method of the subform.

When the entryArea is enterable (TRUE), the user can move the cursor into the area and enter data. When the entryArea is non-enterable (FALSE), the user cannot move the cursor into the area and cannot enter data. Making an object non-enterable does not prevent you from changing its value programmatically.

Example

The following example sets a shipping field, depending on the weight of the shipment. If the shipment is 1 ounce or less, then the shipper is set to US Mail and the field is set to be non-enterable. Otherwise, the field is set to be enterable.

```
    If ([Shipments]Weight<=1)
       [Shipments]Shipper:="US Mail"
⇒      SET ENTERABLE([Shipments]Shipper;False)
    Else
       SET ENTERABLE([Shipments]Shipper;True)
    End if
```

See Also

DISABLE BUTTON, ENABLE BUTTON, SET VISIBLE.

SET VISIBLE

version 6.0

SET VISIBLE ({*; }object; visible)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object parameter is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
visible	Boolean	→	True for visible, False for invisible

Description

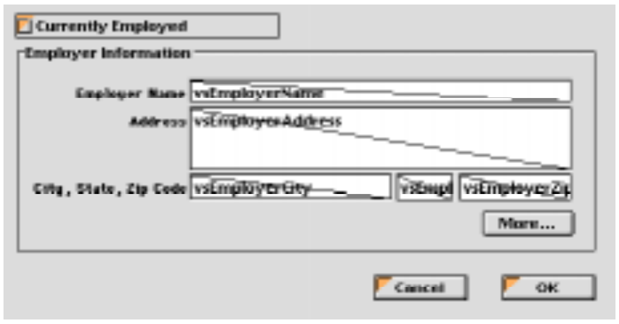
The command SET VISIBLE shows or hides the objects specified by object.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

If you pass visible equal to TRUE, the objects are shown. If you pass visible equal to FALSE, the objects are hidden.

Example

Here is a typical form in the Design environment:

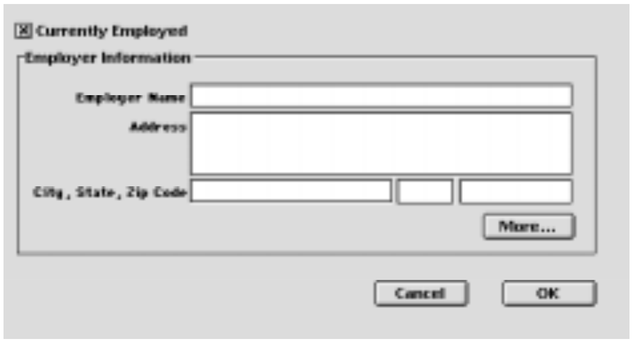


The objects in the **Employer Information** group box each have an object name that contains the expression “employer” (including the group box). When the **Currently Employed** check box is checked, the objects must be visible; when the check box is unchecked, the objects must be invisible.

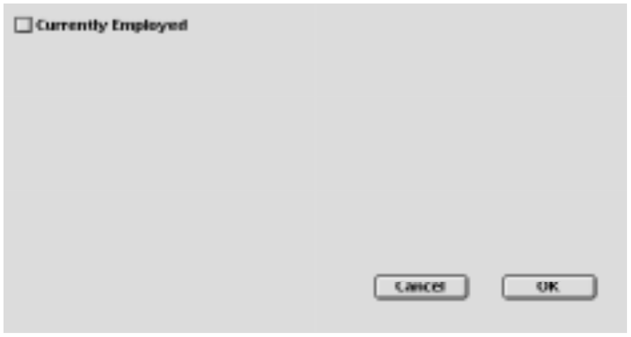
Here is the object method of the check box:

```
    ` cbCurrentlyEmployed Check Box Object Method
  Case of
      : (Form event=On Load)
          cbCurrentlyEmployed:=1

      : (Form event=On Clicked)
              ` Hide or Show all the objects whose name contains "emp"
          SET VISIBLE(*;"@emp@";cbCurrentlyEmployed # 0)
              ` But always keep the check box itself visible
          SET VISIBLE(cbCurrentlyEmployed;True)
  End case
```

Therefore, in the User or Custom Menus environments, the form looks like:



or:



See Also

DISABLE BUTTON, ENABLE BUTTON, SET ENTERABLE.

SET COLOR

version 6.0 (Modified)

SET COLOR ({*; }object; color)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Field or variable	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
color	Number	→	New colors for the object

Description

The command SET COLOR sets the foreground and background colors of the form objects specified by object.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

The color parameter specifies both foreground and background colors. The color is calculated as:

```
Color:=–(Foreground+(256 * Background))
```

where foreground and background are color numbers (from 0 to 255) within the color palette.
Color is always a negative number. For example, if the foreground color is to be 20 and the background color is to be 10, then color is – (20 + (256 * 10)) or –2580.

Note: You can see the color palette in the Form Editor’s Object Properties window.

The numbers of the commonly used colors are provided by the following predefined constants:

Constant	Type	Value
White	Long Integer	0
Yellow	Long Integer	1
Orange	Long Integer	2
Red	Long Integer	3
Purple	Long Integer	4
Dark Blue	Long Integer	5
Blue	Long Integer	6
Light Blue	Long Integer	7
Green	Long Integer	8
Dark Green	Long Integer	9
Dark Brown	Long Integer	10
Dark Grey	Long Integer	11
Light Grey	Long Integer	12
Brown	Long Integer	13
Grey	Long Integer	14
Black	Long Integer	15

Note: While SET COLOR works with indexed colors within the default 4D color palette, version 6 introduces the command SET RGB COLOR, which allows you to work with any RGB color.

Example

The following example sets the color for a button named bInfo. The color is set to the values of the two variables named vForeground and vBackground:

```
⇒	SET COLOR (bInfo; – (vForeground + (256 * vBackground)))
```

See Also

SET RGB COLOR.

SET RGB COLOR

Object Properties

version 6.0

SET RGB COLOR ({*; }object; foregroundColor; backgroundColor)

Parameter	Type		Description
*		→	If specified, Object is an Object Name (String) If omitted, Object is a Field or a Variable
object	Form Object	→	Object Name (if * is specified), or Field or Variable (if * is omitted)
foregroundColor	Number	→	RGB color value for Foreground color
backgroundColor	Number	→	RGB color value for Background color

Description

The command SET RGB COLOR changes the foreground and background colors of the objects specified by object and the optional * parameters.

If you specify the optional * parameter, you indicate an object name (a string) in object. If you omit the optional * parameter, you indicate a field or a variable in object. In this case, you specify a field or variable reference (field or variable objects only) instead of a string. For more information about object names, see the section Object Properties.

You indicate RGB color values in foreground and background. An RGB value is a 4-byte Long Integer whose format (0x00RRGGBB) is described in the following table (bytes are numbered from 0 to 3, from right to left):

Byte	Description
3	Must be zero if absolute RGB color
2	Red component of the color (0..255)
1	Green component of the color (0..255)
0	Blue component of the color (0..255)

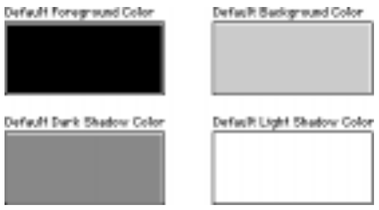
The following table shows some examples of RGB color values:

Value	Description
0x00000000	Black
0x00FF0000	Bright Red
0x0000FF00	Bright Green
0x000000FF	Bright Blue
0x007F7F7F	Gray
0x00FFFF00	Bright Yellow
0x00FF7F7F	Red Pastel
0x00FFFFFF	White

Alternatively, you can specify one of the four automatic colors used by 4th Dimension for drawing objects whose colors are set automatically. The following predefined constants are provided by 4th Dimension:

Constant	Type	Value
Default foreground color	Long Integer	-1
Default background color	Long Integer	-2
Default dark shadow color	Long Integer	-3
Default light shadow color	Long Integer	-4

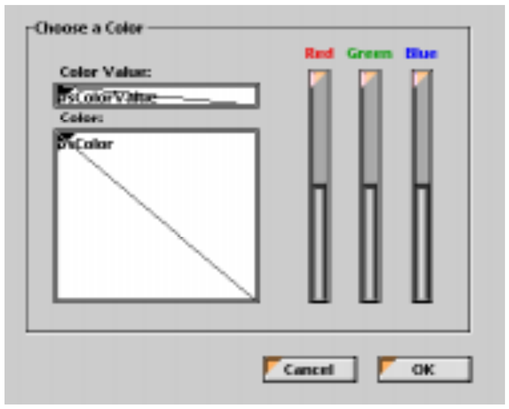
These colors (on a standard system) are shown here:



WARNING: On Windows, these automatic colors are system dependent. If you change your system colors in the Colors Windows Control Panel, 4th Dimension will adjust its automatic colors accordingly. Use the automatic color values for setting objects to the system colors, not for setting them to the example colors shown above.

Examples

This form contains the two non-enterable variables vsColorValue and vsColor as well as the three thermometers: thRed, thGreen, and thBlue.



Here are the methods for these objects:

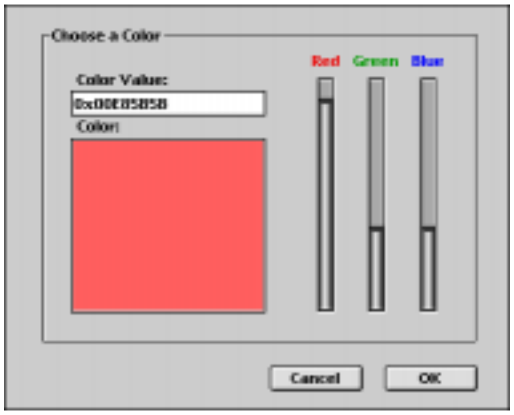
```
    ` vsColorValue non-enterable Object Method
Case of
    : (Form event=On Load)
        vsColorValue:="0x00000000"
End case

    ` vsColor non-enterable variable Object Method
Case of
    : (Form event=On Load)
        vsColor:=""
⇒       SET RGB COLOR(vsColor;0x00FFFFFF;0x0000)
End case

    ` thRed Thermometer Object Method
CLICK IN COLOR THERMOMETER

    ` thGreen Thermometer Object Method
CLICK IN COLOR THERMOMETER

    ` thBlue Thermometer Object Method
CLICK IN COLOR THERMOMETER
```

The project method called by the three thermometers is:

```
    ` CLICK IN COLOR THERMOMETER Project Method
⇒ SET RGB COLOR(vsColor;0x00FFFFFF;(thRed << 16)+(thGreen << 8)+thBlue)
vsColorValue:=String((thRed << 16)+(thGreen << 8)+thBlue;"&x")
If (thRed=0)
    vsColorValue:=Substring(vsColorValue;1;2)+"0000"+Substring(vsColorValue;3)
End if
```

Note the use of the Bitwise operators for calculating the color value from the thermometer values.

In the User or Custom Menus environments, the form looks like this:



See Also

Bitwise Operators, SET COLOR.

28 Obsolete commands

SEARCH BY INDEX

Obsolete commands

version 3

SEARCH BY INDEX

This command is still present in 4th Dimension for compatibility with 4D version 1. For any new programming, use the command QUERY.

WARNING: This command will disappear in future versions. Please do not use it.

SORT BY INDEX

Obsolete commands

version 3

SORT BY INDEX

This command is still present in 4th Dimension for compatibility with 4D version 1. For any new programming, use the command ORDER BY.

WARNING: This command will disappear in future versions. Please do not use it.

ON SERIAL PORT CALL

version 3

ON SERIAL PORT CALL (serialMethod{; process})

Parameter	Type		Description
serialMethod	String	→	Method to be invoked
process	String	→	Process name

Description

This command has been retained for compatibility with 4th Dimension version 2. In most cases, it would be more efficient to use the RECEIVE BUFFER command.

ON SERIAL PORT CALL installs serialMethod as an interrupt method for managing serial port events in a separate process. The interrupt method is automatically called by 4th Dimension when a character enters the serial port buffer opened with SET CHANNEL. An empty string for serialMethod turns off serial port event handling. The optional process parameter names the process created by the ON SERIAL PORT CALL command. If process is prefixed with a dollar sign ($), process is a local process.

Since the interrupt method has been installed as a separate process, it runs concurrently with all other processes.

4th Dimension automatically calls the interrupt method when the serial port buffer contains one or more characters. If you decide to do nothing with the buffer contents, remember to clear the buffer contents by calling RECEIVE BUFFER. If you do not clear the buffer, 4th Dimension will call your installed method again and again.

Warning: With 4th Dimension version 3 , it is better to create your own process in which you can handle the serial port any way you want. From this process, you can receive or send data from and to the serial port, then communicate the data to other processes using interprocess communication. The use of ON SERIAL PORT CALL is discouraged.

Example

The following line installs an interrupt method called Interruption:

```
⇒	ON SERIAL PORT CALL ("Interruption")
```

The Interruption method takes whatever is in the serial buffer and concatenates it into a variable called <>GotIt. This variable can then be read later by other parts of the application. Here is the Interruption method:

```
	RECEIVE BUFFER ($v) ` Read the serial port buffer
	<>GotIt := <>GotIt + $v ` Save the data
```

The following line removes the interrupt method:

```
⇒	ON SERIAL PORT CALL ("")
```

29 On a Series

On a Series

version 3

The functions of this theme perform calculations on a series of values.

The Average, Max, Min, Sum, Sum squares, Std deviation, and Variance functions can be applied to fields or subfields. In the case of a field, they are applied to a selection of records. In the case of a subfield, they are applied to a selection of the subrecords of the current record. Note that the
Sum squares, Std deviation, and Variance functions can be used on a field only during printing.

These functions work on numeric data only. Each of these functions returns a numeric value.

Using a field

When Average, Max, Min, or Sum are used on a field outside a printing operation, they may have to load each record in the current selection to calculate the result. If there are many records, this process may take some time. To avoid this, index the field.

When these functions are used in a report, they behave differently than at other times. This is because the report itself must load each record. Use these functions in a form or object method when printing with the PRINT SELECTION command or when printing by choosing Print from the File menu in the User environment.

When you use these functions in a report, the values that are returned are reliable only at break level 0, and only when break processing is turned on. This means that they are useful only at the end of a report, after all the records have been processed.

You would use these functions only in an object method for a non-enterable area that is included in the B0 Break area.

Remember that the field passed as a parameter to the statistical function must be a numeric.

See Also

Average, Max, Min, Std deviation, Sum, Sum Squares, Variance.

Sum

version 3

Sum (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the sum
Function result	Number	←	Sum for series

Description

The command Sum returns the sum (total of all values) for series. If series is an indexed field, the index is used to total the values.

Example

The following example is an object method for a variable that vTotal placed in a form. The object method assigns the sum of all salaries to vTotal:

```
⇒    vTotal:=Sum([Employees]Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
     ALL RECORDS ([Employees])
     ORDER BY ([Employees];[Employees]LastNm;>)
     BREAK LEVEL (1)
     ACCUMULATE ([Employees]Salary)
     OUTPUT FORM ([Employees];"PrintForm")
     PRINT SELECTION ([Employees])
```

Note: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

See Also

ACCUMULATE, Average, BREAK LEVEL, Max, Min, ORDER BY, PRINT SELECTION, Subtotal.

Average

version 3

Average (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the average
Function result	Number	←	Arithmetic mean (average) of series

Description

Average returns the arithmetic mean (average) of series. If series is an indexed field, the index is used to find the average.

Example

The following example sets the variable vAverage that is in the B0 Break area of an output form. The line of code is the object method for vAverage. The object method is not executed until the level 0 break:

```
⇒	vAverage := Average ([Employees] Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
	ALL RECORDS ([Employees])
	ORDER BY ([Employees];[Employees]LastNm;>)
	BREAK LEVEL (1)
	ACCUMULATE ([Employees]Salary)
	OUTPUT FORM ([Employees];"PrintForm")
	PRINT SELECTION ([Employees])
```

Note: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

See Also

ACCUMULATE, BREAK LEVEL, Max, Min, ORDER BY, PRINT SELECTION, Subtotal, Sum.

Min

On a Series

version 3

Min (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the minimum value
Function result	Number	←	Minimum value in series

Description

Min returns the minimum value in series. If series is an indexed field, the index is used to find the minimum value.

Examples

1. The following example is an object method for the variable vMin placed in the break 0 portion of the form. The variable is printed at the end of the report. The object method assigns the minimum value of the field to the variable, which is then printed in the last break of the report:

```
⇒	vMin:=Min([Employees]Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
	ALL RECORDS ([Employees])
	ORDER BY ([Employees];[Employees]LastNm;>)
	BREAK LEVEL (1)
	ACCUMULATE ([Employees]Salary)
	OUTPUT FORM ([Employees];"PrintForm")
	PRINT SELECTION ([Employees])
```

NOTE: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

2. The following example finds the lowest sale amount of an employee and displays the result in an alert box. The sales amounts are stored in the subfield [Employees]SalesDollars:

```
⇒	ALERT ("Minimum sale = " + String(Min([Employees]SalesDollars)))
```

See Also

Execute on server, Execute on server, GET PROCESS VARIABLE, Max, Processes, SET PROCESS VARIABLE.

Max

On a Series

version 3

Max (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the maximum value
Function result	Number	←	Maximum value in series

Description

Max returns the maximum value in series. If series is an indexed field, the index is used to find the maximum value.

Example

The following example is an object method for the variable vMax placed in the break 0 portion of the form. The variable is printed at the end of the report. The object method assigns the maximum value of the field to the variable, which is then printed in the last break of the report.

```
⇒      vMax := Max ([Employees] Salary)
```

The following method is called to print the records in the selection and to activate break processing:

```
       ALL RECORDS ([Employees])
       ORDER BY ([Employees];[Employees]LastNm;>)
       BREAK LEVEL (1)
       ACCUMULATE ([Employees]Salary)
       OUTPUT FORM ([Employees];"PrintForm")
       PRINT SELECTION ([Employees])
```

Note: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

See Also

Min.

Std deviation

Std deviation (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the standard deviation
Function result	Number	←	Standard deviation of series

Description

Std deviation returns the standard deviation of series. If series is an indexed field, the index is used to find the standard deviation. You can only use a field with this function when printing a report.

Example

The following example is an object method for the variable vDeviate. The object method assigns the standard deviation for a data series to vDeviate:

```
⇒	vDeviate := Std deviation ([Table1]DataSeries)
```

The following method is called to print the records in the selection and to activate break processing:

```
	ALL RECORDS ([Table1])
	ORDER BY ([Table1];[Table1]DataSeries;>)
	BREAK LEVEL (1)
	ACCUMULATE ([Table1]DataSeries)
	OUTPUT FORM ([Table1];"PrintForm")
	PRINT SELECTION ([Table1])
```

NOTE: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

See Also

Average, Sum, Sum Squares, Variance.

Variance

version 3

Variance (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the variance
Function result	Number	←	Variance of series

Description

Variance returns the variance for series. If series is an indexed field, the index is used to find the variance. You can only use a field with this function when printing a report.

Example

The following example is an object method for the variable var. The object method assigns the sum of squares for a data series to var:

```
⇒      var:= Variance (Students]Grades)
```

The following method is called to print the records in the selection and to activate break processing:

```
    ALL RECORDS ([Students])
    ORDER BY ([Students];[Students]Class;>)
    BREAK LEVEL (1)
    ACCUMULATE ([Students]Grades)
    OUTPUT FORM ([Students];"PrintForm")
    PRINT SELECTION ([Students])
```

NOTE: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

See Also

Average, Std deviation, Sum, Sum squares.

Sum squares

On a Series

version 3

Sum squares (series) → Number

Parameter	Type		Description
series	Field or subfield	→	Data for which to return the sum of squares
Function result	Number	←	Sum of squares of series

Description

Sum squares returns the sum of the squares of series. If series is an indexed field, the index is used to find the sum of the squares. You can only use a field with this function when printing a report.

Example

The following example is an object method for the variable vSquares. The object method assigns the sum of squares for a data series to vSquares. The vSquares variable is printed in the last break of the report:

```
⇒	vSquares:=Sum squares ([Table1]DataSeries)
```

The following method is called to print the records in the selection and to activate break processing:

```
	ALL RECORDS ([Table1])
	ORDER BY ([Table1];[Table1]DataSeries;>)
	BREAK LEVEL (1)
	ACCUMULATE ([Table1]DataSeries)
	OUTPUT FORM ([Table1];"PrintForm")
	PRINT SELECTION ([Table1])
```

NOTE: The parameter to the BREAK LEVEL command should be equal to the number of breaks in your report. For more information about break processing, refer to the printing commands.

See Also

Average, Std deviation, Sum, Variance.

30 Operators

Operators are symbols used to specify operations performed between expressions. They:

• Perform calculations on numbers, dates, and times.
• Perform string operations, Boolean operations on logical expressions, and specialized operations on pictures.
• Combine simple expressions to generate new expressions.

Precedence

The order in which an expression is evaluated is called *precedence*. 4 th Dimension has a strict left-to-right precedence, in which algebraic order is not observed. For example:

```
3 + 4 * 5
```

returns 35, because the expression is evaluated as 3 + 4, yielding 7, which is then multiplied by 5, with the final result of 35.

To override the left-to-right precedence, you MUST use parentheses. For example:

```
3 + (4 * 5)
```

returns 23 because the expression (4 * 5) is evaluated first, because of the parentheses. The result is 20, which is then added to 3 for the final result of 23.

Parentheses can be nested inside other sets of parentheses. Be sure that each left parenthesis has a matching right parenthesis to ensure proper evaluation of expressions. Lack of, or incorrect use of parentheses can cause unexpected results or invalid expressions. Furthermore, if you intend to compile your applications with 4D Compiler, you must have matching parentheses—the compiler detects a missing parenthesis as a syntax error.

The Assignment Operator

You MUST distinguish the assignment operator := from the other operators. Rather than combining expressions into a new one, the assignment operator copies the value of the expression to the right of the assignment operator into the variable or field to the left of the operator. For example, the following line places the value 4 (the number of characters in the word Acme) into the variable named MyVar. MyVar is then typed as a numeric value.

```
MyVar := Length ("Acme")
```

Important: Do NOT confuse the assignment operator := with the equality comparison operator =.

The other operators provided by the 4D language are described in the following sections:

String Operators
See the section String Operators.

Numeric Operators
See the section Numeric Operators.

Date Operators
See the section Date Operators.

Time Operators
See the section Time Operators.

Comparison Operators
See the section Comparison Operators.

Logical Operators
See the section Logical Operators.

Picture Operators
See the section Picture Operators.

Bitwise Operators
See the section Bitwise Operators.

See Also

Constants, Data Types, Identifiers.

String Operators

An expression that uses a string operator returns a string. The following table shows the string operators:

Operation	Syntax	Returns	Expression	Value
Concatenation	String + String	String	"abc" + "def"	"abcdef"
Repetition	String * Number	String	"ab" * 3	"ababab"

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, Time Operators.

Numeric Operators

Operators

version 6.0

An expression that uses a numeric operator returns a number. The following table shows the numeric operators:

Operation	Syntax	Returns	Expression	Value
Addition	Number + Number	Number	2 + 3	5
Subtraction	Number – Number	Number	3 – 2	1
Multiplication	Number * Number	Number	5 * 2	10
Division	Number /Number	Number	5 / 2	2.5
Longint division	Number \ Number	Number	5 \ 2	2
Modulo	Number % Number	Number	5 % 2	1
Exponentiation	Number ^ Number	Number	2 ^ 3	8

Modulo Operator

The modulo operator % divides the first number by the second number and returns a whole number remainder. Here are some examples:

- 10 % 2 returns 0 because 10 is evenly divided by 2.
- 10 % 3 returns 1 because the remainder is 1.
- 10.5 % 2 returns 0 because the remainder is not a whole number.

WARNING: The modulo operator % returns significant values with numbers that are in the Long Integer range (from minus 2^31 to 2^31 minus one). To calculate the modulo with numbers outside of this range, use the Mod command.

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Operators, Picture Operators, String Operators, Time Operators.

Date Operators

Operators

version 6.0

An expression that uses a date operator returns a date or a number, depending on the operation. All date operations will result in an accurate date, taking into account the change between years and leap years. The following table shows the date operators:

Operation	Syntax	Returns	Expression	Value
Date difference	Date – Date	Number	!1/20/97! – !1/1/97!	19
Day addition	Date + Number	Date	!1/20/97! + 9	!1/29/97!
Day subtraction	Date – Number	Date	!1/20/97! – 9	!1/11/97!

See Also

Bitwise Operators, Comparison Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, String Operators, Time Operators.

Time Operators

An expression that uses a time operator returns a time or a number, depending on the operation. The following table shows the time operators:

Operation	**Syntax**	**Returns**	**Expression**	**Value**
Addition	Time + Time	Time	?02:03:04? + ?01:02:03?	?03:05:07?
Subtraction	Time – Time	Time	?02:03:04? – ?01:02:03?	?01:01:01?
Addition	Time + Number	Number	?02:03:04? + 65	7449
Subtraction	Time – Number	Number	?02:03:04? – 65	7319
Multiplication	Time * Number	Number	?02:03:04? * 2	14768
Division	Time / Number	Number	?02:03:04? / 2	3692
Longint division	Time \ Number	Number	?02:03:04? \ 2	3692
Modulo	Time % Number	Number	?02:03:04? % 2	0

Tips

(1) To obtain a time expression from an expression that combines a time expression with a number, use the commands Time and Time string.

Example:

```
	` The following line assigns to $vlSeconds the number of seconds that will be elasped
	` between midnight and one hour from now
$vlSeconds:=Current Time+3600

	` The following line assigns to $vHSoon the time it will be in one hour
$vhSoon:=Time(Time string(Current time+3600))
```

The second line could be written in a simpler way:

```
	` The following line assigns to $vHSoon the time it will be in one hour
$vhSoon:=Current time+?00:01:00?
```

However, while developing your application, you may encounter situations where a delay, expressed in seconds and added to a time value, is only available to you as a numeric value.
In this case, use the next tip.

(2) Some situations may require you to convert a time expression into a numeric expression.

For example, you open a document using Open document, which returns a Document Reference (DocRef) that is formally a time expression. Later, you want to pass that DocRef to a 4D Extension routine that expects a numeric value as document reference. In such a case, use the addition with 0 (zero) to get a numeric value from the time value, but without changing its value.

Example:

```
   ` Select and open a document
$vhDocRef:=Open document("")
If (OK=1)
      ` Pass the DocRef time expression
      ` as a numeric expresssion to a 4D Extension routine
   DO SOMETHING SPECIAL (0+$vhDocRef)
End if
```

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, String Operators.

Comparison Operators

Operators

version 6.0

The tables in this section show the comparison operators as they apply to string, numeric, date, time, and pointer expressions. An expression that uses a comparison operator returns a Boolean value, either TRUE or FALSE.

String Comparisons

Operation	Syntax	Returns	Expression	Value
Equality	String = String	Boolean	"abc" = "abc"	True
			"abc" = "abd"	False
Inequality	String # String	Boolean	"abc" # "abd"	True
			"abc" # "abc"	False
Greater than	String > String	Boolean	"abd" > "abc"	True
			"abc" > "abc"	False
Less than	String < String	Boolean	"abc" < "abd"	True
			"abc" < "abc"	False
Greater than or equal to	String >= String	Boolean	"abd" >= "abc"	True
			"abc" >= "abd"	False
Less than or equal to	String <= String	Boolean	"abc" <= "abd"	True
			"abd" <= "abc"	False

Numeric Comparisons

Operation	Syntax	Returns	Expression	Value
Equality	Number = Number	Boolean	10 = 10	True
			10 = 11	False
Inequality	Number # Number	Boolean	10 #11	True
			10 # 10	False
Greater than	Number > Number	Boolean	11 > 10	True
			10 > 11	False
Less than	Number < Number	Boolean	10 < 11	True
			11 < 10	False
Greater than or equal to	Number >= Number	Boolean	11 >= 10	True
			10 >= 11	False
Less than or equal to	Number <= Number	Boolean	10 <= 11	True
			11 <= 10	False

Date Comparisons

Operation	Syntax	Returns	Expression	Value
Equality	Date = Date	Boolean	!1/1/97! =!1/1/97!	True
			!1/20/97! =!1/1/97!	False
Inequality	Date # Date	Boolean	!1/20/97! # !1/1/97!	True
			!1/1/97! # !1/1/97!	False
Greater than	Date > Date	Boolean	!1/20/97! > !1/1/97!	True
			!1/1/97! > !1/1/97!	False
Less than	Date < Date	Boolean	!1/1/97! < !1/20/97!	True
			!1/1/97! < !1/1/97!	False
Greater than or equal to	Date >= Date	Boolean	!1/20/97! >=!1/1/97!	True
			!1/1/97!>=!1/20/97!	False
Less than or equal to	Date <= Date	Boolean	!1/1/97!<=!1/20/97!	True
			!1/20/97!<=!1/1/97!	False

Time Comparisons

Operation	Syntax	Returns	Expression	Value
Equality	Time = Time	Boolean	?01:02:03? = ?01:02:03?	True
			?01:02:03? = ?01:02:04?	False
Inequality	Time # Time	Boolean	?01:02:03? # ?01:02:04?	True
			?01:02:03? # ?01:02:03?	False
Greater than	Time > Time	Boolean	?01:02:04? > ?01:02:03?	True
			?01:02:03? > ?01:02:03?	False
Less than	Time < Time	Boolean	?01:02:03? < ?01:02:04?	True
			?01:02:03? < ?01:02:03?	False
Greater than or equal to	Time >= Time	Boolean	?01:02:03? >=?01:02:03?	True
			?01:02:03? >=?01:02:04?	False
Less than or equal to	Time <= Time	Boolean	?01:02:03? <=?01:02:03?	True
			?01:02:04? <=?01:02:03?	False

Pointer comparisons

With:

```
    ` vPtrA and vPtrB point to the same object
vPtrA:=->anObject
vPtrB:=->anObject
    ` vPtrC points to another object
vPtrC:=->anotherObject
```

Operation	Syntax	Returns	Expression	Value
Equality	Pointer = Pointer	Boolean	vPtrA = vPtrB	True
			vPtrA = vPtrC	False
Inequality	Pointer # Pointer	Boolean	vPtrA # vPtrC	True
			vPtrA # vPtrB	False

More about string comparisons

• Strings are compared on a character-by-character basis.

• When strings are compared, the case of the characters is ignored; thus, "a"="A" returns TRUE. To test if the case of two characters is different, compare their ASCII codes. For example, the following expression returns FALSE:

```
Ascii ("A") = Ascii ("a") ` because 65 is not equal to 97
```

• When strings are compared, diacritical characters are compared using the system character comparison table of your computer. For example, the following expressions return TRUE:

```
"n" = "ñ"
"n" = "Ñ"
"A"="å"
    ` and so on
```

• The wildcard character (@) can be used in any string comparison to match any number of characters. For example, the following expression is TRUE:

```
"abcdefghij" = "abc@"
```

The wildcard character must be used within the second operand (the string on the right side) in order to match any number of characters. The following expression is FALSE, because the @ is considered only as a one character in the first operand:

```
"abc@" = "abcdefghij"
```

The wildcard means “one or more characters or nothing”. The following expressions are TRUE:

```
"abcdefghij" = "abcdefghij@"
"abcdefghij" = "@abcdefghij"
"abcdefghij" = "abcd@efghij"
"abcdefghij" = "@abcdefghij@"
"abcdefghij" = "@abcde@fghij@"
```

On the other hand, whatever the case, a string comparison with two consecutive wildcards will always return FALSE. The following expression is FALSE:

```
"abcdefghij" = "abc@@"
```

Tip
If you obtain a string from data entry, that string may contain the @ character—you cannot treat this wildcard like the other characters. Let’s consider the following example:

```
$vsValue:=Request("Enter the value you are looking for:")
If (OK=1)
    QUERY ([Customers];[Customers]Name=$vsValue+"@")
End if
```

A value is requested, using the Request command. Then this value is used for a “begins with” query. Two consecutive @ characters, as explained previously, forces the comparison result to FALSE, so you can append the @ to the value only if the last character is not already a @.
You can do so in this revised example:

```
$vsValue:=Request("Enter the value you are looking for:")
If (OK=1)
    If (Ascii($vsValue[[Length($vsValue)]])#64)
        $vsValue:=$vsValue+"@"
    End if
    QUERY ([Customers];[Customers]Name=$vsValue)
End if
```

You must use the Ascii command, because the following expression (if $vsValue is not empty) always returns TRUE:

```
$vsValue[[Length($vsValue)]]="@"
```

Continuing with this example, the string entered in the request dialog box may contain several @ characters and even strings like "@@D@OE@@@". The following code will eliminate all the @ characters present in a string:

```
    ` No at signs Project Method
    ` No at signs ( String ) -> String
    ` No at signs ( Any string ) -> String with no @
$0:=""
For ($vlChar;1;Length($1))
    If (Ascii($1[[$vlChar]])#64)
        $0:=$0+$1[[$vlChar]]
    End if
End for
```

In other words, this small project method does the same thing as the command Replace string. However, it is necessary (and it uses ASCII codes) because the following Replace string expression will always return an empty string:

```
Replace String($vsValue;"@";"") ` All characters are removed
```

Finally, the example becomes:

```
$vsValue:=Request("Enter the value you are looking for:")
If (OK=1)
    QUERY ([Customers];[Customers]Name=Not at signs ($vsValue)+"@")
End if
```

With this code, the query will always be a “begins with” query, no matter what string is entered in the request dialog box.

See Also

Bitwise Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, Time Operators.

Logical Operators

4th Dimension supports two logical operators that work on Boolean expressions: conjunction (AND) and inclusive disjunction (OR). A logical AND returns TRUE if both expressions are TRUE. A logical OR returns TRUE if at least one of the expressions is TRUE.

4th Dimension also provides the Boolean functions True, False, and Not. For more information, see the descriptions of these commands.

The following table shows the logical operators:

Operation	Syntax	Returns	Expression	Value
AND	Boolean & Boolean	Boolean	("A" = "A") & (15 # 3)	True
			("A" = "B") & (15 # 3)	False
			("A" = "B") & (15 = 3)	False
OR	Boolean | Boolean	Boolean	("A" = "A") | (15 # 3)	True
			("A" = "B") | (15 # 3)	True
			("A" = "B") | (15 = 3)	False

The following is the truth table for the AND logical operator:

Expr1	Expr2	Expr1 & Expr2
True	True	True
True	False	False
False	True	False
False	False	False

The following is the truth table for the OR logical operator:

Expr1	Expr2	Expr1 | Expr2
True	True	True
True	False	True
False	True	True
False	False	False

Tip

If you need to calculate the exclusive disjunction between Expr1 and Expr2, evaluate:

(Expr1 | Expr2) & **Not**(Expr1 & Expr2)

See Also

Bitwise Operators, Comparison Operators, Date Operators, Numeric Operators, Operators, Picture Operators, String Operators, Time Operators.

Picture Operators

An expression that uses a picture operator returns a picture. The following table shows the picture operators.

Operation	Syntax	Action
Horizontal concatenation	Pict1 + Pict2	Add Pict2 to the right of Pict1
Vertical concatenation	Pict1 / Pict2	Add Pict2 to the bottom of Pict1
Exclusive superimposition	Pict1 & Pict2	Perform an XOR on Pict1 and Pict2
Inclusive superimposition	Pict1 | Pict2	Perform a OR on Pict1 and Pict2
Horizontal move	Picture + Number	Move Picture horizontally Number pixels
Vertical move	Picture / Number	Move Picture vertically Number pixels
Resizing	Picture * Number	Resize Picture by Number ratio
Horizontal scaling	Picture *+ Number	Resize Picture horizontally by Number ratio
Vertical scaling	Picture */ Number	Resize Picture vertically by Number ratio

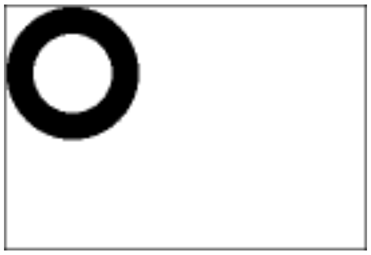
The two operators & and | always return a bitmapped picture, no matter what the nature of the two source pictures. The reason is that 4th Dimension first draws the pictures into memory bitmaps, then calculates the resulting picture by performing graphical exclusive or inclusive OR on the pixels of the bitmaps.

The other picture operators return vectorial pictures if the two source pictures are vectorial. Remember, however, that pictures printed by the display format On Background are printed bitmapped.

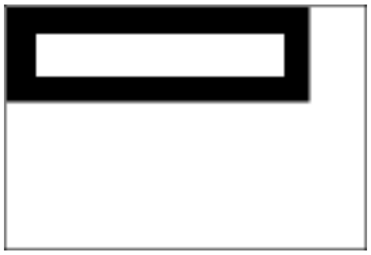
Examples

In the following examples, all of the pictures are shown using the display format On Background.

Here is the picture circle:



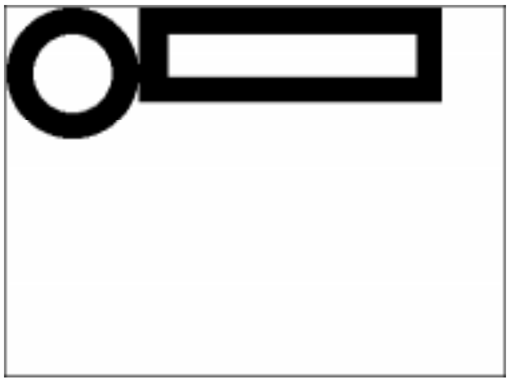
Here is the picture rectangle:



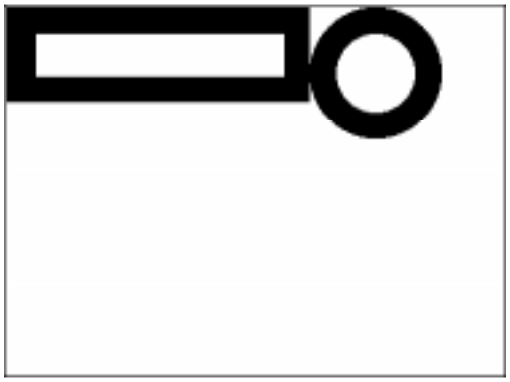
In the following examples, each expression is followed by its graphical representation.

• Horizontal concatenation

circle + rectangle ` Place the rectangle on the right of the circle

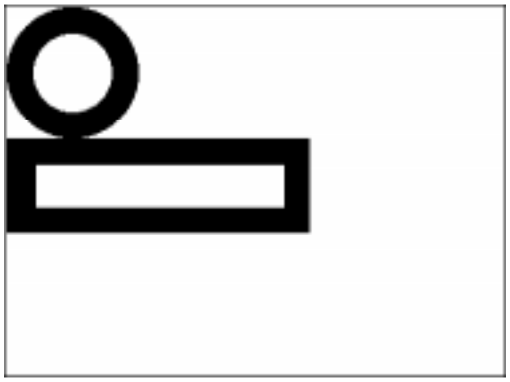


rectangle + circle ` Place the circle on the right of the rectangle

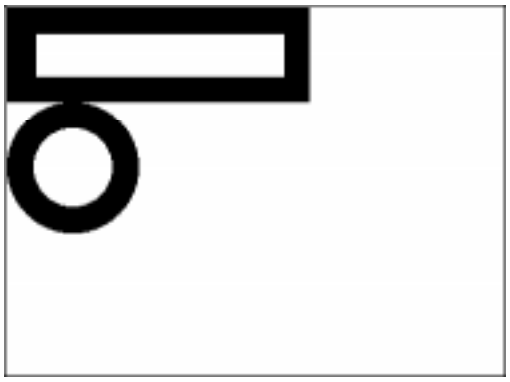


• Vertical concatenation

```
circle / rectangle ` Place the rectangle under the circle
```

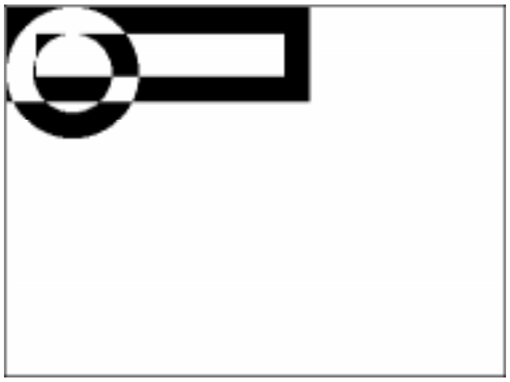


```
rectangle / circle ` Place the circle under the rectangle
```



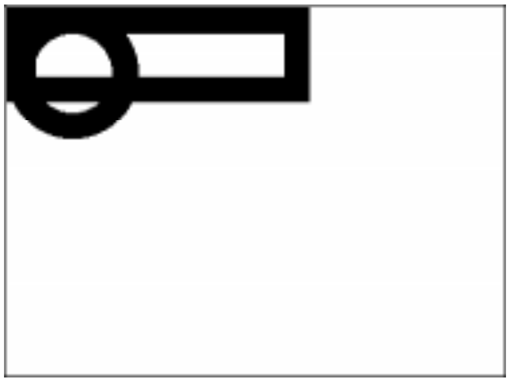
• Exclusive superimposition (XOR)

circle & rectangle ` Exclusive OR of the two pictures



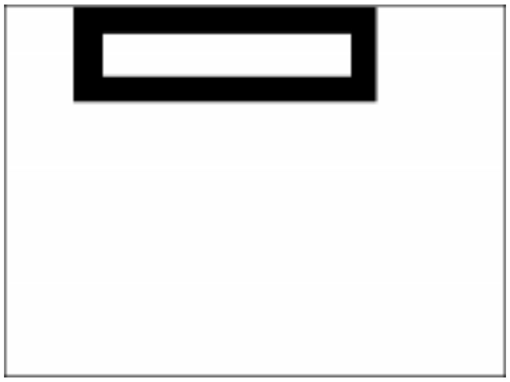
• Inclusive superimposition (OR)

circle | rectangle ` Inclusive OR of the two pictures

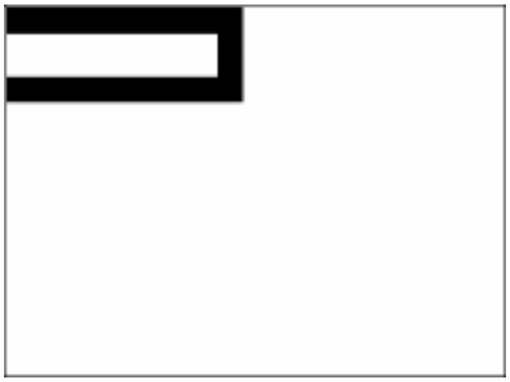


• Horizontal move

```
rectangle + 50 ` Move the rectangle 50 pixels to the right
```

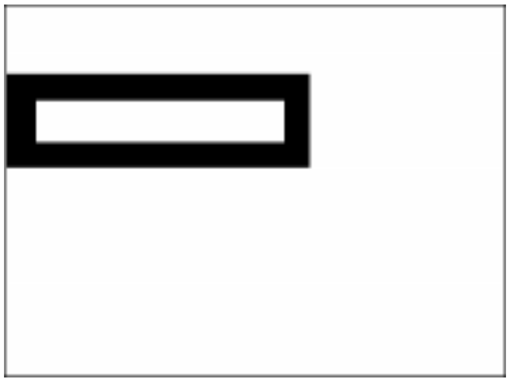


```
rectangle - 50 ` Move the rectangle 50 pixels to the left
```

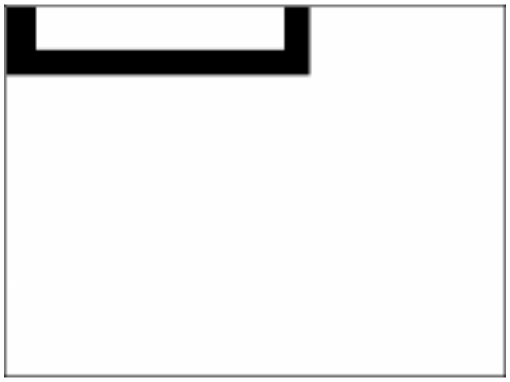


• Vertical move

```
rectangle /50 ` Move down the rectangle by 50 pixels
```

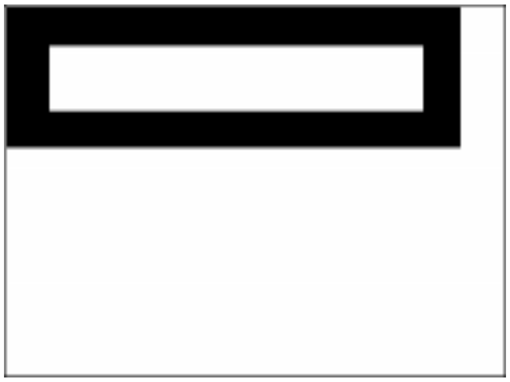


```
rectangle /-20 ` Move up the rectangle by 20 pixels
```

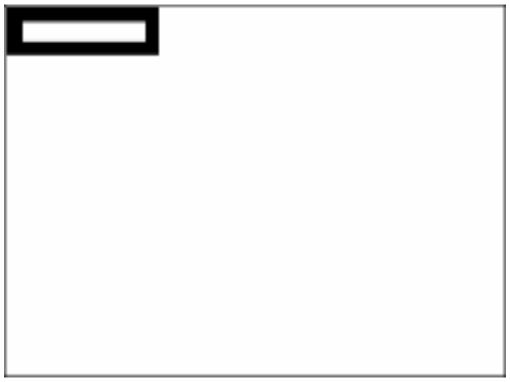


• Resize

```
rectangle * 1.5 ` The rectangle becomes 50% bigger
```

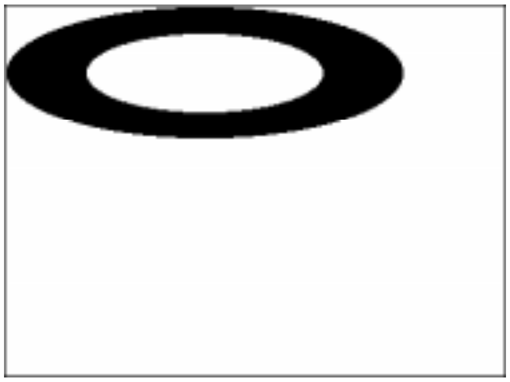


```
rectangle * 0.5 ` The rectangle becomes 50% smaller
```



• Horizontal scaling

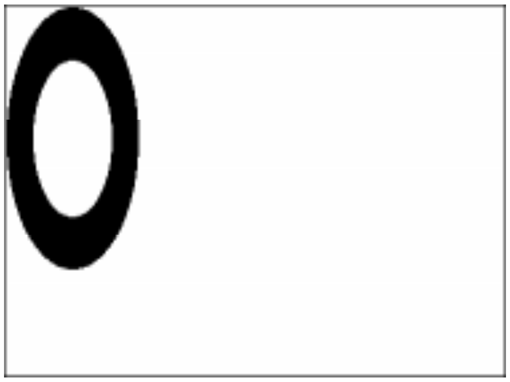
```
circle *+3 ` The circle becomes 3 times wider
```



```
circle *+ 0.25 ` The circle's width becomes a quarter of what it was
```

• Vertical scaling

```
circle */ 2 ` The circle becomes twice taller
```



```
circle */ 0.25 ` The circle's height becomes a quarter of what it was
```

See Also

Bitwise Operators, Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, String Operators, Time Operators.

Bitwise Operators

The bitwise operators operates on Long Integer expressions or values.

Note: If you pass an Integer or a Real value to a bitwise operator, 4th Dimension evaluates the value as a Long Integer value before calculating the expression that uses the bitwise operator.

While using the bitwise operators, you must think about a Long Integer value as an array of 32 bits. The bits are numbered from 0 to 31, from right to left.

Because each bit can equal 0 or 1, you can also think about a Long Integer value as a value where you can store 32 Boolean values. A bit equal to 1 means True and a bit equal to 0 means False.

An expression that uses a bitwise operator returns a Long Integer value, except for the Bit Test operator, where the expression returns a Boolean value. The following table lists the bitwise operators and their syntax:

Operation	Operator	Syntax	Returns	
Bitwise AND	&	Long & Long	Long	
Bitwise OR (inclusive)	|	Long | Long	Long	
Bitwise OR (exclusive)	^|	Long ^| Long	Long	
Left Bit Shift	<<	Long << Long	Long	(see note 1)
Right Bit Shift	>>	Long >> Long	Long	(see note 1)
Bit Set	?+	Long ?+ Long	Long	(see note 2)
Bit Clear	?-	Long ?- Long	Long	(see note 2)
Bit Test	??	Long ?? Long	Boolean	(see note 2)

Notes
(1) For the Left Bit Shift and Right Bit Shift operations, the second operand indicates the number of positions by which the bits of the first operand will be shifted in the resulting value. Therefore, this second operand should be between 0 and 32. Note however, that shifting by 0 returns an unchanged value and shifting by more than 31 bits returns 0x00000000 because all the bits are lost. If you pass another value as second operand, the result is non significant.
(2) For the Bit Set, Bit Clear and Bit Test operations , the second operand indicates the number of the bit on which to act. Therefore, this second operand must be between 0 and 31. Otherwise, the expression returns the value of the first operand unchanged for Bit Set and Bit Clear, and returns False for Bit Test.

The following table lists the bitwise operators and their effects:

Operation	**Description**
Bitwise AND operands.	Each resulting bit is the logical AND of the bits in the two Here is the logical AND table: 1 & 1 → 1 0 & 1 → 0 1 & 0 → 0 0 & 0 → 0 In other words, the resulting bit is 1 if the two operand bits are 1; otherwise the resulting bit is 0.
Bitwise OR (inclusive) operands.	Each resulting bit is the logical OR of the bits in the two Here is the logical OR table: 1 | 1 → 1 0 | 1 → 1 1 | 0 → 1 0 | 0 → 0 In other words, the resulting bit is 1 if at least one of the two operand bits is 1; otherwise the resulting bit is 0.
Bitwise OR (exclusive)	Each resulting bit is the logical XOR of the bits in the two operands. Here is the logical XOR table: 1 ^| 1 → 0 0 ^| 1 → 1 1 ^| 0 → 1 0 ^| 0 → 0 In other words, the resulting bit is 1 if only one of the two operand bits is 1; otherwise the resulting bit is 0.
Left Bit Shift	The resulting value is set to the first operand value, then the resulting bits are shifted to the left by the number of positions indicated by the second operand. The bits on the left are lost and the new bits on the right are set to 0. Note: Taking into account only positive values, shifting to the left by N bits is the same as multiplying by 2^N.
Right Bit Shift	The resulting value is set to the first operand value, then the resulting bits are shifted to the right by the number of position indicated by the second operand. The bits on the right are lost and the new bits on the left are set to 0. Note: Taking into account only positive values, shifting to the right by N bits is the same as dividing by 2^N.

Bit Set	The resulting value is set to the first operand value, then the resulting bit, whose number is indicated by the second operand, is set to 1. The other bits are left unchanged.
Bit Clear	The resulting value is set to the first operand value, then the resulting bit, whose number is indicated by the second operand, is set to 0. The other bits are left unchanged.
Bit Test	Returns True if, in the first operand, the bit whose number is indicated by the second operand is equal to 1. Returns False if, in the first operand, the bit whose number is indicated by the second operand is equal to 0.

Examples

(1) The following table gives an example of each bit operator:

Operation	Example	Result
Bitwise AND	0x0000FFFF & 0xFF00FF00	0x0000FF00
Bitwise OR (inclusive)	0x0000FFFF | 0xFF00FF00	0xFF00FFFF
Bitwise OR (exclusive)	0x0000FFFF & 0xFF00FF00	0xFF0000FF
Left Bit Shift	0x0000FFFF << 8	0x000FFFF0
Right Bit Shift	0x0000FFFF >> 8	0x00000FFF
Bit Set	0x00000000 ?+ 16	0x00010000
Bit Clear	0x00010000 ?- 16	0x00000000
Bit Test	0x00010000 ?? 16	True

(2) 4th Dimension provides many predefined constants. The literals of some of these constants end with “bit” or “mask.” For example, this is the case of the constants provided in the
Resources properties theme:

Constant	Type	Value
System heap resource mask	Long Integer	64
System heap resource bit	Long Integer	6
Purgeable resource mask	Long Integer	32
Purgeable resource bit	Long Integer	5
Locked resource mask	Long Integer	16
Locked resource bit	Long Integer	4
Protected resource mask	Long Integer	8
Protected resource bit	Long Integer	3
Preloaded resource mask	Long Integer	4
Preloaded resource bit	Long Integer	2
Changed resource mask	Long Integer	2
Changed resource bit	Long Integer	1

These constants enable you to test the value returned by Get resource properties or to create the value passed to SET RESOURCE PROPERTIES. Constants whose literal ends with “bit” give the position of the bit you want to test, clear, or set. Constants whose literal ends with “mask” gives a long integer value where only the bit (that you want to test, clear, or set) is equal to one.

For example, to test whether a resource (whose properties have been obtained in the variable $vlResAttr) is purgeable or not, you can write:

```
	If ($vlResAttr ?? Purgeable resource bit) ` Is the resource purgeable?
```
or:
```
	If (($vlResAttr & Purgeable resource mask) # 0) Is the resource purgeable?
```

Conversely, you can use these constants to set the same bit. You can write:

```
	$vlResAttr:=$vlResAttr ?+ Purgeable resource bit
```
or:
```
	$vlResAttr:=$vlResAttr | Purgeable resource bit
```

(3) This example stores two Integer values into a Long Integer value. You can write:

```
	$vlLong:=($viIntA<<16) | $viIntB ` Store two Integers in a Long Integer

	$vlIntA:=$vlLong>>16 ` Extract back the integer stored in the high-word

	$viIntB:=$vlLong & 0xFFFF ` Extract back the Integer stored in the low-word
```

Tip: Be careful when manipulating Long Integer or Integer values with expressions that combine numeric and bitwise operators. The high bit (bit 31 for Long Integer, bit 15 for Integer) sets the sign of the value—positive if it is cleared, negative if it is set. Numeric operators use this bit for detecting the sign of a value, bitwise operators do not care about the meaning of this bit.

See Also

Comparison Operators, Date Operators, Logical Operators, Numeric Operators, Operators, Picture Operators, String Operators, Time Operators.

31 Printing

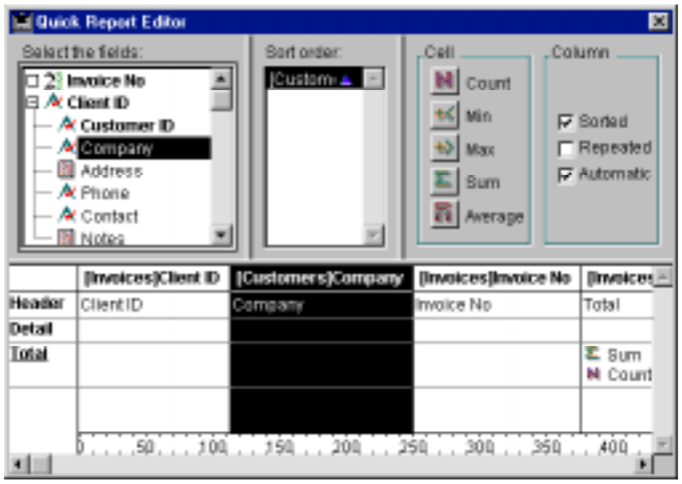
REPORT

REPORT ({table; }document{; *})

Parameter	Type		Description
table	Table	→	Table to print, or Default table, if omitted
document	String	→	Quick Report document
*		→	Suppress the printing dialog boxes

Description

REPORT prints a report for table, created with the Quick Report editor shown here.



Column: Sorted [✓], Repeated [ ], Automatic [✓]

The document parameter is a report document that was created with the Quick Report editor and saved on disk. You save a report document by choosing Save or Save As from the File menu in the Quick Report editor. The document stores the specifications of the report, not the records to be printed.

If an empty string ("") is specified for document, REPORT displays an Open File dialog box and the user can select the report to print. After a report is selected, the dialog boxes for printing are displayed, unless the * parameter is specified. If this parameter is specified, these dialog boxes are not displayed. The report is then printed.

If the document parameter specifies a document that does not exist (for example, pass Char(1) in document), the Quick Report editor is displayed.

The Quick Report editor allows users to create their own reports. When the Quick Report editor is displayed, the menu bar displays the same four menus that manage the editor in the User environment: File, Edit, Font and Style. The user has complete control over the editor. See the *4th Dimension User Reference* for details on creating reports with the Quick Report editor.

If the Quick Report editor is not involved, the OK variable is set to 1 if a report is printed; it is set to 0 (zero) if not (i.e., if the user clicked Cancel in the printing dialog boxes).

Examples

1. The following example lets the user query the [People] table, and then automatically prints the report “Detailed Listing”:

```
      QUERY ([People])
      If (OK=1)
⇒        REPORT ([People];"Detailed Listing";*)
      End if
```

2. The following example lets the user query the [People] table, and then lets the user choose which report to print:

```
      QUERY ([People])
      If (OK=1)
⇒        REPORT ([People];"")
      End if
```

3. The following example lets the user query the [People] table, and then displays the Report editor so the user can design, save, load and print any reports:

```
      QUERY ([People])
      If (OK=1)
⇒        REPORT ([People];Char(1))
      End if
```

See Also

PRINT LABEL, PRINT SELECTION.

PRINT LABEL

PRINT LABEL ({table}{; document}{; *})

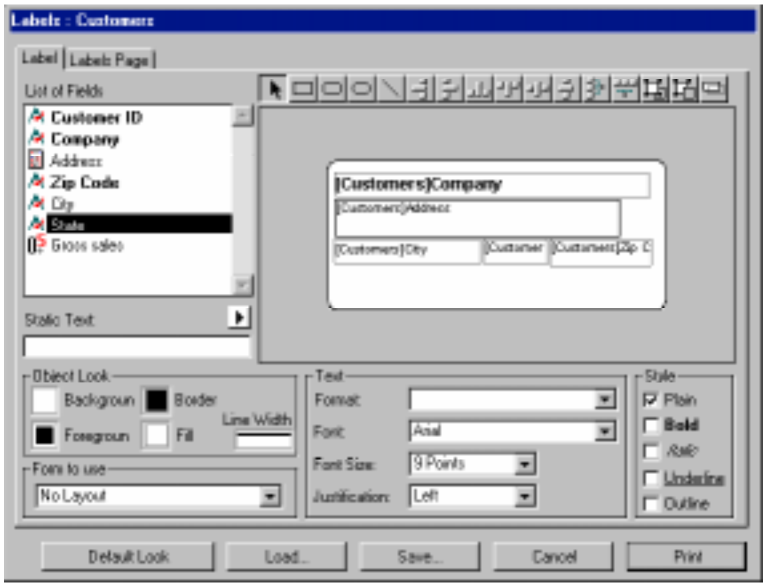
Parameter	Type		Description
table	Table	→	Table to print, or Default table, if omitted
document	String	→	Name of disk label document
*		→	Suppress the printing dialog boxes

Description

PRINT LABEL enables you to print labels with the data from the selection of table.

If do not specify the document parameter, PRINT LABEL prints the current selection of table as labels, using the current output form. You cannot use this command to print subforms. For details about creating forms for labels, refer to the *4th Dimension Design Reference* manual.

If you specify the document parameter, PRINT LABEL enables you to access the Label Wizard (shown below) or to print an existing Label document stored on disk. See the following discussion.


Labels : Customers

List of Fields: Customer ID, Company, Address, Zip Code, City, State, Gross sales

[Customers]Company
[Customers]Address
[Customers]City [Customer [Customers]Zip C

Form to use: No Layout

Font: Arial
Font Size: 9 Points
Justification: Left

Style: [✓ Plain] Bold Italic Underline Outline

In both cases, to suppress the printing dialog boxes, pass the optional * parameter. Note that this parameter has no effect if the Label Wizard is involved.

If the Label Wizard is not involved, the OK variable is set to 1 if all labels are printed; otherwise, it is set to 0 (zero) (i.e., if user clicked Cancel in the printing dialog boxes).

If you specify the document parameter, the labels are printed with the label setup defined in document. If document is an empty string (""), PRINT LABEL will present an Open File dialog box so the user can specify the file to use for the label setup. If document is the name of a document that does not exist (for example, pass char(1) in document), the Label Wizard is displayed and the user can define the label setup.

Examples

1. The following example prints labels using the output form of a table. The example uses two methods. The first is a project method that sets the correct output form and then prints labels:

```
     ALL RECORDS([Addresses])  ` Select all records
     OUTPUT FORM ([Addresses]; "Label Out")  ` Select the output form
⇒    PRINT LABEL([Addresses])  ` Print the labels
     OUTPUT FORM ([Addresses];"Output")  ` Restore default output form
```

The second method is the form method for the form "Label Out". The form contains one variable named vLabel, which is used to hold the concatenated fields. If the second address field (Addr2) is blank, it is removed by the method. Note that this task is performed automatically with the Label Wizard. The form method creates the label for each record:

```
        ` [Addresses]; "Label Out" form method
     Case of
        : (Form event=On Printing Detail)
           vLabel:=[Addresses]Name1+" "+[Addresses]Name2+Char(13)+
                                                   [Addresses]Addr1+Char(13)
           If ([Addresses]Addr2 # "")
              vLabel:=vLabel +[Addresses]Addr2+Char(13)
           End if
           vLabel:=vLabel+[Addresses]City+", "+[Addresses]State+" "+[Addresses]ZipCode
     End case
```

2. The following example lets the user query the [People] table, and then automatically prints the labels “My Labels”:

```
     QUERY ([People])
     If (OK=1)
⇒       PRINT LABEL ([People];"My Labels";*)
     End if
```

3. The following example lets the user query the [People] table, and then lets the user choose the labels to be printed:

```
        QUERY ([People])
        If (OK=1)
⇒           PRINT LABEL ([People];"")
        End if
```

4. The following example lets the user query the [People] table, and then displays the Label Wizard so the user can design, save, load and print any labels:

```
        QUERY ([People])
        If (OK=1)
⇒           PRINT LABLE ([People];Char(1))
        End if
```

See Also

PRINT SELECTION, REPORT.

PRINT SELECTION

version 3

PRINT SELECTION ({table}{; }{*})

Parameter	Type		Description
table	Table	→	Table for which to print the selection, or Default table, if omitted
*		→	If specified, suppress the printing dialog boxes

Description

PRINT SELECTION prints the current selection of table. The records are printed with the current output form of the table in the current process. PRINT SELECTION performs the same action as the Print menu command in the User environment. If the selection is empty, PRINT SELECTION does nothing.

By default, PRINT SELECTION displays the printer dialog boxes before printing. You can suppress these dialog boxes by using the optional * parameter. If the user cancels either of the printer dialog boxes, the command is canceled and the report is not printed. Using the optional * causes the report to be printed with the page setup that was in effect when the form was created, or with the page setup set by the PAGE SETUP command.

During printing, the output form method and/or the form's object methods are executed depending on the events that are enabled in the Form and Object Properties windows in the Design environment, as well as on the events actually occurring:

• An On Printing Header event is generated just before a header area is printed.
• An On Printing Detail event is generated just before a record is printed.
• An On Printing Break event is generated just before a break area is printed.
• An On Printing Footer event is generated just before a footer is printed.

You can check whether PRINT SELECTION is printing the first header by testing Before selection during an On Printing Header event. You can also check for the last footer, by testing End selection during an On Printing Footer event. For more information, see the description of these commands, as well as those of Form event and Level.

To print a sorted selection with subtotals or breaks using PRINT SELECTION, you must first sort the selection. Then, in each Break area of the report, include a variable with an object method that assigns the subtotal to the variable. You can also use statistical and arithmetic functions like Sum and Average to assign values to variables. For more information, see the descriptions of Subtotal, BREAL LEVEL and ACCUMULATE.

Warning: Do not use the PAGE BREAK command with the PRINT SELECTION command. PAGE BREAK is to be used with the PRINT FORM command.

After a call to PRINT SELECTION, the OK variable is set to 1 if the printing has been completed. If the printing was interrupted, the OK variable is set to 0 (zero) (i.e., the user clicked Cancel in the printing dialog boxes).

Example

The following example selects all the records in the [People] table. It then uses the DISPLAY SELECTION command to display the records and allows the user to highlight the records to print. Finally, it uses the selected records with the USE SET command, and prints them with PRINT SELECTION:

```
       ALL RECORDS([People])  ` Select all records
       DISPLAY SELECTION ([People]; *)  ` Display the records
       USE SET ("UserSet")  ` Use only records picked by user
⇒      PRINT SELECTION([People])  ` Print the records that the user picked
```

See Also

ACCUMULATE, BREAK LEVEL, Level, PAGE SETUP, Subtotal.

Printing page

Printing page → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Page number of page currently being printed

Description

Printing page returns the printing page number. It can be used only when you are printing with PRINT SELECTION or the Print menu in the User environment.

Example

The following example changes the position of the page numbers on a report so that the report can be reproduced in a double-sided format. The form for the report has two variables that display page numbers. A variable in the lower-left corner (vLeftPageNum) will print the even page numbers. A variable in the lower-right corner (vRightPageNum) will print the odd page numbers. The example tests for even pages, then clears and sets the appropriate variables:

```
        Case of
            : (Form event=On Printing Footer)
⇒               If ((Printing page % 2) = 0)   ` Modulo is 0, it is an even page
⇒                   vLeftPageNum:=String(Printing page)   ` Set the left page number
                    vRightPageNum:=""   ` Clear the right page number
                Else   ` Otherwise it is an odd page
                    vLeftPageNum:="" ` Clear the left page number
⇒                   vRightPageNum:=String (Printing page)   ` Set the right page number
                End if
        End case
```

See Also

PRINT SELECTION.

BREAK LEVEL

version 3

BREAK LEVEL (level{; pageBreak})

Parameter	Type		Description
level	Number	→	Number of break levels
pageBreak	Number	→	Break level for which to do a page break

Description

BREAK LEVEL specifies the number of break levels in a report performed using PRINT SELECTION.

Warning: In compiled mode, you **must** execute BREAK LEVEL and ACCUMULATE before every report for which you want to do break processing. These commands activate break processing for a report. See the explanation for the Subtotal command.

The level parameter indicates the deepest level for which you want to perform break processing. You must have sorted the records with at least that many levels. If you have sorted more levels, those levels will be printed as sorted, but will not be processed for breaks.

Each break level that is generated will print the corresponding Break areas and Header areas in the form. There should be at least as many Break areas in the form as the number you pass in level. If there are more Break areas, they will be ignored and will not be printed.

The second, optional, argument, pageBreak, is used to cause page breaks during printing.

Example

The following example prints a report with two break levels. The selection is sorted on four levels, but the BREAK LEVEL command specifies to break on only two levels. One field is accumulated with the ACCUMULATE command:

```
   ORDER BY ([Emp]Dept;>;[Emp]Title;>;[Emp]Last;>;[Emp]First;>)   ` Sort on four levels
⇒  BREAK LEVEL (2)   ` Turn on break processing to 2 levels (Dept and Title)
   ACCUMULATE ([Emp]Salary)  ` Accumulate the salaries
   OUTPUT FORM ([Emp];"Dept salary")  ` Select the report form
   PRINT SELECTION([Emp])  ` Print the report
```

See Also

ACCUMULATE, ORDER BY, PRINT SELECTION, Subtotal.

ACCUMULATE

ACCUMULATE (data{; data2; ...; dataN})

Parameter	Type		Description
data	Field or variable	→	Numeric field or variable on which to accumulate

Description

ACCUMULATE specifies the fields or variables to be accumulated during a form report performed using PRINT SELECTION.

Warning: In compiled mode, you **must** execute BREAK LEVEL and ACCUMULATE before every report for which you want to do break processing. These commands activate break processing for a report. See the explanation for the Subtotal command.

Use ACCUMULATE when you want to include subtotals for numeric fields or variables in a form report. ACCUMULATE tells 4th Dimension to store subtotals for each of the Data arguments. The subtotals are accumulated for each break level specified with the BREAK LEVEL command.

Execute ACCUMULATE before printing the report with PRINT SELECTION.

Use the Subtotal function in the form method or an object method to return the subtotal of one of the data arguments.

Example

See the example for the BREAK LEVEL command.

See Also

BREAK LEVEL, ORDER BY, PRINT SELECTION, Subtotal.

Subtotal

Subtotal (data{; pageBreak}) → Number

Parameter	Type		Description
data	Field	→	Numeric field or variable to return subtotal
pageBreak	Number	→	Break level for which to cause a page break
Function result	Number	←	Subtotal of data

Description

Subtotal returns the subtotal for data for the current or last break level. Subtotal works only when a sorted selection is being printed with PRINT SELECTION or when printing using Print in the User environment. The data parameter must be of type real, integer, or long integer. Assign the result of the Subtotal function to a variable placed in the Break area of the form.

Warning: In compiled mode, you **must** execute BREAK LEVEL and ACCUMULATE before every form report for which you want to do break processing and calculate subtotals. See discussion at the end of the description of this command.

Subtotal should be in the form method or an object method for the form. 4th Dimension scans the form method and object methods before printing; if Subtotal is present, break processing will be initiated (in interpreted mode only).

The second, optional, argument to Subtotal is used to cause page breaks during printing. If pageBreak is 0, Subtotal does not issue a page break. If pageBreak equals 1, Subtotal issues a page break for each level 1 break. If pageBreak equals 2, Subtotal issues a page break for each level 1 and level 2 break, and so on.

To have breaks on N sort levels, you must sort the current selection on N + 1 levels (unless you use BREAK LEVEL or ACCUMULATE, in which case N levels is sufficien). This lets you sort on a last field, so that the field does not create unwanted breaks. To have the last sort field generate a break, sort the field twice.

Tip: If you execute Subtotal from within an output form displayed at the screen, an error will be generated, triggering an infinite loop of updates between the form and the error window. To get out of this loop, press Alt+Shift (Windows) or Option-Shift (Macintosh) when you click on the Abort button in the Error window (you may have to do so several times). This temporarily stops the updates for the form's window. Select another form as the output form so the error will occur again. Go back to the Design Environment and isolate the call to Subtotal into a test Form event=On Printing Break if you use the form both for display and printing.

Example

The following example is a one-line object method in a Break area of a form (B0, the area above the B0 marker). The vSalary variable is placed in the Break area. The variable is assigned the subtotal of the Salary field for this break level:

```
        Case of
           : (Form event=On Printing Break)
⇒              vSalary:=Subtotal ([Employees]Salary)
        End case
```

For more information about designing forms with header and break areas, see the *4th Dimension Design Reference* manual.

Activating Break Processing in Form Reports

Break processing in form reports can be activated in two ways:
• The first uses the Subtotal function.
• The second uses the BREAK LEVEL and ACCUMULATE commands.

Both methods can achieve the same results, but have different advantages.

Using Subtotal For Break Processing (Interpreted Mode Only)

To turn on break processing with the Subtotal function, the function must appear in the form method or an object method for a variable located in a Break area of the form. Before printing the report, 4th Dimension scans the form and object methods for the Subtotal function.

If 4th Dimension finds the function, break processing is activated. The Subtotal function does not need to be executed for it to turn on break processing. For example, it could be in a method of an object that is below the Footer line and therefore would never be printed or executed.

When Subtotal is used to activate break processing, you must sort on one more level than you break on. For example, to have two levels of breaks in your report, sort on three levels.

Using BREAK LEVEL and ACCUMULATE for Break Processing

You can also use the BREAK LEVEL and ACCUMULATE commands to turn on break processing. To do so, you must execute both of these commands before printing a form report. In this scheme, the Subtotal function is still required in order to display values on a form. You do not need to sort on one extra level; you must, of course, sort on at least as many levels as you need to break on.

Comparing the Two Methods

The primary advantage of using Subtotal to initiate break processing is that you do not need to execute a method prior to printing the report. This is especially useful in the User environment.

The process to print the report in the User environment is typically like this:
1. Select the records to be printed.
2. Order by (sort) the records, sorting on one extra level.
3. Choose Print from the File menu.

4th Dimension scans the form and object methods, finds the Subtotal function, turns on break processing, and prints the report. There are two disadvantages to using Subtotal to trigger break processing:
• You cannot use Subtotal to activate break processing in compiled databases.
• You must sort on one extra level; if you have many records, this may be time consuming.

Using BREAK LEVEL and ACCUMULATE to activate break processing is the recommended method when using methods to generate form reports. The process to print a report using this method is typically like this:
1. Select the records to be printed.
2. Sort the records using ORDER BY. Sort on at least the same number of levels as breaks.
3. Execute BREAK LEVEL and ACCUMULATE.
4. Print the report using PRINT SELECTION.

You must use BREAK LEVEL and ACCUMULATE to activate break processing in compiled mode. However, the Subtotal function is still necessary in order to display values on a form.

See Also

ACCUMULATE, BREAK LEVEL, Level, PRINT SELECTION.

Level

Level → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Current break or header level

Description

Level is used to determine the current header or break level. It returns the level number during the On Printing Header and On Printing Break events.

Level 0 is the last level to be printed and is appropriate for printing a grand total. Level returns 1 when 4th Dimension prints a break on the first sorted field, 2 when 4th Dimension prints a break on the second sorted field, and so on.

Example

This example is a template for a form method. It shows each of the possible events that can occur while a summary report uses a form as an output form. Level is called when a header or a break is printed:

```
           ` Method of a form being used as output form for a summary report
        $vpFormTable:=Current form table
        Case of
                ` ...
            : (Form event=On Printing Header)
                    ` A header area is about to be printed
                Case of
                    : (Before selection($vpFormTable->))
                        ` Code for the first break header goes here
⇒                   : (Level = 1)
                        ` Code for a break header level 1 goes here
⇒                   : (Level = 2)
                        ` Code for a break header level 2 goes here
                        ` ...
                End case
            : (Form event=On Printing Details)
                    ` A record is about to be printed
                    ` Code for each record goes here
```

```
      : (Form event=On Printing Break)
            ` A break area is about to be printed
         Case of
⇒           : (Level = 0)
               ` Code for a break level 0 goes here
⇒           : (Level = 1)
               ` Code for a break level 1 goes here
               ` ...
         End case
      : (Form event=On Printing Footer)
         If(End selection($vpFormTable->))
            ` Code for the last footer goes here
         Else
            ` Code for a footer goes here
         End if
   End case
```

See Also

ACCUMULATE, BREAK LEVEL, Form event, PRINT SELECTION.

PRINT RECORD

PRINT RECORD ({table}{; }{*})

Parameter	Type		Description
table	Table	→	Table for which to print the current record or Default table if omitted
*		→	Suppress the printer dialog boxes

Description

PRINT RECORD prints the current record of table, without modifying the current selection. The current output form is used for printing. If there is no current record for table, PRINT RECORD does nothing.

You can print subforms and external objects with the PRINT RECORD command. This is not possible with PRINT FORM.

Note: If there are modifications to the record that have not been saved, this command prints the modified field values, not the field values located on disk.

If you pass the optional * parameter, the printing dialog boxes are not displayed. In this case, the record is printed with the default print settings, unless you call PAGE SETUP before calling PRINT RECORD.

Example

The following example prints the current record of the [Invoices] table. The code is contained in the object method of a Print button on the input form. When the user clicks the button, the record is printed using an output form designed for this purpose.

```
	 ` Select the right output form for printing
	OUTPUT FORM([Invoices];"Print One From Data Entry")
	 ` Print the Invoices as it is (without showing the printing dialog boxes)
⇒	PRINT RECORD([Invoices];*)
	 ` Restore the previous output form
	OUTPUT FORM([Invoices];"Standard Output")
```

See Also

PRINT FORM.

PAGE SETUP

PAGE SETUP ({table; }form)

Parameter	Type		Description
table	Table	→	Table owning form, or Default table, if omitted
form	String	→	Form to use for page setup

Description

PAGE SETUP sets the page setup for the printer to that stored with form. The page setup is stored with the form when the form is saved in the Design environment.

In the following three cases, the printing dialog boxes are not displayed and the printing is performed with the default print settings. :
• Calling PRINT SELECTION to which you pass the optional * parameter
• Calling PRINT RECORD to which you pass the optional * parameter
• Issuing a series of calls to PRINT FORM not preceeded by a call to PRINT SETTINGS.

Calling PAGE SETUP enables you to skip the printing dialog boxes AND to use print settings other than the default ones.

Example

Several (empty) forms are created for a table named [Design Stuff]. The form “PS100” is assigned a page setup with a scaling of 100%, the form “PS90” is assigned a page setup with a scaling of 90%, and so on. The following project method enables you to print the selection of a table using various scalings without having to specify the scaling in the printing dialog boxes (which are not displayed), each time:

```
     ` AUTOMATIC SCALED PRINTING project method
     ` AUTOMATIC SCALED PRINTING ( Pointer ; String {; Long } )
     ` AUTOMATIC SCALED PRINTING ( ->[Table]; "Output form" {; Scaling } )
  If (Count parameters>=3)
⇒    PAGE SETUP([Design Stuff];"PS"+String($3))
     If (Count parameters>=2)
        OUTPUT FORM($1->;$2)
     End if
  End if
  If (Count parameters>=1)
     PRINT SELECTION($1->;*)
  Else
     PRINT SELECTION(*)
  End if
```

Once this project method is written, you call it in this way:

```
    ` Look for current invoices
QUERY ([Invoices];[Invoices]Paid=False)
    ` Print Summary Report in 90% reduction
AUTOMATIC SCALED PRINTING (->[Invoices];"Summary Report";90)
    ` Print Detailed Report in 50% reduction
AUTOMATIC SCALED PRINTING (->[Invoices];"Detailed Report";50)
```

See Also

PRINT FORM, PRINT RECORD, PRINT SELECTION.

PRINT SETTINGS

version 3

PRINT SETTINGS

Parameter	Type	Description
This command does not require any parameters		

Description

PRINT SETTINGS displays the printing dialog boxes. First, it displays the Print Setup dialog box. Then, it displays the Print Job dialog box.

You should include PRINT SETTINGS before any group of PRINT FORM commands. On the other hand, PRINT SETTINGS has no effect on printing performed with other commands.

The Print Job dialog box contains a Preview on Screen check box that allows the user to specify to print to the screen. You can preset or reset this check bok by calling SET PRINT PREVIEW before calling PRINT SETTINGS.

Example

See example for the command PRINT FORM.

System Variables or Sets

If the user clicks OK in both dialog boxes, the OK system variable is set to 1. Otherwise, the OK system variable is set to 0.

See Also

PAGE BREAK, PRINT FORM, SET PRINT PREVIEW.

SET PRINT PREVIEW

SET PRINT PREVIEW (preview)

Parameter	Type		Description
preview	Boolean	→	Preview on screen (TRUE), or No preview (FALSE)

Description

SET PRINT PREVIEW allows you to programmatically check or uncheck the Preview on Screen option of the Print dialog box. If you pass TRUE in preview, Preview on Screen will be checked, if you pass FALSE in preview , Preview on Screen will be unchecked. This setting is local to a process and does not affect the printing of other processes or users.

Example

The following example turns on the Preview on Screen option to display the results of a query on screen, and then turns it off.

```
     QUERY([Customers])
     If (OK=1)
⇒       SET PRINT PREVIEW (True)
        PRINT SELECTION ([Customers] ; *)
⇒       SET PRINT PREVIEW (False)
     End if
```

See Also

PRINT RECORD, PRINT SELECTION, PRINT SETTINGS.

PRINT FORM

Printing

version 3

PRINT FORM ({table; }form)

Parameter	Type		Description
table	Table	→	Table owning the form, or Default table, if omitted
form	String	→	Form to print

Description

PRINT FORM simply prints form with the current values of fields and variables. It prints only the Detail area (the area between the Header line and the Detail line) of the form. It is usually used to print very complex reports that require complete control over the printing process. PRINT FORM does not do any record processing, break processing, page breaks, headers, or footers. These operations are your responsibility. PRINT FORM prints fields and variables in a fixed size frame only.

Since PRINT FORM does not issue a page break after printing the form, it is easy to combine different forms on the same page. Thus, PRINT FORM is perfect for complex printing tasks that involve different tables and different forms. To force a page break between forms, use the PAGE BREAK command.

The printer dialog boxes do not appear when you use PRINT FORM. The report does not use the print settings that were assigned to the form in the Design environment. There are two ways to specify the print settings before issuing a series of calls to PRINT FORM:
• Call PRINT SETTINGS. In this case, you let the user choose the settings.
• Call PAGE SETUP. In this case, print settings are specified programmatically.

PRINT FORM builds each printed page in memory. Each page is printed when the page in memory is full or when you call PAGE BREAK. To ensure the printing of the last page after any use of PRINT FORM, you must conclude with the PAGE BREAK command. Otherwise, if the last page is not full, it stays in memory and is not printed.

Warning: Subforms and external objects are not printed with PRINT FORM. To print only one form with such objects, use PRINT RECORD instead.

PRINT FORM generates only one On Printing Detail event for the form method.

Example

The following example performs as a PRINT SELECTION command would. However, the report uses one of two different forms, depending on whether the record is for a check or a deposit:

```
        QUERY([Register])  ` Select the records
        If (OK=1)
            ORDER BY([Register])  ` Sort the records
            If (OK=1)
                PRINT SETTINGS ` Display Printing dialog boxes
                If (OK=1)
                    For ($vlRecord; 1; Records in selection([Register]))
                        If ([Register]Type = "Check")
                                ` Use one form for checks
⇒                           PRINT FORM ([Register]; "Check Out")
                        Else
                                ` Use another form for deposits
⇒                           PRINT FORM ([Register]; "Deposit Out")
                        End if
                        NEXT RECORD([Register])
                    End for
                    PAGE BREAK ` Make sure the last page is printed
                End if
            End if
        End if
```

See Also

PAGE BREAK, PAGE SETUP, PRINT SETTINGS.

PAGE BREAK

PAGE BREAK {(* | >)}

Parameter	Type		Description
* | >		→	* Cancel printing job started with PRINT FORM, or > Force one printing job

Description

PAGE BREAK triggers the printing of the data that has been sent to the printer, and ejects the page. PAGE BREAK is used with PRINT FORM to force page breaks and to print the last page. Do not use PAGE BREAK with the PRINT SELECTION command. Instead, use Subtotal or BREAK LEVEL with the optional parameter to generate page breaks.

The * and > parameters are both optional.

The * parameter allows you to cancel a print job started with the PRINT FORM command. Executing this command immediately stops the print job in progress.

The > parameter modifies the way in which the PAGE BREAK command behaves. This syntax has two effects:
• It holds the print job open until the PAGE BREAK command is executed again without a parameter.
• It gives priority to the print job. No other printing can take place until the print job is finished.
The second option is particularly useful when used with a spooled print job. The > parameter guarantees that the print job will be spooled to one file. This will reduce printing time.

Example

See example for the PRINT FORM command.

See Also

PRINT FORM.

32 Pictures

Pictures

Supported Formats

The following charts summarize the support for various picture formats on the Macintosh and Windows platforms.

Cut and Paste: Supported formats

	PICT	EMF	WMF	BITMAP
Macintosh	Yes	-	-	-
Windows	Yes	Yes embedded in PicComment	Yes embedded in PicComment	Yes converted to Macintosh PICT

Display: Supported formats

	PICT	QuickTime	embedded WMF	embedded EMF
Macintosh	Yes	Yes	No	No
Windows	Yes	Yes NT & WIN 95 + QT 32 bit	Yes	Yes

ACI_Pack ReadPictureFile supported formats (see Note)

	PICT	BMP	WMF	EMF	JPEG
Macintosh	Yes	Yes converted to Mac PICT	No	No	Yes expanded to Mac PICT
Windows	Yes	Yes converted to Mac PICT	Yes embedded in PicComment	Yes embedded in PicComment	Yes expanded to Mac PICT

Note: ACI Pack is a 4D Plug-in from ACI, delivered with 4D.

Apple QuickTime Compression

Apple uses QuickTime to implement new compression technologies, such as JPEG. Apple has added new opcodes to the original PICT specifications, so Macintosh applications can handle QuickTime pictures without modification. When the application asks the system to draw a picture containing embedded QuickTime data, the bitmap is expanded and displayed if QuickTime is present; the QuickTime opcode is ignored if QuickTime is not installed. This technology is transparent to the user and takes a minimal amount of memory, because a 1 megabyte picture can be stored in a 40 kilobyte PICT, and need not be expanded before it is displayed.

QuickTime Compressor Types

Following is a list of the compressor types available in QuickTime:
• **Photo compressor** (‘jpeg’): Uses the Joint Photographic Experts Group (JPEG) algorithm for picture compression. JPEG is an international standard for compressing still pictures.
• **Video compressor** (‘rpza’): Allows very fast decompression while maintaining good picture quality.
• **Animation compressor** (‘rle’): Used with animation and computer-generated video content.
• **Raw compressor** (‘raw’): Reduces picture storage requirements by converting a picture’s pixel depth.
• **Graphics compressor** (‘smc’): An alternative to animation, but has poorer performance quality.
• **Compact video compressor** (‘cdvc’): Similar to video compressor, but obtains higher compression ratios, better picture quality, and faster playback.

When specifying the compressor type, be sure to include a space in the method parameter if indicated. If method is an empty string, the picture is loaded but not compressed.

Picture Compression Errors

When you try to use a picture compression command and QuickTime is not installed in your system, 4th Dimension returns the error code -9955. Other errors generated by QuickTime can also be returned. You can catch these errors using an error-handling method installed with ON ERR CALL.

Using Apple QuickTime on Windows

• Since Altura and 4D are both real 32-bit applications, you need to install 32 bit QuickTime for Windows (version 2.1.1 b50 or higher).

• QuickTime for Windows does not support Windows 3.11, so there is no way to display a QuickTime picture in Windows 3.11.

• In order to display the picture, QuickTime for Windows must use a work file on disk. Each time a QuickTime picture is displayed on screen, it is first written on the disk, then deleted. Usually, the temporary file remains in the cache and is not really written on the disk, so the operation is very fast. However, on a slow PC, or on one with insufficient memory, this operation can be slower.

• QuickTime for Windows can only display QuickTime pictures—there is no way to compress pictures. This is why the 4D commands dealing with QuickTime compression do not work on Windows (and will not, until QuickTime for Windows supports compression). The commands that do not work on Windows are: COMPRESS PICTURE, COMPRESS PICTURE FILE, LOAD COMPRESS PICTURE FROM FILE.

SAVE PICTURE TO FILE does not use QuickTime, so it works on Windows the same as it does on Macintosh. It generates Macintosh PICT files, which can be opened on Windows by advanced graphics applications such as Photoshop.

See Also

COMPRESS PICTURE, COMPRESS PICTURE FILE, LOAD COMPRESS PICTURE FROM FILE, PICTURE PROPERTIES, Picture size, SAVE PICTURE TO FILE.

COMPRESS PICTURE

COMPRESS PICTURE (picture; method; quality)

Parameter	Type		Description
picture	Picture	→	Picture to be compressed
		←	Compressed picture
method	String	→	4-character string compression method
quality	Number	→	Compression quality (1..1000)

Description

The command COMPRESS PICTURE compresses the picture contained in the field or variable picture.

The parameter method is a 4-character string indicating the compressor type.

The parameter quality is an integer between 1 and 1000 indicating the quality of the compressed picture. In general, reducing the quality will allow for greater compression of the picture.

Warning: The compression ratio possible for a given quality depends on the size and nature of the picture you are compressing. Compressing small pictures may not produce any decrease in size.

See Also

COMPRESS PICTURE FILE, LOAD COMPRESS PICTURE FROM FILE, Pictures.

LOAD COMPRESS PICTURE FROM FILE

LOAD COMPRESS PICTURE FROM FILE (document; method; quality; picture)

Parameter	Type		Description
document	DocRef	→	Document reference number
method	String	→	4-character string compression method
quality	Number	→	Compression quality (1..1000)
picture	Picture	←	Compressed picture

Description

This command compresses a picture loaded from a document on disk.

You can open a PICT document using the Open document function. You can then use the document reference returned by this function to load and compress the PICT found in the document. This command loads the picture into memory, compresses it using the method and quality you have specified, and then returns it into Picture.

The picture is loaded into memory before it is compressed. If there is not enough memory to load the picture, use COMPRESS PICTURE FILE before calling LOAD COMPRESS PICTURE FROM FILE.

The parameter method is a 4-character string indicating the compressor type. The parameter quality is an integer between 1 and 1000 indicating the quality of the compressed picture. In general, reducing the quality will allow for greater compression of the picture.

Warning: The compression ratio possible for a given quality depends on the size and nature of the picture you are compressing. Compressing small pictures may not produce any decrease in size.

Example

The following example presents an Open File dialog box that allows you to select a PICT file. The picture in the PICT file is loaded into memory, compressed, and stored in a picture variable. The file is then closed.

```
     vRef:=Open document ("";"PICT")
     If (OK=1)
⇒        LOAD COMPRESS PICTURE FROM FILE(vRef;"jpeg";500;Picture)
         CLOSE DOCUMENT(vRef)
     End if
```

See Also

COMPRESS PICTURE, COMPRESS PICTURE FILE, Pictures, SAVE PICTURE TO FILE.

COMPRESS PICTURE FILE

COMPRESS PICTURE FILE (document; method; quality)

Parameter	Type		Description
document	DocRef	→	Document reference number
method	String	→	4-character string compression method
quality	Number	→	Compression quality (1..1000)

Description

This command compresses a picture document on disk. Use this command to compress a picture that you know cannot be loaded with the available memory. Once compressed, it can be loaded into memory using LOAD COMPRESS PICTURE FROM FILE.

The parameter method is a 4-character string indicating the compressor type.

The parameter quality is an integer between 1 and 1000 indicating the quality of the compressed picture. In general, reducing the quality will allow for greater compression of the picture.

Warning: The compression ratio possible for a given quality depends on the size and nature of the picture you are compressing. Compressing small pictures may not produce any decrease in size.

Example

The following example presents the Open File dialog box that allows you to select a PICT file. Only PICT files will be displayed. The picture is compressed, loaded into memory, and stored in a picture variable. The file is then closed.

```
	vRef:=Open document ("";"PICT")
	If (OK=1)
⇒		COMPRESS PICTURE FILE(vRef;"jpeg";500)
		LOAD COMPRESS PICTURE FROM FILE(vRef;"";500;vPict)
		CLOSE DOCUMENT(vRef)
	End if
```

See Also

COMPRESS PICTURE, LOAD COMPRESS PICTURE FROM FILE, SAVE PICTURE TO FILE.

SAVE PICTURE TO FILE

SAVE PICTURE TO FILE (document; picture)

Parameter	Type		Description
document	DocRef	→	Document reference number
picture	Picture	→	Picture to be saved

Description

This command saves picture in a document that was created using the Create document function.

Example

The following example creates a document and saves a picture in it:

```
        vRef:=Create document("";"PICT")
        If (OK=1)
⇒           SAVE PICTURE TO FILE(vRef;vPict)
            CLOSE DOCUMENT(vRef)
        End if
```

See Also

COMPRESS PICTURE FILE, LOAD COMPRESS PICTURE FROM FILE.

Picture size

Picture size (picture) → Number

Parameter	Type		Description
picture	Picture	→	Picture for which to return the size in bytes
Function result	Number	←	Size in bytes of the picture

Description

This function returns the size of picture in bytes.

See Also

PICTURE PROPERTIES.

PICTURE PROPERTIES

PICTURE PROPERTIES (picture; width; height{; hOffset{; vOffset{; mode}}})

Parameter	Type		Description
picture	Picture	→	Picture for which to get information
width	Number	←	Width of the picture expressed in pixels
height	Number	←	Height of the picture expressed in pixels
hOffset	Number	←	Horizontal offset when displayed on background
vOffset	Number	←	Vertical offset when displayed on background
mode	Number	←	Transfer mode when displayed on background

Description

The command PICTURE PROPERTIES returns information about the picture you pass in picture.

The parameters width and height return the width and height of the picture.

The parameters hOffset, vOffset, and mode return the horizontal and vertical positions and the transfer mode of the picture when displayed on the background in a form.

See Also

Picture size.

PICTURE LIBRARY LIST

version 6.0.2

PICTURE LIBRARY LIST (picRefs; picNames)

Parameter	Type		Description
picRefs	Numeric Array	←	Reference numbers of the Picture Library graphics
picNames	String Array	←	Names of the Picture Library graphics

Description

The command PICTURE LIBRARY LIST returns the reference numbers and the names of the pictures currently stored in the Picture Library of the database.

After the call, you retrieve the reference numbers in the array picRefs and the names in the array picNames. The two arrays are synchronized: the nth element of picRefs is the reference number of the Picture Library graphic whose name is returned in the nth element of picNames.

The array picRefs can be a Real, Long Integer or Integer array. In interpreted mode, if the array is not declared prior to the call to PICTURE LIBRARY LIST, a Long Integer array is created by default.

The array picNames can be a String or Text array. In interpreted mode, if the array is not declared prior to the call PICTURE LIBRARY LIST, a Text array is created by default.

The maximum length of a Picture Library graphic name is 31 characters. If you use a String array as picNames, declare it with a large enough fixed length to avoid having a truncated name returned.

If there are no pictures in the Picture Library, both arrays are returned empty.

To obtain the number of pictures currently stored in the Picture Library, use the Size of array command to get the size of one of the two arrays.

Examples

1. The following code returns the catalog of the Picture Library in the arrays alPicRef and asPicName:

```
⇒	PICTURE LIBRARY LIST(alPicRef;asPicName)
```

2. The following example tests whether or not the Picture Library is empty:

```
PICTURE LIBRARY LIST(alPicRef;asPicName)
If (Size of array(alPicRef)=0)
   ALERT("The Picture Library is empty.")
Else
   ALERT("The Picture Library contains "+String(Size of array(alPicRef))+" pictures.")
End if
```

3. The following example exports the Picture Library to a document on disk:

```
⇒      PICTURE LIBRARY LIST($alPicRef;$asPicName)
       $vlNbPictures:=Size of array($alPicRef)
       If ($vlNbPictures>0)
          SET CHANNEL(12;"")
          If (OK=1)
             $vsTag:="4DV6PICTURELIBRARYEXPORT"
             SEND VARIABLE($vsTag)
             SEND VARIABLE($vlNbPictures)
             gError:=0
             For($vlPicture;1;$vlNbPictures)
                $vlPicRef:=$alPicRef{$vlPicture}
                $vsPicName:=$asPicName{$vlPicture}
⇒               GET PICTURE FROM LIBRARY(alPicRef{$vlPicture};$vgPicture)
                If (OK=1)
                   SEND VARIABLE($vlPicRef)
                   SEND VARIABLE($vsPicName)
                   SEND VARIABLE($vgPicture)
                Else
                   $vlPicture:=$vlNbPictures+1
                   gError:=-108
                End if
             End for
             SET CHANNEL(11)
             If (gError#0)
                ALERT("The Picture Library could not be exported, retry with more memory.")
                DELETE DOCUMENT (Document)
             End if
          End if
       Else
          ALERT("The Picture Library is empty.")
       End if
```

See Also

GET PICTURE FROM LIBRARY, REMOVE PICTURE FROM LIBRARY, SET PICTURE TO LIBRARY.

GET PICTURE FROM LIBRARY

GET PICTURE FROM LIBRARY (picRef; picture)

Parameter	Type		Description
picRef	Number	→	Reference number of Picture Library graphic
picture	Picture Variable	←	Picture from the Picture Library

Description

The GET PICTURE FROM LIBRARY command returns in the picture parameter the Picture Library graphic whose reference number is passed in picRef.

If there is no picture with that reference number, GET PICTURE FROM LIBRARY leaves picture unchanged.

Examples

1. The following example returns in vgMyPicture the picture whose reference number is stored in the local variable $vlPicRef:

```
⇒	GET PICTURE FROM LIBRARY($vlPicRef;vgMyPicture)
```

2. See the third example for the command PICTURE LIBRARY LIST.

See Also

PICTURE LIBRARY LIST, REMOVE PICTURE FROM LIBRARY, SET PICTURE TO LIBRARY.

System Variables and Sets

The OK variable is set to 1 if the Picture Library graphic exists. Otherwise, OK is set to zero.

Error Handling

If there is not enough memory to return the picture, an error -108 is generated. You can catch this error using an error-handling method.

SET PICTURE TO LIBRARY

version 6.0.2

SET PICTURE TO LIBRARY (picture; picRef; picName)

Parameter	Type		Description
picture	Picture	→	New picture
picRef	Number	→	Reference number of Picture Library graphic
picName	String	→	New name of the picture

Description

The command SET PICTURE TO LIBRARY creates a new picture or replaces a picture in the Picture Library.

Before the call, you pass:
• the picture reference number in picRef (range 1..32767)
• the picture itself in picture.
• the name of the picture in picName (maximum length: 31 characters).

If there is an already existing Picture Library graphic with the same reference number, the picture contents are replaced and the picture is renamed according to the values you pass in picture and picName. If there is no Picture Library graphic with the reference number you pass in picRef, a new picture is added to the Picture Library.

4D Server: SET PICTURE TO LIBRARY cannot be used from within a method executed on the server machine (stored procedure or trigger). If you call SET PICTURE TO LIBRARY on a server machine, nothing happens—the call is ignored.

Warning: Design objects (hierarchical list items, menu items and so on) may refer to Picture Library graphics. Use caution when modifying a Picture Library graphic programmatically.

Note: If you pass an empty picture in picture or a negative or null value in picRef, the command does nothing.

Examples

1. No matter what the current contents of the Picture Library, the following example adds a new picture to the Picture Library by first looking for a unique picture reference number:

```
⇒      PICTURE LIBRARY LIST($alPicRef;$asPicNames)
       Repeat
          $vlPicRef:=1+Abs(Random)
       Until (Find in array($alPicRef;$vlPicRef)<0)
⇒      SET PICTURE TO LIBRARY(vgPicture;$vlPicRef;"New Picture")
```

2. The following example imports into the Picture Library the pictures (stored in a document on disk) created by the third example for the command PICTURE LIBRARY LIST:

```
      SET CHANNEL(10;"")
      If (OK=1)
         RECEIVE VARIABLE($vsTag)
         If ($vsTag="4DV6PICTURELIBRARYEXPORT")
            RECEIVE VARIABLE($vlNbPictures)
            If ($vlNbPictures)
               For($vlPicture;1;$vlNbPictures)
                  RECEIVE VARIABLE($vlPicRef)
                  If (OK=1)
                     RECEIVE VARIABLE($vlPicName)
                  End if
                  If (OK=1)
                     RECEIVE VARIABLE ($vgPicture)
                  End if
                  If (OK=1)
⇒                    SET PICTURE TO LIBRARY($vgPicture;$vlPicRef;$vlPicName)
                  Else
                     $vlPicture:=$vlNbPictures+1
                     ALERT("This file looks like being damaged.")
                  End if
               End for
            Else
               ALERT("This file looks like being damaged.")
            End if
         Else
            ALERT("The file “"+Document+"” is not a Picture Library export file.")
         End if
         SET CHANNEL(11)
      End
```

See Also

GET PICTURE FROM LIBRARY, PICTURE LIBRARY LIST, REMOVE PICTURE FROM LIBRARY.

System Variables and Sets

None is affected.

Error Handling

If there is not enough memory to add the picture to the Picture Library, an error -108 is generated. Note that I/O errors may also be returned (i.e., the structure file is locked). You can catch these errors using an error-handling method.

REMOVE PICTURE FROM LIBRARY

version 6.0.2

REMOVE PICTURE FROM LIBRARY (picRef)

Parameter	Type		Description
npicRef	Number	→	Reference number of Picture Library graphic

Description

The command REMOVE PICTURE FROM LIBRARY removes from the Picture Library the picture whose reference number is passed in picRef.

If there is no picture with that reference number, the command does nothing.

4D Server: REMOVE PICTURE FROM LIBRARY cannot be used from within a method executed on the server machine (stored procedure or trigger). If you call REMOVE PICTURE FROM LIBRARY on a server machine, nothing happens—the call is ignored.

Warning: Design objects (hierarchical list items, menu items and so on) may refer to Picture Library graphics. Use caution when deleting a Picture Library graphic programmatically.

Examples

1. The following example deletes the picture #4444 from the Picture Library.

```
⇒	REMOVE PICTURE FROM LIBRARY(4444)
```

2. The following example deletes from the Picture Library the pictures whose names begin with a dollar sign ($):

```
	PICTURE LIBRARY LIST($alPicRef;$asPicName)
	For($vlPicture;1;Size of array($alPicRef))
	   If ($asPicName{$vlPicture}="$@")
⇒	      REMOVE PICTURE FROM LIBRARY($alPicRef{$vlPicture})
	   End if
	End for
```

See Also

GET PICTURE FROM LIBRARY, PICTURE LIBRARY LIST, SET PICTURE TO LIBRARY.

33 Process (Communications)

Semaphore

version 3

Semaphore (semaphore) → Boolean

Parameter	Type		Description
semaphore	String	→	Semaphore to test and set
Function result	Boolean	←	Semaphore has been successfully set (FALSE) or Semaphore was already set (TRUE)

Description

A semaphore is a flag shared among workstations (each user's computer) or among processes on the same workstation. A semaphore simply exists or does not exist. The methods that each user is running can test for the existence of a semaphore. By creating and testing semaphores, methods can communicate between workstations.

The Semaphore function returns TRUE if semaphore exists. If semaphore does not exist, Semaphore creates it and returns FALSE. Only one user at a time can create a semaphore. If Semaphore returns FALSE, it means that the semaphore did not exist, but it also means that the semaphore has been set for the process in which the call has been made.

Semaphore returns FALSE if the semaphore was not set. It also returns FALSE if the semaphore is already set by the same process in which the call has been made. A semaphore is limited to 15 characters. If you pass a longer string, the semaphore will be tested with the truncated string.

There are two types of semaphores in 4th Dimension: local semaphores and global semaphores.

A local semaphore is accessible by all processes on the same workstation and only on the workstation. A local semaphore can be created by prefixing the name of the semaphore with a dollar sign ($). You use local semaphores to monitor operations among processes executing on the same workstation. For example, a local semaphore can be used to monitor access to an interprocess array shared by all the processes in your single-user database or on the workstation.

A global semaphore is accessible to all users and all their processes. You use global semaphores to monitor operations among users of a multi-user database.

Global and local semaphores are identical in their logic. The difference resides in their scope, In 4D Server, global semaphores are shared among all the processes running on all clients. A local semaphore is only shared among the processes running on the client where it has been created.

In 4th Dimension, global or local semaphores have the same scope because you are the only user. However, if your database is being used in both setups, make sure to use global or local semaphores depending on what you want to do.

You do not use semaphores to protect record access. This is automatically done by 4th Dimension and 4D Server. Use semaphores to prevent several users from performing the same operation at the same time.

Examples

1. In this example, you want to prevent two users from doing a global update of the prices in a Products table. The following method uses semaphores to manage this:

```
⇒   If (Semaphore("UpdatePrices"))  ` Try to create the semaphore
       ALERT("Another user is already updating prices. Retry later.")
    Else
       DoUpdatePrices  ` Update all the prices
       CLEAR SEMAPHORE("UpdatePrices"))  ` Clear the semaphore
    End if
```

2. The following example uses a local semaphore. In a database with several processes, you want to maintain a To Do list. You want to maintain the list in an interprocess array and not in a table. You use a semaphore to prevent simultaneous access. In this situation, you only need to use a local semaphore, because your To Do list is only for your use.

The interprocess array is initialized in the Startup method:

```
    ARRAY TEXT(◊ToDoList;0) ` The To Do list is initially empty
```

Here is the method used for adding items to the To Do list:

```
       ` ADD TO DO LIST project method
       ` ADD TO DO LIST ( Text )
       ` ADD TO DO LIST ( To do list item )
    C_TEXT($1)
⇒   While (Semaphore("$AccessToDoList"))
       DELAY PROCESS(Current process;1)
    End while
    $vlElem:=Size of array(◊ToDoList)+1
    INSERT ELEMENT(◊ToDoList;$vlElem)
    ◊ToDoList{$vlElem}:=$1
    CLEAR SEMAPHORE("$AccessToDoList")  ` Clear the semaphore
```

You can call the above method from any process.

See Also

CLEAR SEMAPHORE.

CLEAR SEMAPHORE

CLEAR SEMAPHORE (semaphore)

Parameter	Type		Description
semaphore	String	→	Semaphore to clear

Description

CLEAR SEMAPHORE erases semaphore previously set by the Semaphore function.

As a rule, all semaphores that have been created should be cleared. If semaphores are not cleared, they remain in memory until the process that creates them ends. A process can only clear semaphores that it has created. If you try to clear a semaphore from within a process that did not create it, nothing happens.

Example

See the example for Semaphore.

See Also

Semaphore.

CALL PROCESS

Process (Communications)

version 3

CALL PROCESS (process)

Parameter	Type		Description
process	Number	→	Process number

Description

CALL PROCESS calls the form displayed in the frontmost window of process.

Important: CALL PROCESS only works between processes running on the same machine.

If you call a process that does not exist, nothing happens.

If process (the target process) is not currently displaying a form, nothing happens. The form displayed in the target process receives an On Outside call event. This event must be enabled for that form in the Design environment **Form Properties** window, and you must manage the event in the form method. If the event is not enabled or if it is not managed in the form method, nothing happens.

The caller process (the process from which CALL PROCESS is executed) does not "wait"—CALL PROCESS has an immediate effect. If necessary, you must write a waiting loop for a reply from the called process, using interprocess variables or using process variables (reserved for this purpose) that you can read and write between the two processes (using GET PROCESS VARIABLE and SET PROCESS VARIABLE).

To communicate between processes that do not display forms, use the commands GET PROCESS VARIABLE and SET PROCESS VARIABLE.

CALL PROCESS has the alternate syntax CALL PROCESS(-1).

In order not to slow down the execution of methods, 4th Dimension does not redraw interprocess variables each time they are modified. If you pass -1 instead of a process reference number in the process parameter, 4th Dimension does not call any process. Instead, it redraws all the interprocess variables currently displayed in all windows of any process running on the same machine.

Example

See example for On Exit Database Method.

See Also

Form event, GET PROCESS VARIABLE, SET PROCESS VARIABLE.

GET PROCESS VARIABLE (process; srcVar; dstVar{; srcVar2; dstVar2; ...; srcVarN; dstVarN})

Parameter	Type		Description
process	Number	→	Source process number
srcVar	Variable	→	Source variable
dstVar	Variable	←	Destination variable

Description

The command GET PROCESS VARIABLE reads the srcVar process variables (srvVar2, etc.) from the source process whose number is passed in process, and returns their current values in the dstVar variables (dstVar2, etc.) of the current process.

Each source variable can be a variable, an array or an array element. However, see the restrictions listed later in this section.

In each couple of srcVar;dstVar variables, the two variables must be of compatible types, otherwise the values you obtain may be meaningless.

The current process "peeks" the variables from the source process—the source process is not warned in any way that another process is reading the instance of its variables.

Restrictions

GET PROCESS VARIABLE does not accept local variables as source variables.

On the other hand, the destination variables can be interprocess, process or local variables. You "receive" the values only into variables, not into fields.

GET PROCESS VARIABLE accepts any type of source process or interprocess variable, except:
• Pointers
• Array of pointers
• Two-dimensional arrays

The source process must be a user process; it cannot be a kernel process. If the source process does not exist, an error is generated. You can catch this error using an error-handling method installed with ON ERR CALL.

Note: In interpreted mode, if a source variable does not exist, the undefined value is returned. You can detect this by using the Type function to test the corresponding destination variable.

Examples

1. The following line of code reads the value of the text variable vtCurStatus from the process whose number is $vlProcess. It returns the value in the process variable vtInfo of the current process:

```
⇒      GET PROCESS VARIABLE($vlProcess;vtCurStatus;vtInfo)
```

2. The following line of code does the same thing, but returns the value in the local variable $vtInfo for the method executing in the current process:

```
⇒      GET PROCESS VARIABLE($vlProcess;vtCurStatus;$vtInfo)
```

3. The following line of code does the same thing, but returns the value in the variable vtCurStatus of the current process:

```
⇒      GET PROCESS VARIABLE($vlProcess;vtCurStatus;vtCurStatus)
```

Note: The first vtCurStatus designates the instance of the variable in the source process The second vtCurStatus designates the instance of the variable in the current process.

4. The following example sequentially reads the elements of a process array from the process indicated by $vlProcess:

```
       GET PROCESS($vlProcess;vl_IPCom_Array;$vlSize)
       For($vlElem;1;$vlSize)
⇒         GET PROCESS VARIABLE($vlProcess;at_IPCom_Array{$vlElem};$vtElem)
             ` Do something with $vtElem
       End for
```

Note: In this example, the process variable vl_IPCom_Array contains the size of the array at_IPCom_Array, and must be maintained by the source process.

5. The following example does the same thing as the previous one, but reads the array as a whole, instead of reading the elements sequentially:

```
⇒      GET PROCESS($vlProcess;at_IPCom_Array;$anArray)
       For($vlElem;1;Size of array($anArray))
          ` Do something with $anArray{$vlElem}
       End for
```

6. The following example reads the source process instances of the variables v1,v2,v3 and returns their values in the instance of the same variables for the current process:

```
⇒	GET PROCESS VARIABLE($vlProcess;v1;v1;v2;v2;v3;v3)
```

7. See the example for the command DRAG AND DROP PROPERTIES.

See Also

CALL PROCESS, Drag and Drop, DRAG AND DROP PROPERTIES, Processes, SET PROCESS VARIABLE, VARIABLE TO VARIABLE.

SET PROCESS VARIABLE

Process (Communications)

version 6.0

SET PROCESS VARIABLE (process; dstVar; expr{; dstVar2; expr2; ...; dstVarN; exprN})

Parameter	Type		Description
process	Number	→	Destination process number
dstVar	Variable	→	Destination variable
expr	Variable	→	Source expression (or source variable)

Description

The command SET PROCESS VARIABLE writes the dstVar process variables (dstVar2, etc.) of the destination process whose number is passed in process using the values passed in expr1 (expr2, etc.).

Each destination variable can be a variable or an array element. However, see the restrictions listed later in this section.

For each couple of dstVar;expr variables, the expression must be of a type compatible with the destination variable, otherwise you may end up with a meaningless value in the variable. In interpreted mode, if a destination variable does not exist, it is created and assigned with the expression.

The current process “pokes” the variables of the destination process—the destination process is not warned in any way that another process is writing the instance of its variables.

Restrictions

SET PROCESS VARIABLE does not accept local variables as destination variables.

SET PROCESS VARIABLE accepts any type of destination process variables except:
• Pointers
• Arrays of any type. To write an array as a whole from one process to another one, use the command VARIABLE TO VARIABLE. Note, however, that SET PROCESS VARIABLE allows you to write the element of an array.
• You cannot write the element of an array of pointers or the element of a two-dimensional array.

The destination process must be a user process; it cannot be a kernel process. If the destination process does not exist, an error is generated. You can catch this error using an error-handling method installed with ON ERR CALL.

Examples

1. The following line of code sets (to the empty string) the text variable vtCurStatus of the process whose number is $vlProcess:

```
⇒     SET PROCESS VARIABLE($vlProcess;vtCurStatus;"")
```

2. The following line of code sets the text variable vtCurStatus of the process whose number is $vlProcess to the value of the variable $vtInfo from the executing method in the current process:

```
⇒     SET PROCESS VARIABLE($vlProcess;vtCurStatus;$vtInfo)
```

3. The following line of code sets the text variable vtCurStatus of the process whose number is $vlProcess to the value of the same variable in the current process:

```
⇒     SET PROCESS VARIABLE($vlProcess;vtCurStatus;vtCurStatus)
```

Note: The first vtCurStatus designates the instance of the variable in the destination process. The second vtCurStatus designates the instance of the variable in the current process.

4. The following example sequentially sets to uppercase all elements of a process array from the process indicated by $vlProcess:

```
      GET PROCESS VARIABLE($vlProcess;vl_IPCom_Array;$vlSize)
      For($vlElem;1;$vlSize)
         GET PROCESS VARIABLE($vlProcess;at_IPCom_Array{$vlElem};$vtElem)
⇒        SET PROCESS VARIABLE($vlProcess;at_IPCom_Array{$vlElem};Uppercase($vtElem))
      End for
```

Note: In this example, the process variable vl_IPCom_Array contains the size of the array at_IPCom_Array and must be maintained by the source/destination process.

5. The following example writes the destination process instance of the variables v1, v2 and v3 using the instance of the same variables from the current process:

```
⇒     SET PROCESS VARIABLE($vlProcess;v1;v1;v2;v2;v3;v3)
```

See Also

CALL PROCESS, GET PROCESS VARIABLE, Processes, VARIABLE TO VARIABLE.

VARIABLE TO VARIABLE

Process (Communications)

version 6.0.2

VARIABLE TO VARIABLE (process; dstVar; srcVar{; dstVar2; srcVar2; ...; dstVarN; srcVarN})

Parameter	Type		Description
process	Number	→	Destination process number
dstVar	Variable	→	Destination variable
srcVar	Variable	→	Source variable

Description

The command VARIABLE TO VARIABLE writes the dstVar process variables (dstVar2, etc.) of the destination process whose number is passed in process using the values of the variables srcVar1 (srcVar2, etc.).

VARIABLE TO VARIABLE does the same thing as SET PROCESS VARIABLE, with the following differences:
• While you pass source expressions to SET PROCESS VARIABLE (and therefore cannot pass an array as a whole), you must exclusively pass source variables to VARIABLE TO VARIABLE (and therefore can pass an array as a whole).
• With SET PROCESS VARIABLE, each destination variable can be a variable or an array element, but cannot be an array as a whole. With VARIABLE TO VARIABLE, each destination variable can be a variable or an array or an array element.

For each couple of dstVar;expr variables, the source variable must be of a type compatible with the destination variable, otherwise you may end up with a meaningless value in the variable. In interpreted mode, if a destination variable does not exist, it is created and assigned with the type and value of the source variable.

The current process "pokes" the variables of the destination process—the destination process is not warned in any way that another process is writing the instance of its variables.

Restrictions

VARIABLE TO VARIABLE does not accept local variables as destination variables.

VARIABLE TO VARIABLE accepts any type of destination process or interprocess variables except:
• Pointers
• Array of pointers
• Two-dimensional arrays

The destination process must be a user process; it cannot be a kernel process. If the destination process does not exist, an error is generated. You can catch this error using an error-handling method installed with ON ERR CALL.

Example

The following example reads a process array from the process indicated by $vlProcess, sequentially sets the elements to uppercase and then writes back the array as a whole:

```
   GET PROCESS VARIABLE($vlProcess;at_IPCom_Array;$anArray)
   For($vlElem;1;Size of array($anArray))
      $anArray{$vlElem}:=Uppercase($anArray{$vlElem})
   End for
⇒  VARIABLE TO VARIABLE($vlProcess;at_IPCom_Array;$anArray)
```

See Also

GET PROCESS VARIABLE, Processes, SET PROCESS VARIABLE.

34 Process (User Interface)

HIDE PROCESS

Process (User Interface)

version 3

HIDE PROCESS (process)

Parameter	Type		Description
process	Number	→	Process number or process to be hidden

Description

HIDE PROCESS hides all windows that belong to process. All interface elements of process are hidden until the next SHOW PROCESS. The menu bar of the process is also hidden. This means that opening a window while the process is hidden does not make the screen redraw or display. If the process is already hidden, the command has no effect.

The only exception to this rule is the Debugger window. If the Debugger window is displayed when process is a hidden process, process is displayed and becomes the frontmost process.

If you do not want a process to be displayed when it is created, HIDE PROCESS should be the first command in the process method. The User/Custom Menus and Cache Manager processes cannot be hidden using this command.

Even though a process may be hidden, the process is still executing.

Example

The following example hides all the windows belonging to the current process:

```
⇒	HIDE PROCESS (Current process)
```

See Also

Process state, SHOW PROCESS.

SHOW PROCESS

SHOW PROCESS (process)

Parameter	Type		Description
process	Number	→	Process number of process to be shown

Description

SHOW PROCESS displays all the windows belonging to process. This command does not bring the windows of process to the frontmost level. To do this, use the BRING TO FRONT command.
If the process was already displayed, the command has no effect.

Example

The following example displays a process called Customers, if it has been previously hidden. The process reference to the Customers process is stored in the interprocess variable <>Customers:

```
⇒	SHOW PROCESS (<>Customers)
```

See Also

BRING TO FRONT, HIDE PROCESS, Process state.

BRING TO FRONT

Process (User Interface)

version 3

BRING TO FRONT (process)

Parameter	Type		Description
process	Number	→	Process number of the process to pass to the frontmost level

Description

BRING TO FRONT brings all the windows belonging to process to the front. The order of the windows is retained. If the process is already the frontmost process, the command does nothing. If the process is hidden, you must use SHOW PROCESS to display the process, otherwise BRING TO FRONT has no effect.

The User/Custom Menus and Design processes can be brought to the front using this command.

Example

The following example is a method that can be executed from a menu. It checks to see if <>vlAddCust_PID is the frontmost process. If not, the method brings it to the front:

```
    If (Frontmost process#<>vlAddCust_PID)
⇒       BRING TO FRONT (<>vlAddCust_PID)
    End if
```

See Also

HIDE PROCESS, Process state, SHOW PROCESS.

Frontmost process

Process (User Interface)

version 3

Frontmost process {(*)} → Integer

Parameter	Type		Description
*		→	Process number for first non-floating window
Function result	Integer	←	Number of the process whose windows are in the front

Description

Frontmost process returns the number of the process whose window (or windows) are in the front.

When you have one or more floating windows open, there are two window layers:
• Regular windows
• Floating windows

If the Frontmost process function is used from within a floating window form method or object method, the function returns the process reference number of the frontmost floating window in the floating window layer. If you specify the optional * parameter, the function returns the process reference number of the frontmost active window in the regular window layer.

Example

See the example for BRING TO FRONT.

See Also

BRING TO FRONT, WINDOW LIST.

35 Processes

Processes

version 6.0

Multi-tasking in 4th Dimension is the ability to have distinct database operations that are executed simultaneously. These operations are called processes.

Multiple processes are like multiple users on the same computer, each working on his or her own task. This essentially means that each method can be executed as a distinct database task.

This section covers the following topics:

- Creating and clearing processes
- Elements of a process
- User processes
- Processes created by 4th Dimension
- Local and global processes
- Record locking between processes

Note: This section does not cover stored procedures. See the section Stored Procedures in the *4D Server Reference manual*.

Creating and Clearing Processes

There are three ways to create a new process:

- Execute a method in the User environment after checking the **New Process** check box in the **Execute Method** dialog box. The method chosen in the Execute Method dialog box is the process method.
- Processes can be run by choosing menu commands. In the Design environment's **Menu Bar editor**, select the menu command and click the **Start a New Process** check box. The method associated with the menu command is the process method.
- Use the New process function. The method passed as a parameter to the New process function is the process method.

A process can be cleared under the following conditions. The first two conditions are automatic:

- When the process method finishes executing
- When the user quits from the database
- If you stop the process procedurally or use the Abort button in the Debugger
- If you choose **Abort** from the **Process** menu in the Design environment

A process can create another process. Processes are not organized hierarchically—all processes are equal, regardless of the process from which they have been created. Once the "parent" process creates a "child" process, the child process will continue regardless of whether or not the parent process is still executing.

Elements of a Process

Each process contains specific elements. There are three types of distinctly different elements in a process:
• **Interface elements**: Elements that are necessary to display a process.
• **Data elements**: Information that is related to the data in the database.
• **Language elements**: Elements that are used procedurally or are that are important for developing your own application.

Interface Elements

Interface elements consist of the following:
• **Menu bar**: Each process can have its own current menu bar. The menu bar of the frontmost process is the current menu bar for the database.
• **One or more windows**: Each process can have more than one window open simultaneously. On the other hand, some processes have no windows at all.
• **One active (frontmost) window**: Even though a process can have several windows open simultaneously, each process has only one active window. To have more than one active window, you must start more than one process.

Data Elements

Data elements refer to the data used by the database. The data elements are:
• **Current selection per table**: Each process has a separate current selection. One table can have a different current selection in different processes.
• **Current record per table**: Each table can have a different current record in each process.

Note: This description of the data elements is valid if your processes are global in scope. By default, all processes are global. See the “Global and Local Processes” section below.

Language Elements

The language elements of a process are the elements related to programming in 4th Dimension.

• **Variables**: Every process has its own process variables. See Variables for more information. Process variables are recognized only within the domain of their native process.
• **Default table**: Each process has its own default table. However, note that the DEFAULT TABLE command is only a typing convention for programming.
• **Input and Output forms**: Default input and output forms can be set procedurally for each table in each process.
• **Process sets**: Each process has its own process sets. UserSet and LockedSet are process sets. Process sets are cleared as soon as the process method ends.
• **On Error Call per process**: Each process has its own error-handling method.
• **Debugger window**: Each process can have its own Debugger window.

User Processes

User processes are processes that you create to perform certain tasks. They share processing time with the kernel processes. As an example, Web connection processes are user processes.

Processes Created by 4th Dimension

The following processes are created and managed by 4th Dimension:

• **User/Custom Menus process**: The User/Custom Menus process consists of the Custom Menus and the User environments. The default splash screen window in the Custom Menus environment is also a part of the User/Custom Menus process. This process is created as soon as 4th Dimension is run.

• **Design process**: The Design process consists of the Design environment running as a separate process. It can be closed using the **Exit Design Environment** menu command in the **File** menu of the Design environment. There is no Design process in a compiled database. The Design process is created only when the user enters the Design environment for the first time. If the application opens in the User or Custom Menus environment, by default, the process will not be created.

• **Web Server process**: The Web Server process runs when the database is published on the Web. See the section Web Services, Web Connection Processes for more information.

• **Cache Manager process**: The Cache Manager process manages disk I/ O for the database. This process is created as soon as 4th Dimension or 4D Server are run.

• **Indexing process**: The Indexing process manages the indexing of fields in a database as a separate process. This process is created when an index for a field is built or deleted.

• **On Serial Port Manager process**: The On Serial Port Manager process has the serial port-handling method as the process method. This process is created when a serial event handling method is installed by the ON SERIAL PORT CALL command.

• **On Event Manager process**: This process is created when an event-handling method is installed by the ON EVENT CALL command. It executes the event method installed by ON EVENT CALL whenever there is an event. The event method is the process method for this process. This process executes continuously, even if no method is executing. Event handling also occurs in the Design environment.

Global and Local Processes

Processes can be either global or local in scope. By default, all processes are global.

Global processes can perform any operation, including accessing and manipulating data. In most cases, you will want to use global processes.

Local processes should be used only for operations that do not access data. For example, you can use a local process to run an event-handling method or to control interface elements such as floating windows.

You specify that a process is local in scope through its name. The name of local process must start with a dollar sign ($).

Warning: If you attempt to access data from a local process, you access it though the User/Custom Menus process, risking conflicts with operations performed within that process.

4D Server: Using local processes on the Client side for operations that do not require data access reserves more processing time for server-intensive tasks.

Record Locking Between Processes

A record is locked when another process has successfully loaded the record for modification. A locked record can be loaded by another process, but cannot be modified. The record is unlocked only in the process in which the record is being modified. A table must be in read/write mode for a record to be loaded unlocked.

See Also

Methods, Project Methods, Variables.

New process

version 6.0 (modified)

New process (method; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number

Parameter	Type		Description
method	String	→	Method to be executed within the process
stack	Number	→	Stack size in bytes
name	String	→	Name of the process created
param	Expression	→	Parameter(s) to the method
*		→	Unique process
Function result	Number	←	Process number for newly created process or already executing process

Description

The command New process starts a new process (on the same machine) and returns the process number for that process.

If the process could not be created (for example, if there is not enough memory), New process returns zero (0) and an error is generated. You can catch this error using an error-handling method installed using ON ERR CALL.

Process Method: In method, you pass the name of the process method for the new process. After 4D has set up the context for the new process, it starts executing this method, which therefore becomes the process method.

Process Stack: In stack, you pass the amount of memory allocated for the stack of the process. It is the space in memory used to “pile up” method calls, local variables, parameters in subroutines, and stacked records. It is expressed in bytes; you will usually pass at least 32K (around 32000 bytes), but you can pass more if the process can perform large chain calls (subroutines calling subroutines in cascade). For example, you can pass 200K (around 200000 bytes), if necesary.

Note: The stack is NOT the total memory for the process. Processes share memory for records, interprocess variables, and so on. A process also uses extra memory for storing its process variables. The stack only holds local variables, method calls, parameters in subroutines and stacked records.

Process Name: You pass the name of the new process in name. This name will appear in the **Process List** window of the Design environment, and will be returned by the command PROCESS PROPERTIES when applied to this new process. A process name can be up to 31 characters long. You can omit this parameter; if you do so, the name of the process will be the empty string. You can make a process local in scope by prefixing its name with the dollar sign ($).

Important: Remember that local processes should not access data in Client/Server.

Parameter to Process Method: Starting with version 6, you can pass parameters to the process method. You can pass parameters in the same way as you would pass them to a subroutine. However, there is a restriction—you cannot pass pointer expressions. Also, remember that arrays cannot be passed as parameters to a method. Upon starting execution in the context of the new process, the process method receives the parameters values in $1, $2, etc.

Note: If you pass parameters to the process method, you must pass the name parameter; it cannot be omitted in this case.

The optional * parameter: Specifying this last parameter tells 4D to first check whether or not a process with the name you passed in name is already running. If it is, 4D does not start a new process and returns the process number of the process with that name.

Example

Given the following project method:

```
   ` ADD CUSTOMERS
MENU BAR (1)
Repeat
   ADD RECORD([Customers];*)
Until (OK=0)
```

If you attach this project method to a custom menu item in the Design environment **Menu Bar Editor** window whose **Start a New Process** property is set, 4D will automatically start a new process running that method. The call MENU BAR(1) adds a menu bar to the new process. In the absence of any window (that you could open with Open window), the call to ADD RECORD will automatically open one.

To be able to start this Add Customers process when you click on a button in a custom control panel, you can write:

```
      ` bAddCustomers button object method
⇒   $vlProcessID:=New process("Add Customers";32*1024;"Adding Customers")
```

The button does the same thing as the custom menu item.

While choosing the menu item or clicking the button, if you want to start the process (if it does not exist) or bring it to the front (if it is already running), you can create the method START ADD CUSTOMERS:

```
	` START ADD CUSTOMERS
⇒	$vlProcessID:=New process("Add Customers";32*1024;"Adding Customers";*)
	If ($vlProcessID#0)
		BRING TO FRONT ($vlProcessID)
	End if
```

The object method of the bAddCustomers becomes:

```
	` bAddCustomers button object method
	START ADD CUSTOMERS
```

In the Menu Bar editor, you replace the method ADD CUSTOMERS with the method START ADD CUSTOMERS, and you deselect the **Start a New Process** property for the menu item.

See Also

Execute on server, Methods, Processes, Project Methods, Variables.

Execute on server

version 6.0

Execute on server (procedure; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number

Parameter	Type		Description
procedure	String	→	Procedure to be executed within the process
stack	Number	→	Stack size in bytes
name	String	→	Name of the process created
param	Expression	→	Parameter(s) to the procedure
*		→	Unique process
Function result	Number	←	Process number for newly created process or already executing process

Description

The command Execute on server starts a new process on the Server machine (if it is called in Client/Server) or on the same machine (if it is called in single-user) and returns the process number for that process.

You use Execute on server to start a stored procedure. For more information about stored procedures, see the section Stored Procedures in the *4D Server Reference* manual.

If you call Execute on server on a Client machine, the command returns a negative process number. If you call Execute on server on the Server machine, Execute on server returns a positive process number. Note that calling New process on the Server machine does the same thing as calling Execute on server.

If the process could not be created (for example, if there is not enough memory), Execute on server returns zero (0) and an error is generated. You can catch this error using an error-handling method installed using ON ERR CALL.

Process Method: In method, you pass the name of the process method for the new process. After 4D has set up the context for the new process, it starts executing this method, which therefore becomes the process method.

Process Stack: In stack, you pass the amount of memory allocated for the stack of the process. It is the space in memory used to “pile up” method calls, local variables, parameters in subroutines, and stacked records. It is expressed in bytes; you will usually pass at least 32K (around 32000 bytes), but you can pass more if the process can perform large chain calls (subroutines calling subroutines in cascade). For example, you can pass 200K (around 200000 bytes), if necesary.

Note: The stack is NOT the total memory for the process. Processes share memory for records, interprocess variables, and so on. A process also uses extra memory for storing its process variables. The stack only holds local variables, method calls, parameters in subroutines and stacked records.

Process Name: You pass the name of the new process in name. In single-user, this name will appear in the Process List window of the Design environment, and will be returned by the command PROCESS PROPERTIES when applied to this new process. In Client/Server, this name will appear in blue in the Stored Procedure list of the 4D Server main window.

A process name can be up to 31 characters long. You can omit this parameter; if you do so, the name of the process will be the empty string.

Warning: Contrary to New Process, do not attempt to make a process local in scope by prefixing its name with the dollar sign ($) while using Execute on server. This will work in single-user, because Execute on server acts as New Process in this environment. On the other hand, in Client/Server, this will generate an error.

Parameter to Process Method: Starting with version 6, you can pass parameters to the process method. You can pass parameters in the same way as you would pass them to a subroutine. However, there is a restriction—you cannot pass pointer expressions. Also, remember that arrays cannot be passed as parameters to a method. Upon starting execution in the context of the new process, the process method receives the parameters values in $1, $2, etc.

Note: If you pass parameters to the process method, you must pass the name parameter; it cannot be omitted in this case.

The optional * parameter: Specifying this last parameter tells 4D to first check whether or not a process with the name you passed in name is already running. If it is, 4D does not start a new process and returns the process number of the process with that name.

Example

(1) The following example shows how importing data can be dramatically accelerated in Client/Server. The Regular Import method listed below allows you to test how long it takes to import records using the IMPORT TEXT command on the Client side:

```
   ` Regular Import Project Method
$vhDocRef:=Open document("")
If (OK=1)
   CLOSE DOCUMENT($vhDocRef)
   INPUT FORM([Table1];"Import")
   $vhStartTime:=Current time
   IMPORT TEXT([Table1];Document)
   $vhEndTime:=Current time
   ALERT("It took "+String(0+($vhEndTime-$vhStartTime))+" seconds.")
End if
```

With the regular import data, 4D Client performs the parsing of the text file, then, for each record, create a new record, fills out the fields with the imported data and sends the record to the Server machine so it can be added to the database. There are consequently many requests going over the network. A way to optimize the operation is to use a stored procedure to do the job locally on the Server machine. The Client machine loads the document into a BLOB, start a stored procedure passing the BLOB as parameter. The stored procedure stores the BLOB into a document on the server machine disk, then imports the document locally. The import data is therefore performed locally at a single-user version-like speed because most the network requests are eliminated. Here is the CLIENT IMPORT project method. Executed on the Client machine, it starts the SERVER IMPORT stored procedure listed just below:

```
	` CLIENT IMPORT Project Method
	` CLIENT IMPORT ( Pointer ; String )
	` CLIENT IMPORT ( -> [Table] ; Input form )

C_POINTER($1)
C_STRING(31;$2)
C_TIME($vhDocRef)
C_BLOB($vxData)
C_LONGINT(spErrCode)

	` Select the document do be imported
$vhDocRef:=Open document("")
If (OK=1)
		` If a document was selected, do not keep it open
	CLOSE DOCUMENT($vhDocRef)
	$vhStartTime:=Current time
		` Try to load it in memory
	DOCUMENT TO BLOB(Document;$vxData)
	If (OK=1)
			` If the document could be loaded in the BLOB,
			` Start the stored procedure that will import the data on the server machine
		$spProcessID:=Execute on server("SERVER IMPORT";32*1024;
							"Server Import Services";Table($1);$2;$vxData)
			` At this point, we no longer need the BLOB in this process
		CLEAR VARIABLE($vxData)
			` Wait for the completion of the operation performed by the stored procedure
		Repeat
			DELAY PROCESS(Current process;300)
			GET PROCESS VARIABLE($spProcessID;spErrCode;spErrCode)
			If (Undefined(spErrCode))
					` Note: if the stored procedure has not initialized its own instance
					` of the variable spErrCode, we may be returned an undefined variable

				spErrCode:=1
			End if
		Until (spErrCode<=0)
```

```
            ` Tell the stored procedure that we acknowledge
         spErrCode:=1
         SET PROCESS VARIABLE($spProcessID;spErrCode;spErrCode)
         $vhEndTime:=Current time
         ALERT("It took "+String(0+($vhEndTime-$vhStartTime))+" seconds.")
      Else
         ALERT("There is not enough memory to load the document.")
      End if
   End if
```

Here is the SERVER IMPORT project method executed as a stored procedure:

```
      ` SERVER IMPORT Project Method
      ` SERVER IMPORT ( Long ; String ; BLOB )
      ` SERVER IMPORT ( Table Number ; Input form ; Import Data )

   C_LONGINT($1)
   C_STRING(31;$2)
   C_BLOB($3)
   C_LONGINT(spErrCode)

      ` Operation is not finished yet, set spErrCode to 1
   spErrCode:=1
   $vpTable:=Table($1)
   INPUT FORM($vpTable->;$2)
   $vsDocName:="Import File "+String(1+Random)
   DELETE DOCUMENT($vsDocName)
   BLOB TO DOCUMENT($vsDocName;$3)
   IMPORT TEXT($vpTable->;$vsDocName)
   DELETE DOCUMENT($vsDocName)
      ` Operation is finished, set spErrCode to 0
   spErrCode:=0
      ` Wait until the requester Client got the result back
   Repeat
      DELAY PROCESS(Current process;1)
   Until (spErrCode>0)
```

Once these two project methods have been implemented in a database, you can perform a “Stored Procedure-based” import data by, for instance, writing:

```
   CLIENT IMPORT (->[Table1];"Import")
```

With some benchmarks you will discover that using this method you can import records up to 60 times faster than the regular import.

See Also

New process, Stored Procedures.

DELAY PROCESS

DELAY PROCESS (process; duration)

Parameter	Type		Description
process	Number	→	Process number
duration	Number	→	Duration expressed in ticks

Description

DELAY PROCESS delays the execution of a process for a number of ticks (1 tick = 1/60th of a second). During this period, process does not take any processing time. Even though the execution of a process may be delayed, it is still in memory.

If the process is already delayed, this command delays it again. The parameter duration is not added to the time remaining, but replaces it. Therefore pass zero (0) for duration if you no longer want to delay a process.

If the process does not exist, the command does nothing.

Warning: DELAY PROCESS has no effect on the Kernel processes (all environments), including the User/Custom Menus process.

Tip: To "delay" the User/Custom Menus process, write a small "waiting" subroutine that loops measuring the elasped time using Current time, or Tickcount or Milliseconds). For example, if you want to display, for a given time period, a message in a window that you open and close for this purpose.

Tip: On the one machine, when a process is waiting for another process that executes code rather than waiting for a user action, calling DELAY PROCESS repeatedly (with duration equal to 1) from within the "waiting" process provides the best result in terms of execution speed. The reason for this is that the scheduler "gives" most of the time to the executing process, not to the "waiting" process. (See the second example for Semaphore.)

Examples

1. See example for the command Semaphore.
2. See example in Record Locking.
3. See example for the command Process number.

See Also

HIDE PROCESS, PAUSE PROCESS.

PAUSE PROCESS

PAUSE PROCESS (process)

Parameter	Type		Description
process	Number	→	Process number

Description

PAUSE PROCESS suspends the execution of process until it is reactivated by the RESUME PROCESS command. During this period, process does not take any time on your machine. Even though a process may be paused, the process is still in memory.

If process is already paused, PAUSE PROCESS does nothing. If the process has been delayed using the DELAY PROCESS command, the process is paused. RESUME PROCESS resumes the process immediately.

While process execution is suspended, the windows belonging to this process are not enterable. In this case, to avoid confusing the user, consider hiding the process. If process does not exist, the command does nothing.

Warning: Use PAUSE PROCESS only in processes that you have started. PAUSE PROCESS will not affect the original User/Custom Menus process.

See Also

DELAY PROCESS, HIDE PROCESS, RESUME PROCESS.

RESUME PROCESS

RESUME PROCESS (process)

Parameter	Type		Description
process	Number	→	Process number

Description

RESUME PROCESS resumes a process whose execution has been paused or delayed. If process is not paused or delayed, RESUME PROCESS does nothing.

If process has been delayed before, see the PAUSE PROCESS or DELAY PROCESS commands. If process does not exist, the command does nothing.

See Also

DELAY PROCESS, PAUSE PROCESS.

Current process

Processes

version 3

Current process → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Process number

Description

Current process returns the process reference number of the process within which this command is called.

Example

See the examples for DELAY PROCESS and PROCESS ATTRIBUTES.

See Also

Process number, PROCESS PROPERTIES, Process state.

Process state

Process state (process) → Number

Parameter	Type		Description
process	Number	→	Process number
Function result	Number	←	State of the process

Description

The command Process state returns the state of the process whose number you pass in process.

The function result can be one of the values provided by the following predefined constants:

Constant	Type	Value
Aborted	Long Integer	-1
Delayed	Long Integer	1
Does not exist	Long Integer	-100
Executing	Long Integer	0
Hidden modal dialog	Long Integer	6
Paused	Long Integer	5
Waiting for input output	Long Integer	3
Waiting for internal flag	Long Integer	4
Waiting for user event	Long Integer	2

If the process does not exist (which means you did not pass a number in the range 1 to Count tasks), Process state returns Does not exist (-100).

Example

The following example puts the name and process reference number for each process into the asProcName and aiProcNum arrays. The method checks to see if the process has been aborted. In this case, the process name and number are not added to the arrays:

```
$vlNbTasks:=Count tasks
ARRAY STRING(31;arProcName; $vlNbTasks)
ARRAY INTEGER(aiProcNum; $vlNbTasks)
$vlActualCount:=0
For ($vlProcess;1; $vlNbTasks)
    If (Process state($vlProcess)>=Executing)
        $vlActualCount:=$vlActualCount+1
        PROCESS PROPERTIES($vlProcess; asProcName{$vlActualCount};
                                                                $vlState;$vlTime)
        aiProcNum{$vlActualCount}:=$vlProcess
    End if
End for
    ` Eliminate unused extra elements
ARRAY STRING(31;asProcName;$vlActualCount)
ARRAY INTEGER(aiProcNum;$vlActualCount)
```

See Also

Count tasks, PROCESS PROPERTIES.

PROCESS PROPERTIES

version 6.0 (Modified)

PROCESS PROPERTIES (process; procName; procState; procTime{; procVisible})

Parameter	Type		Description
process	Number	→	Process number
procName	String	←	Process name
procState	Number	←	Process state
procTime	Number	←	Cumulative time taken by process in ticks
procVisible	Boolean	←	Visible (TRUE) or Hidden (FALSE)

Description

The command PROCESS PROPERTIES returns information about the process whose process number you pass in process.

After the call:

• procName returns the name of the process. Some things to note about the process name:

- If the process was started from the User environment **Execute Method** dialog box (with the **New Process** option selected), its name is "P_" followed by a number.
- If the process was started from a Custom menu item whose **Start a New Process** property is checked, the name of the process is "M_" or "ML_" followed by a number.
- If the process has been aborted (and its "slot" not reused yet), the name of the process is still returned. To detect if a process is aborted, test procState=-1 (see below).

• procState returns the state of the process at the moment of the call. This parameter can return one of the values provided by the following predefined constants:

Constant	Type	Value
Aborted	Long Integer	-1
Delayed	Long Integer	1
Does not exist	Long Integer	-100
Executing	Long Integer	0
Hidden modal dialog	Long Integer	6
Paused	Long Integer	5
Waiting for input output	Long Integer	3
Waiting for internal flag	Long Integer	4
Waiting for user event	Long Integer	2

• procTime returns the cumulative time that the process has used since it started, in ticks (1/60th of a second) .

• procVisible, if specified, returns TRUE if the process is visible, FALSE if hidden.

If the process does not exist, which means you did not pass a number in the range 1 to Count tasks, PROCESS PROPERTIES leaves the variable parameters unchanged.

Examples

1. The following example returns the name, state, and time taken in the variables vName, vState, and vTimeSpent for the current process:

```
C_STRING(80; vName) ` Initialize the variables
C_INTEGER(vState)
C_INTEGER(vTime)
PROCESS PROPERTIES (Current process; vName; vState; vTimeSpent)
```

2. See example for On Exit Database Method.

See Also

Count tasks, Process state.

Process number

Process number (name) → Number

Parameter	Type		Description
name	String	→	Name of process for which to retrieve the process number
Function result	Number	←	Process number

Description

The command Process number returns the number of the process whose name you pass in name. If no process is found, Process number returns 0.

Example

You create a custom floating window, run in a separate process, in which you implement your own tools to interact with the Design environment. For example, when selecting an item in a hierarchical list of keywords, you want to paste some text into the frontmost window of the Design environment. To do so, you can use the clipboard, but the pasting event must occur within the Design process. The following small function returns the process number of the Design process (if running):

```
     ` Design process number Project Method
     ` Design process number -> LongInt
     ` Design process number -> Design process number

⇒  $0:=Process number(Get indexed string(170;3))
     ` The name of the Design process is stored in the 'STR#" resource ID=170,
     ` string #3 within 4D
     ` Note: This can break in the future if the resource changes
```

Using this function, the project method listed below, paste the text received as parameter to the frontmost window of the Design environment (if applicable):

```
     ` PASTE TEXT TO DESIGN Project Method
     ` PASTE TEXT TO DESIGN ( Text )
     ` PASTE TEXT TO DESIGN ( Text to Paste in frontmost Design window )

C_TEXT($1)
C_LONGINT($vlDesignPID;$vlCount)

$vlDesignPID:=Design process number
If ($vlDesignPID # 0)
          ` Put the text into the clipboard
     SET TEXT TO CLIPBOARD($1)
          ` Post a Ctrl-V / Command-V event
     POST KEY(Ascii("v");Command key mask;$vlDesignPID)
          ` Call repeatedly DELAY PROCESS so the scheduler gets a chance to pass over
          ` the event to the Design process
     For ($vlCount;1;5)
          DELAY PROCESS(Current process;1)
     End for
End if
```

See Also

PROCESS PROPERTIES, Process state.

Count users

Count users → Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Integer	←	Number of users connected to the server

Description

The Count users function returns the number of users connected to the database.

In the single-user version of 4th Dimension, Count users returns 1.

See Also

Count tasks, Count user processes.

Count tasks

Count tasks → Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Integer	←	Number of open processes (including kernel processes)

Description

Count tasks returns the number of processes open on a workstation or in single-user 4th Dimension.

This number takes into account all processes, even those that are automatically managed by 4th Dimension. These include the User/Custom Menus process, Design process, Cache Manager process, Indexing process, and Web Server process.

The number returned by Count tasks also takes into account processes that have been aborted.

Example

See the example for Process state and On Exit Database Method.

See Also

Count user processes, Count users, PROCESS PROPERTIES, Process state.

Count user processes

Processes

version 3

Count user processes → Integer

Parameter	Type		Description
This command does not require any parameters			
Function result	Integer	←	Number of open processes (excluding kernel processes)

Description

The Count user processes function returns the number of open processes, except those processes that are managed automatically by 4th Dimension.

The User/Custom Menus process is considered to be a user process. Therefore, this process will be counted when determining the number of user processes.

See Also

Count tasks, Count users.

36 Queries

QUERY BY EXAMPLE

QUERY BY EXAMPLE {(table)}

Parameter	Type		Description
table	Table	→	Table for which to return a selection of records, or Default table, if omitted

Description

QUERY BY EXAMPLE performs the same action as does the Query by Example menu command in the User environment. It displays the current input form as a query window. QUERY BY EXAMPLE queries table for the data that the user enters into the query window. The form must contain the fields that you want the user to be able to query. The query is optimized; indexed fields are automatically used to optimize the query.

See the 4th Dimension User Reference for information about using the Query by Example menu command in the User environment.

Examples

The method in the following example displays the form named MyQuery to the user. If the user accepts the form and performs the query (that is, if the OK system variable is set to 1), the records that meet the query criteria are displayed:

```
   INPUT FORM ([People]; "MyQuery")  ` Switch to query form
⇒  QUERY BY EXAMPLE ([People])  ` Display form and perform query
   If (OK=1)  ` If the user performed the query
      DISPLAY SELECTION ([People])  ` Display the records
   End if
```

See Also

ORDER BY, QUERY.

System Variables or Sets

If the user clicks an Accept button or presses the Enter key, the OK system variable is set to 1 and the query is performed. If the user clicks a Cancel button or presses the “cancel” key combination, the OK system variable is set to 0 and the query is canceled

QUERY

QUERY ({table}{; queryArgument}{; *})

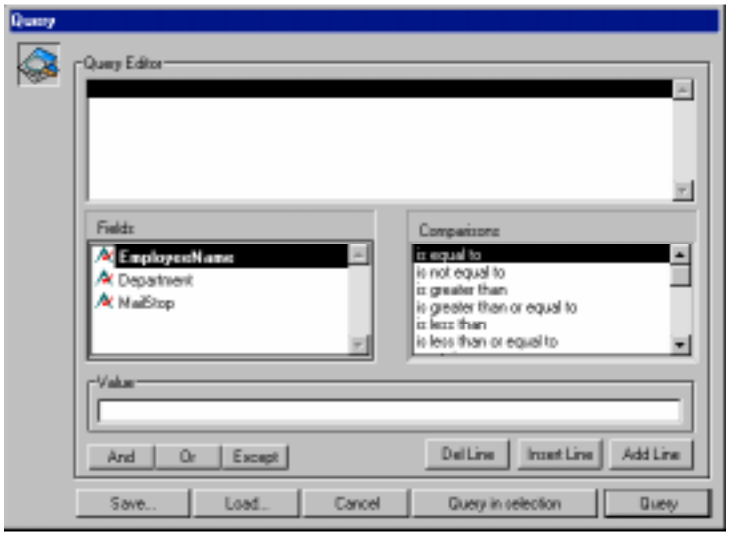
Parameter	Type		Description
table	Table	→	Table for which to return a selection of records, or Default table, if omitted
queryArgument		→	Query argument
*		→	Continue query flag

Description

QUERY looks for records matching the criteria specified in queryArgument and returns a selection of records for table. QUERY changes the current selection of table for the current process and makes the first record of the new selection the current record.

If the table parameter is omitted, the command applies to the default table. If no default table has been set, an error occurs.

If you do not specify queryArgument or the * parameters, QUERY displays the Query editor for table. The User environment Query editor is shown here:



For more information about using the Query Editor, refer to the *4th Dimension User Reference* manual.

The user builds the query, then clicks the Query button or chooses Query in selection to perform the query. If the query is performed without interruption, the OK variable is set to 1. If the user clicks Cancel, the QUERY terminates with no query actually performed, and sets the OK variable to 0 (zero).

Examples

1. The following line displays the Query editor for the [Products] table:

```
⇒	QUERY([Products])
```

2. The following line displays the Query editor for the default table (if it has been set)

```
⇒	QUERY
```

If you specify the queryArgument parameter, the standard Query editor is not presented and the query is defined programmatically. For simple queries (search on only one field) you call QUERY once with queryArgument. For multiple queries (search on multiple fields or with multiple conditions), you call QUERY as many times as necessary with queryArgument, and you specify the optional * parameter, except for the last QUERY call, which starts the actual query operation. The queryArgument parameter is described further in this section.

Examples

3. The following line looks for the [People] whose name starts with an “a”:

```
⇒	QUERY([People];[People]Last name="a@")
```

4. The following line looks for the [People] whose name starts with “a” or “b”:

```
⇒	QUERY([People];[People]Name="a@";*) ` * indicates that there are further search criteria
⇒	QUERY([People]; | ;[People]Name="b@") ` No * ends the query definition and starts the
actual query operation
```

Specifying the Query Argument

• The queryArgument parameter uses the following syntax:

{ conjunction ; } field comparator value

• The conjunction is used to join QUERY calls when defining multiple queries. The conjunctions available are the same as those in the User environment Query editor:

Conjunction	Symbol to use with QUERY
AND	&
OR	|
Except	#

The conjunction is optional and not used for the first QUERY call of a multiple query, or if the query is a simple query.

• The field is the field to query. The field may belong to another table if it belongs to a One table related to table with an automatic relation. The table on which QUERY is applied to must be the Many table.

• The comparator is the comparison that is made between field and value. The comparator is one of the symbols shown here:

Comparison	**Symbol to use with QUERY**
Equal to	=
Not equal to	#
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

• The value is the data against which field will be compared. The value can be any expression that evaluates to the same data type as field. The value is evaluated once, at the beginning of the query. The value is not evaluated for each record. To query for a string contained in a string (a “contains” query), use the wildcard symbol (@) in value.

Here are the rules for building multiple queries:
• The first query argument must not contain a conjunction.
• Each successive query argument must begin with a conjunction.
• The first query and every other query, except the last, must use the * parameter.
• To perform the query, do not specify the * parameter in the last QUERY command. Alternatively, you may execute the QUERY command without any parameters other than the table (the Query editor is not presented; instead, the multiple query you just defined is performed).

Note: Each table maintains its own current built query. This means that you can create multiple built queries simultaneously, one for each table. You must use the table parameter or set the default table to specify which table to use.

No matter the way a query has been defined:

• If the actual query operation is going to take some time to be performed, 4th Dimension automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the commands MESSAGES ON and MESSAGES OFF. If the progress thermometer is displayed, the user can click on the Stop button to interrupt the query. If the query is completed, OK is set to 1. Otherwise, if the query is interrupted, OK is set to 0 (zero).

• If any indexed fields are specified, the query is optimized every time that it is possible (indexed fields are searched first) resulting in a query that takes the least amount of time possible.

Examples

5. The following command finds the records for all the people named Smith:

```
⇒	QUERY([People];[People]Last Name="Smith")
```

Note: If the Last Name field were indexed, the QUERY command would automatically use the index for a fast query.

Reminder: This query will find records like “Smith”, “smith”,“SMITH”, etc. To distinguish lowercase from uppercase, perform additional queries using the ASCII codes.

6. The following example finds the records for all people named John Smith. The Last Name field is indexed. The First Name field is not indexed.

```
⇒	QUERY ([People]; [People]Last Name = "smith"; *)   ` Find every person named Smith
⇒	QUERY ([People]; &; [People]First Name = "john")   ` with John as first name
```

When the query is performed, it quickly does an indexed search on Last Name and reduces the selection of records to those of people named Smith. The query then sequentially searches on First Name in this selection of records.

7. The following example finds the records of people named Smith or Jones. The Last Name field is indexed.

```
⇒	QUERY ([People]; [People]Last Name="smith"; *)  ` Find every person named Smith...
⇒	QUERY ([People]; | ; [People]Last Name="jones") ` ...or Jones
```

The QUERY command uses the Last Name index for both queries. The two queries are performed, and their results put into internal sets that are eventually combined using a union.

8. The following example finds the records for people who do not have a company name. It does this by finding entries with empty fields (the empty string).

```
⇒	QUERY ([People]; [People]Company="") ` Find every person with no company
```

9. The following example finds the record for every person whose last name is Smith and who works for a company based in New York. The second query uses a field from another table. This query can be done because the [People] table is related to the [Company] table with a many to one relation:

```
	   ` Find every person named Smith...
⇒	QUERY ([People]; [People]Last Name = "smith"; *)
	   ` ... and who works for a company based in NY
⇒	QUERY ([People]; & ; [Company]State = "NY")
```

10. The following example finds the record for every person whose name falls between A (included) and M (included):

```
⇒	QUERY ([People]; [People]Name < "n")  ` Find every person from A to M
```

11. The following example finds the records for all the people living in the San Francisco or Los Angeles areas (ZIP codes beginning with 94 or 90):

```
⇒	QUERY ([People]; [People]ZIP Code = "94@"; *)   ` Find every person in the SF...
⇒	QUERY ([People]; | ; [People]ZIP Code = "90@")   ` ...or Los Angeles areas
```

12. The following example queries an indexed subfield. The query returns a selection of parent records (records for the [People] table). It does not return a selection of subrecords. The result of the query would be the selection of records for all the people who have a child named Sabra:

```
⇒	QUERY ([People]; [People]Children'Name = "Sabra") ` Find people with child named
Sabra
```

13. The following example finds the record that matches the invoice number entered in the request dialog box:

```
	vFind:=Request("Find invoice number:")  ` Get an invoice number from the user
	If (OK = 1) ` If the user pressed OK
		` Find the invoice number that matches vFind
⇒	   QUERY ([Invoice]; [Invoice]Number = vFind)
	End if
```

14. The following example finds the records for the invoices entered in 1996. It does this by finding all records entered after 12/31/95 and before 1/1/97:

```
⇒	QUERY ([Invoice]; [Invoice]In Date > !12/31/95!; *)   ` Find invoices after 12/31/95...
⇒	QUERY ([Invoice]; &; [Invoice]In Date < !1/1/97!)   ` and before 1/1/97
```

15. The following example finds the record for each employee whose salary is between $10,000 and $50,000. The query includes the employees who make $10,000, but excludes those who make $50,000:

```
	   ` Find employees who make between...
⇒	QUERY ([Employee]; [Employee]Salary >= 10000; *)
⇒	QUERY ([Employee]; & ; [Employee]Salary < 50000)  ` ...$10,000 and $50,000
```

16. The following example finds the records for the employees in the marketing department who have salaries over $20,000. The Salary field is queried first because it is indexed. Notice that the second query uses a field from another table. It can do this because the [Dept] table is related to the [Employee] table with an automatic many to one relation. Although the [Dept]Name field is indexed, this is not an indexed query because the relation must be activated sequentially for each record in the [Employee] table:

```
          ` Find employees with salaries over $20,000 and...
⇒     QUERY ([Employee]; [Employee]Salary > 20000; *)
          ` ...who are in the marketing ` department
⇒     QUERY ([Employee]; & [Dept]Name = "marketing")
```

17. The following example queries for information that was entered into the variable myVar.

```
⇒     QUERY ([Laws]; [Laws]Text = myVar)  ` Find all laws that match myVar
```

The query could have many different results, depending on the value of myVar. The query will also be performed differently. For example:
• If myVar equals "Copyright@", the selection contains all laws with texts beginning with Copyright.
• If myVar equals "@Copyright@", the selection contains all laws with texts containing at least one occurrence of Copyright.

See Also

QUERY SELECTION.

QUERY SELECTION

QUERY SELECTION ({table}{; queryArgument}{; *})

Parameter	Type		Description
table	Table	→	Table for which to return a selection of records, or Default table, if omitted
queryArgument		→	Query argument
*		→	Continue query flag

Description

QUERY SELECTION looks for records in table. QUERY SELECTION command changes the current selection of table for the current process and makes the first record of the new selection the current record.

QUERY SELECTION works and performs the same actions as QUERY. The difference between the two commands is the scope of the query:
• QUERY looks for records among all the records in the table.
• QUERY SELECTION looks for records among the records currently selected in the table.

For more information, see the description of the command QUERY.

Example

This example illustrates the difference between QUERY and QUERY SELECTION. Here are two queries:

```
    ` Find ALL companies located in New York City
QUERY ([Company]; [Company]City="New York City")
    ` Find ALL companies doing Stock Exchange business
    ` (no matter where they are located)
QUERY ([Company]; [Company]Type Business="Stock Exchange")
```

Note that the second QUERY simply “ignores” the result of the first one. Compare this with:

```
    ` Find ALL companies located in New York City
QUERY ([Company]; [Company]City="New York City")
    ` Find companies doing Stock Exchange business
    ` and that are located in New York City
QUERY SELECTION ([Company]; [Company]Type Business="Stock Exchange")
```

QUERY SELECTION looks only among the selected records, therefore, in this example, among the companies located in New York City.

QUERY BY FORMULA

QUERY BY FORMULA ({table}{; }{queryFormula})

Parameter	Type		Description
table	Table	→	Table for which to return a selection of records, or Default table, if omitted
queryFormula	Boolean	→	Query formula

Description

QUERY BY FORMULA looks for records in table. QUERY BY FORMULA changes the current selection of table for the current process and makes the first record of the new selection the current record.

QUERY BY FORMULA and QUERY SELECTION BY FORMULA work exactly the same way, except that QUERY BY FORMULA queries every record in the entire table and QUERY SELECTION BY FORMULA queries only the records in the current selection.

Both commands apply queryFormula to each record in the table or selection. The queryFormula is a Boolean expression that must evaluate to either TRUE or FALSE. If queryFormula evaluates as TRUE, the record is included in the new selection.

The queryFormula may be simple, perhaps comparing a field to a value; or it may be complex, perhaps performing a calculation or even evaluating information in a related table. The queryFormula can be a 4th Dimension function (command), or a function (method) or expression you have created. You can use wildcards in queryFormula when working with Alpha or text fields.

When the query is complete, the first record of the new selection is loaded from disk and made the current record.

These commands always perform a sequential search, not an indexed search. QUERY BY FORMULA and QUERY SELECTION BY FORMULA are slower than QUERY when used on indexed fields. The query time is proportional to the number of records in the table or selection.

4D Server: The server does not execute the query formula. Each record is sent to the local workstation and the query formula is evaluated on the workstation. This makes the command less efficient with 4D Server than the QUERY command.

Examples

1. The following example finds the records for all invoices that were entered in December of any year. It does this by applying the Month of function to each record. This query could not be performed any other way without creating a separate field for the month:

```
            ` Find the invoices entered in December
⇒       QUERY BY FORMULA ([Invoice]; Month of ([Invoice]Entered) = 12)
```

2. The following example finds records for all the people who have names with more than ten characters:

```
            ` Find names longer than ten characters
⇒       QUERY BY FORMULA ([People]; Length ([People]Name)>10)
```

See Also

QUERY, QUERY SELECTION, QUERY SELECTION BY FORMULA.

QUERY SELECTION BY FORMULA

QUERY SELECTION BY FORMULA ({table}{; }{queryFormula})

Parameter	Type		Description
table	Table	→	Table for which to return a selection of records, or Default table, if omitted
queryFormula	Boolean	→	Query formula

Description

QUERY SELECTION BY FORMULA looks for records in table. QUERY SELECTION BY FORMULA changes the current selection of table for the current process and makes the first record of the new selection the current record.

QUERY SELECTION BY FORMULA performs the same actions as QUERY BY FORMULA. The difference between the two commands is the scope of the query:
• QUERY BY FORMULA looks for records among all the records in the table.
• QUERY SELECTION BY FORMULA looks for records among the records currently selected in the table.

For more information, see the description of the command QUERY BY FORMULA.

See Also

QUERY, QUERY BY FORMULA, QUERY SELECTION.

SET QUERY DESTINATION

version 6.0

SET QUERY DESTINATION (destinationType{; destinationObject})

Parameter	Type		Description
destinationType	Number	→	0 current selection 1 set 2 named selection 3 variable
destinationObject	String | Variable	→	Name of the set, or Name of the named selection, or Variable

Description

SET QUERY DESTINATION enables you to tell 4th Dimension where to put the result of any subsequent query for the current process.

You specify the type of the destination in the parameter destinationType.4th Dimension provides the following predefined constants:

Constant	Type	Value
Into current selection	Long Integer	0
Into set	Long Integer	1
Into named selection	Long Integer	2
Into variable	Long Integer	3

You specify the destination of the query itself in the optional destinationObject parameter according to the following table:

destinationType parameter	destinationObject parameter
0 (current selection)	You omit the parameter.
1 (set)	You pass the name of a set (existing or to be created)
2 (named selection)	You pass the named of a named selection (existing or to be created)
3 (variable)	You pass a numeric variable (existing or to be created)

With:

SET QUERY DESTINATION(Into current selection)

The records found by any subsequent query will end up in a new current selection for the table involved by the query.

With:

SET QUERY DESTINATION(Into set;"mySet")

The records found by any subsequent query will end up in the set "mySet". The current selection and the current record for the table involved by the query are left unchanged.

With:

SET QUERY DESTINATION(Into named selection;"myNamedSel")

The records found by any subsequent query will end up in the named selection "myNamedSel".The current selection and the current record for the table involved by the query are left unchanged.

With:

SET QUERY DESTINATION(Into variable;$vlResult)

The number of records found by any subsequent query will end up in the variable $vlResult.The current selection and the current record for the table involved by the query are left unchanged.

Warning: SET QUERY DESTINATION affects all subsequent queries made within the current process. REMEMBER to always counterbalance a call to SET QUERY DESTINATION (where destinationType#0) with a call to SET QUERY DESTINATION(0) in order to restore normal query mode.

SET QUERY DESTINATION changes the behavior of the query commands only:

• QUERY
• QUERY SELECTION
• QUERY BY EXAMPLE
• QUERY BY FORMULA
• QUERY SELECTION BY FORMULA

On the other hand, SET QUERY DESTINATION does not affect other commands that may change the current selection of a table such as ALL RECORDS, RELATE MANY and so on.

Examples

1. You create a form that will display the records from a [Phone Book] table. You create a Tab Control named asRolodex (with the 26 letters of the alphabet) and a subform displaying the [Phone Book] records. Choosing one Tab from the Tab Control displays the records whose names start with the corresponding letter.

In your application, the [Phone Book] table contains a set of quite static data, so you do not want to (or need to) perform a query each time you select a Tab. In this way, you can save precious database engine time.

To do so, you can redirect your queries into named selections that you reuse as needed. You write the object method of the Tab Control asRolodex as follows:

```
           ` asRolodex object method
       Case of
           : (Form event=On Load)
                 ` Before the form appears on the screen,
                 ` initialize the rolodex and a array of booleans that
                 ` will tell us if a query for the corresponding letter
                 ` has been performed or not
               ARRAY STRING(1;asRolodex;26)
               ARRAY BOOLEAN(abQueryDone;26)
               For ($vlElem;1;26)
                   asRolodex{$vlElem}:=Char(64+$vlElem)
                   abQueryDone{$vlElem}:=False
               End for

           : (Form event=On Clicked)
                 ` When a click on the Tab control occurs,
                 ` check whether the corresponding query
                 ` has been performed or not
               If (Not(abQueryDone{asRolodex}))
                     ` If not, redirect the next query(ies) toward a named selection
⇒                  SET QUERY DESTINATION(Into named selection;
                                                         "Rolodex"+asRolodex{asRolodex})
                     ` Perform the query
                   QUERY([Phone Book];[Phone Book]Last name=asRolodex{asRolodex}+"@")
                     ` Restore normal query mode
⇒                  SET QUERY DESTINATION(Into current selection)
                     ` Next time we choose that letter, we won't perform the query again
                   abQueryDone{asRolodex}:=True
               End if
                   ` Use the named selection for displaying the records
                   ` corresponding to the choosen letter
               USE NAMED SELECTION("Rolodex"+asRolodex{asRolodex})
```

```
        : (Form event=On Unload)
              ` After the form disappeared from the screen
              ` Clear the named selections we created
           For ($vlElem;1;26)
              If(abQueryDone{$vlElem})
                 CLEAR NAMED SELECTION("Rolodex"+asRolodex{$vlElem})
              End if
           End for
              ` Clear the two arrays we no longer need
           CLEAR VARIABLE(asRolodex)
           CLEAR VARIABLE(abQueryDone)
     End case
```

2. The Unique values project method in this example allows you to verify the uniqueness of the values for any number of fields in a table. The current record can be an existing or a newly created record.

```
        ` Unique values project method
        ` Unique values ( Pointer ; Pointer { ; Pointer... } ) -> Boolean
        ` Unique values ( ->Table ; ->Field { ; ->Field2...  } ) -> Yes or No

     C_BOOLEAN($0;$2)
     C_POINTER(${1})
     C_LONGINT($vlField;$vlNbFields;$vlFound;$vlCurrentRecord)
     $vlNbFields:=Count parameters-1
     $vlCurrentRecord:=Record number($1->)
     If ($vlNbFields>0)
        If ($vlCurrentRecord#-1)
           If ($vlCurrentRecord<0)
                 ` The current record is an unsaved new record (record number is -3)
                 ` therefore we can stop the query as soon as at least one record is found
              SET QUERY LIMIT(1)
           Else
                 ` The current record is an existing record,
                 ` therefore we can stop the query as soon as at least two records are found
              SET QUERY LIMIT(2)
           End if
              ` The query will return its result in $vlFound
              ` without changing the current record nor the current selection
⇒          SET QUERY DESTINATION(Into variable;$vlFound)
```

```
            ` Make the query according to the number of fields that are specified
        Case of
            : ($vlNbFields=1)
                QUERY($1->;$2->=$2->)
            : ($vlNbFields=2)
                QUERY($1->;$2->=$2->;*)
                QUERY($1->; & ;$3->=$3->)
            Else
                QUERY($1->;$2->=$2->;*)
                For ($vlField;2;$vlNbFields-1)
                    QUERY($1->; & ;${1+$vlField}->=${1+$vlField}->;*)
                End for
                QUERY($1->; & ;${1+$vlNbFields}->=${1+$vlNbFields}->)
        End case
⇒       SET QUERY DESTINATION(Into current selection)  ` Restore normal query mode
        SET QUERY LIMIT(0)  ` No longer limit queries
            ` Process query result
        Case of
            : ($vlFound=0)
                $0:=True  ` No duplicated values
            : ($vlFound=1)
                If ($vlCurrentRecord<0)
                        ` Found an existing record with the same values
                        ` as the unsaved new record
                    $0:=False
                Else
                    $0:=True  ` No duplicated values, just found the very same record
                End if
            : ($vlFound=2)
                $0:=False  ` Whatever the case is, the values are duplicated
        End case
    Else
        If (<>DebugOn)  ` Does not make sense, signal it if development version
            TRACE  ` WARNING! Unique values is called with NO current record
        End if
        $0:=False  ` Can't guaranty the result
    End if
Else
    If (<>DebugOn)  ` Does not make sense, signal it if development version
        TRACE  ` WARNING! Unique values is called with NO query condition
    End if
    $0:=False  ` Can't guaranty the result
End if
```

After this project method is implemented in your application, you can write:

```
          ` ...
     If (Unique values (->[Contacts];->[Contacts]Company);->[Contacts]Last name
                                                            ;->[Contacts]First name)
             ` Do appropriate actions for that record which has unique values
     Else
          ALERT("There is already a Contact with this name for this Company.")
     End if
          ` ...
```

See Also

QUERY, QUERY BY EXAMPLE, QUERY BY FORMULA, QUERY SELECTION, QUERY SELECTION BY FORMULA, SET QUERY LIMIT.

SET QUERY LIMIT

version 6.0

SET QUERY LIMIT (limit)

Parameter	Type		Description
limit	Number	→	Number of records, or 0 for no limit

Description

SET QUERY LIMIT allows you to tell 4th Dimension to stop any subsequent query for the current process as soon as at least the number of records you pass in limit has been found.

For example, if you pass limit equal to 1, any subsequent query will stop browsing an index or the data file as soon as one record that matches the query conditions has been found.

To restore queries with no limit, call SET QUERY LIMIT again with limit equal to 0.

Warning: SET QUERY LIMIT affects all the subsequent queries made within the current process. REMEMBER to always counterbalance a call to SET QUERY LIMIT(limit) (where limit>0) with a call to SET QUERY LIMIT(0) in order to restore queries with no limit.

SET QUERY LIMIT changes the behavior of the query commands:

• QUERY
• QUERY SELECTION
• QUERY BY EXAMPLE
• QUERY BY FORMULA
• QUERY SELECTION BY FORMULA

On the other hand, SET QUERY LIMIT does not affect the other commands that may change the current selection of a table, such as ALL RECORDS, RELATE MANY, and so on.

Examples

1. To perform a query corresponding to the request "...give me any ten customers whose gross sales are greater than $1 M...", you would write:

```
⇒	SET QUERY LIMIT(10)
	QUERY([Customers];[Customers]Gross sales>1000000)
⇒	SET QUERY LIMIT(0)
```

2. See the second example for the command SET QUERY DESTINATION.

ORDER BY

ORDER BY ({table}{; field}{; > or <}{; field2; > or <2; ...; fieldN; > or <N})

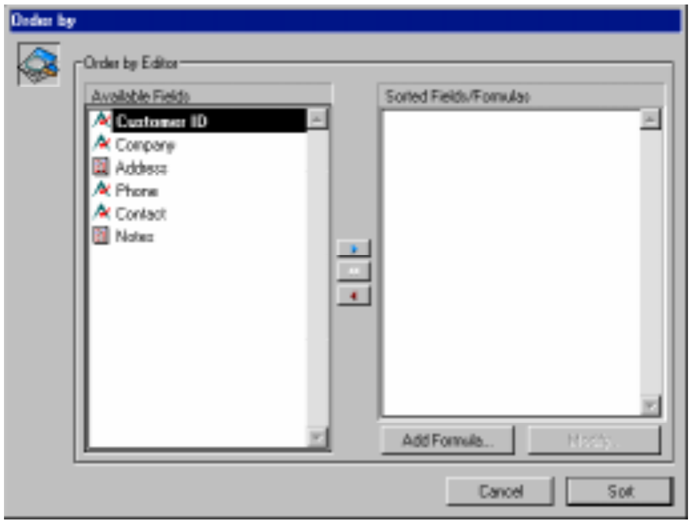
Parameter	Type		Description
table	Table	→	Table for which to order selected records, or Default table, if omitted
field	Field	→	Field on which to set the order for each level
> or <		→	Ordering direction for each level: > to order in ascending order, or < to order in descending order

Description

ORDER BY sorts (reorders) the records of the current selection of table for the current process. After the sort has been completed, the new first record of the selection becomes the current record.

If you omit the table parameter, the command applies to the default table. If no default table has been set, an error occurs.

If you do not specify the field or the > or < parameters, ORDER BY displays the Order By editor for table. The User environment's Order By editor is shown here:



For more information about using the Order By editor, refer to the *4th Dimension User Reference* manual.

The user builds the sort, then clicks the Sort button to perform the sort. If the sort is performed without interruption, the OK variable is set to 1. If the user clicks Cancel, the ORDER BY terminates with no sort actually performed, and sets the OK variable to 0 (zero).

Examples

1. The following line displays the Order By editor for the [Products] table:

```
⇒	ORDER BY([Products])
```

2. The following line displays the Order By editor for the default table (if it has been set)

```
⇒	ORDER BY
```

If you specify the field and > or < parameters, the standard Order By editor is not presented and the sort is defined programmatically. You can sort the selection on one level or on several levels. For each sort level, you specify a field in field and the sorting order in > or <. If you pass the "greater than" symbol (>), the order is ascending. If you pass the "less than" symbol (<), the order is descending.

Examples

3. The following line orders the selection of [Products] by name in ascending order:

```
⇒	ORDER BY([Products];[Products]Name;>)
```

4. The following line orders the selection of [Products] by name in descending order:

```
⇒	ORDER BY([Products];[Products]Name;<)
```

5. The following line orders the selection of [Products] by type and price in ascending order for both levels:

```
⇒	ORDER BY([Products];[Products]Type;>;[Products]Price;>)
```

6. The following line orders the selection of [Products] by type and price in descending order for both levels:

```
⇒	ORDER BY([Products];[Products]Type;<;[Products]Price;<)
```

7. The following line orders the selection of [Products] by type in ascending order and by price in descending order:

```
⇒	ORDER BY([Products];[Products]Type;>;[Products]Price;<)
```

8. The following line orders the selection of [Products] by type in descending order and by price in ascending order:

```
⇒	ORDER BY([Products];[Products]Type;<;[Products]Price;>)
```

If you omit the sorting order parameter > or <, ascending order is the default.

Example

9. The following line orders the selection of [Products] by name in ascending order:

```
⇒	ORDER BY([Products];[Products]Name)
```

If only one field is specified (one level sort) and it is indexed, the index is used for the order. If the field is not indexed or if there is more than one field, the order is performed sequentially. The field may belong to the (selection's) table being reordered or to a One table related to table with an automatic relation. (Remember, the table to which ORDER BY is applied must be the Many table.) In this case, the sort is always sequential.

Examples

10. The following line performs an indexed sort if [Products]Name is indexed:

```
⇒	ORDER BY([Products];[Products]Name;>)
```

11. The following line performs a sequential sort, whether or not the fields are indexed:

```
⇒	ORDER BY([Products];[Products]Type;>;[Products]Price;>)
```

12. The following line performs a sequential sort using a related field:

```
⇒	ORDER BY([Invoices];[Companies]Name;>) ` Invoices are sorted alphabetically on the
Company name field
```

No matter what way a sort has been defined, if the actual sort operation is going to take some time to be performed, 4th Dimension automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the commands MESSAGES ON and MESSAGES OFF. If the progress thermometer is displayed, the user can click the Stop button to interrupt the sort. If the sort is completed, OK is set to 1. Otherwise, if the sort is interrupted, OK is set to 0 (zero).

See Also

ORDER BY FORMULA.

ORDER BY FORMULA

ORDER BY FORMULA ({table}{; expression}{; > or <}{; expression2; > or <2; ...; expressionN; > or <N})

Parameter	Type		Description
table	Table	→	Table for which to order selected records, or Default table, if omitted
expression		→	Expression on which to set the order for each level (can be of type Alphanumeric, Real, Integer, Long Integer, Date, Time or Boolean)
> or <		→	Ordering direction for each level: > to order in ascending order, or < to order in descending order

Description

ORDER BY FORMULA sorts (reorders) the records of the current selection of table for the current process. After the sort has been completed, the new first record of the selection becomes the current record.

Note that you must specify table. You cannot use a default table.

You can sort the selection on one level or on several levels. For each sort level, you specify a expression in expression and the sorting order in > or <. If you pass the “greater than” symbol (>), the order is ascending. If you pass the “less than” symbol (<), the order is descending. If you do not specify the sorting order, ascending order is the default.

The parameter expression can be of type Alphanumeric, Real, Integer, Long Integer, Date, Time or Boolean.

No matter what way a sort has been defined, if the actual sort operation is going to take some time to be performed, 4th Dimension automatically displays a message containing a progress thermometer. These messages can be turned on and off by using the commands MESSAGES ON and MESSAGES OFF. If the progress thermometer is displayed, the user can click the Stop button to interrupt the sort. If the sort is completed, OK is set to 1. Otherwise, if the sort is interrupted, OK is set to 0 (zero).

4D Server: Since expression cannot be interpreted by 4D Server, each record is sent to the local workstation; the order formula is evaluated on the workstation. This will make the order inefficient. Use the ORDER BY command whenever possible.

Unlike ORDER BY, ORDER BY FORMULA always performs a sequential sort.

Examples

The following example orders the records of the [People] table in descending order, based on the length of each person's last name. The record for the person with the longest last name will be first in the current selection:

```
⇒	ORDER BY FORMULA ([People]; Length([People]Last Name);<)
```

See Also

ORDER BY.

37 Record Locking

Record Locking

version 3

4th Dimension and 4D Server/4D Client automatically manage databases by preventing multi-user or multi-process conflicts. Two users or two processes cannot modify the same record or object at the same time. However, the second user or process can have read-only access to the record or object at the same time.

There are several reasons for using the multi-user commands:
• Modifying records by using the language.
• Using a custom user interface for multi-user operations.
• Saving related modifications inside a transaction.

There are three important concepts to be aware of when using commands in a multi-processing database:
• Each table is in either a read-only or a read/write state.
• Records become locked when they are loaded and unlocked when they are unloaded.
• A locked record cannot be modified.

As a convention in the following sections, the person performing an operation on the multi-user database is referred to as the **local user**. Other people using the database are referred to as the **other users**. The discussion is from the perspective of the local user. Also, from a multi-process perspective, the process executing an operation on the database is the **current process**. Any other executing process is referred to as **other processes**. The discussion is from the point of view of the current process.

Locked Records

A locked record cannot be modified by the local user or the current process. A locked record can be loaded, but cannot be modified. A record is locked when one of the other users or processes has successfully loaded the record for modification. Only the user who is modifying the record sees that record as unlocked. All other users and processes see the record as locked, and therefore unavailable for modification. A table must be in a read/write state for a record to be loaded unlocked.

Read-Only and Read/Write States

Each table in a database is in either a read/write or a read-only state for each user and process of the database. **Read-only** means that records for the table can be loaded but not modified. **Read/write** means that records for the table can be loaded and modified if no other user has locked the record first.

Note that if you change the status of a table, the change takes effect for the next record loaded. If there is a record currently loaded when you change the table's status, that record is not affected by the status change.

Read-Only State

When a table is read-only and a record is loaded, the record is always locked. In other words, the record can be displayed, printed, and otherwise used, but it cannot be modified.

Note that read-only status applies only to editing existing records. Read-only status does not affect the creation of new records. You can add records to a read-only table using CREATE RECORD and ADD RECORD or the New Record menu command from the User environment's Enter menu.

4th Dimension automatically sets a table to read-only for commands that do not require write access to records. These commands are:

- DISPLAY SELECTION
- DISTINCT VALUES
- EXPORT DIF
- EXPORT SYLK
- EXPORT TEXT
- GRAPH TABLE
- PRINT SELECTION
- PRINT LABELS
- REPORT
- SELECTION TO ARRAY
- SUBSELECTION TO ARRAY

Before executing any of these commands, 4th Dimension saves the current state of the table (read-only or read/write) for the current process. After the command has executed, the state is restored.

Read/Write State

When a table is read/write and a record is loaded, the record will become unlocked if no other user has locked the record first. If the record is locked by another user, the record is loaded as a locked record that cannot be modified by the local user.

A table must be set to read/write and the record loaded for it to become unlocked and thus modifiable.

If a user loads a record from a table in read/write mode, no other users can load that record for modification. However, other users can add records to the table, either through the CREATE RECORD or ADD RECORD commands or manually in the User environment.

Read/write is the default state for all tables when a database is opened and a new process is started.

Changing the Status of a Table

You can use the READ ONLY and READ WRITE commands to change the state of a table. If you want to change the state of a table in order to make a record read-only or read/write, you must execute the command before the record is loaded. Any record that is already loaded is not affected by the
READ ONLY and READ WRITE commands.

Each process has its own state (read-only or read/write) for each table in the database.

Loading, Modifying and Unloading Records

Before the local user can modify a record, the table must be in the read/write state and the record must be loaded and unlocked.

Any of the commands that loads a current record (if there is one) — such as NEXT RECORD, QUERY, ORDER BY, RELATE ONE, etc. — sets the record as locked or unlocked. The record is loaded according to the current state of its table (read-only or read/write) and its availability. A record may also be loaded for a related table by any of the commands that cause an automatic relation to be established.

If a table is in the read-only state, then a record loaded from that table is locked. A locked record cannot be saved or deleted. Read-only is the preferred state, because it allows other users to load, modify, and then save the record.

If a table is in the read/write state, then a record that is loaded from that table is unlocked only if no other users have locked the record first. An unlocked record can be modified and saved. A table should be put into the read/write state before a record needs to be loaded, modified, and then saved.

If the record is to be modified, you use the Locked function to test whether or not a record is locked by another user. If a record is locked (Locked returns True), load the record with the LOAD RECORD command and again test whether or not the record is locked. This sequence must be continued until the record becomes unlocked (Locked returns False).

When modifications to be made to a record are finished, the record must be released (and therefore unlocked for the other users) with UNLOAD RECORD. If a record is not unloaded, it will remain locked for all other users until a different current record is selected. Changing the current record of a table automatically unlocks the previous current record. You need to explicitly call UNLOAD RECORD if you do not change the current record. This discussion applies to existing records. When a new record is created, it can be saved regardless of the state of the table to which it belongs. Use the LOCKED ATTRIBUTES command to see which user and/or process have locked a record.

Loops to Load Unlocked Records

The following example shows the simplest loop with which to load an unlocked record:

```
READ WRITE ([Customers])   ` Set the table's state to read/write
Repeat   ` Loop until the record is unlocked
   LOAD RECORD ([Customers])   ` Load record and set locked status
Until (Not (Locked([Customers])))
   ` Do something to the record here
READ ONLY ([Customers])   ` Set the table's state to read-only
```

The loop continues until the record is unlocked.

A loop like this is used only if the record is unlikely to be locked by anyone else, since the user would have to wait for the loop to terminate. Thus, it is unlikely that the loop would be used as is unless the record could only be modified by means of a method.

The following example uses the previous loop to load an unlocked record and modify the record:

```
READ WRITE([Inventory])
Repeat   ` Loop until the record is unlocked
   LOAD RECORD([Inventory])   ` Load record and set it to locked
Until (Not (Locked([Inventory])))
[Inventory]Part Qty := [Inventory]Part Qty – 1   ` Modify the record
SAVE RECORD ([Inventory])   ` Save the record
UNLOAD RECORD ([Inventory])   ` Let other users modfiy it
READ ONLY([Inventory])
```

The MODIFY RECORD command automatically notifies the user if a record is locked, and prevents the record from being modified. The following example avoids this automatic notification by first testing the record with the Locked function. If the record is locked, the user can cancel.

This example efficiently checks to see if the current record is locked for the table [Commands]. If it is locked, the process is delayed by the procedure for one second. This technique can be used both in a multi-user or multi-process situation:

```
Repeat
   READ ONLY([Commands]) ` You do not need read/write right now
   QUERY([Commands])
      ` If the search was completed and some records were returned
   If ((OK=1) & (Records in selection([Commands])>0))
      READ WRITE([Commands]) ` Set the table to read/write state
      LOAD RECORD([Commands])
```

```
        While (Locked([Commands]) & (OK=1)) `If the record is locked,
              ` loop until the record is unlocked
              ` Who is the record locked by?
           LOCKED ATTRIBUTES([Commands];$Process;$User;$Machine;$Name)
           If ($Process=-1) ` Has the record been deleted?
              ALERT("The record has been deleted in the meantime.")
              OK:=0
           Else
              If ($User="") ` Are you in single-user mode
                 $User:="you"
              End if
              CONFIRM("The record is already used by "+$User+" in the "
                                                      +$Name+" Process.")
              If (OK=1) ` If you want to wait for a few seconds
                 DELAY PROCESS(Current process;120) ` Wait for a few seconds
                 LOAD RECORD([Commands])` Try to load the record
              End if
           End if
        End while
        If (OK=1) ` The record is unlocked
           MODIFY RECORD([Commands]) ` You can modify the record
           UNLOAD RECORD([Commands])
        End if
        READ ONLY([Commands]) ` Switch back to read-only
        OK:=1
     End if
  Until (OK=0)
```

Using Commands in Multi-user or Multi-process Environment

A number of commands in the language perform specific actions when they encounter a locked record. They behave normally if they do not encounter a locked record.

Here is a list of these commands and their actions when a locked record is encountered.

• MODIFY RECORD: Displays a dialog box stating that the record is in use. The record is not displayed, therefore the user cannot modify the record. In the User environment, the record is shown in read-only state.
• MODIFY SELECTION: Behaves normally except when the user double-clicks a record to modify it. MODIFY SELECTION displays dialog box stating that the record is in use and then allows read-only access to the record.
• APPLY TO SELECTION: Loads a locked record, but does not modify it. APPLY TO SELECTION can be used to read information from the table without special care. If the command encounters a locked record, the record is put into the LockedSet system set.

• DELETE SELECTION: Does not delete any locked records; it skips them. If the command encounters a locked record, the record is put into the LockedSet system set.
• DELETE RECORD: This command is ignored if the record is locked. No error is returned. You must test that the record is unlocked before executing this command.
• SAVE RECORD: This command is ignored if the record is locked. No error is returned. You must test that the record is unlocked before executing this command.
• ARRAY TO SELECTION: Does not save any locked records. If the command encounters a locked record, the record is put into the LockedSet system set.
• GOTO RECORD: Records in a multi-user/multi-process database may be deleted and added by other users, therefore the record numbers may change. Use caution when directly referencing a record by number in a multi-user database.
• Sets: Take special care with sets, as the information that the set was based on may be changed by another user or process.

See Also

LOAD RECORD, Locked, LOCKED ATTRIBUTES, Methods, READ ONLY, Read only state, READ WRITE, UNLOAD RECORD, Variables.

READ WRITE

READ WRITE {(table | *)}

Parameter	Type		Description
table | *	Table	→	Table for which to set read-write state, or * for all the tables, or Default table, if omitted

Description

READ WRITE changes the state of table to read/write for the process in which it is called. If the optional * parameter is specified, all tables are changed to read/write state.

After a call to READ WRITE, when a record is loaded, the record is unlocked if no other user has locked the record. This command does not change the status of the currently loaded record, only that of subsequently loaded records.

The default state for all tables is read/write.

Use READ WRITE when you must modify a record and save the changes. Also use READ WRITE when you must lock a record for other users, even if you are not making any changes. Setting a table to read/write mode prevents other users from editing that table. However, other users can create new records.

Note: This command is not retroactive. A record is loaded according to the table's read/write status at the time of loading. To load a record from a read-only table in read/write mode, you must first change the table state to read/write.

See Also

READ ONLY, Read only state, Record Locking.

READ ONLY

READ ONLY {(table | *)}

Parameter	Type		Description
table | *	Table	→	Table for which to set read-only state, or * for all the tables, or Default table, if omitted

Description

READ ONLY changes the state of tTable to read-only for the process in which it is called. All subsequent records that are loaded are locked, and you cannot make any changes made to them. If the optional * parameter is specified, all tables are changed to read-only state.

Use READ ONLY when you do not need to modify the record or records.

Note: This command is not retroactive. A record is loaded according to the table's read/write status at the time of loading. To load a record from a read/write table in read-only mode, you must first change the table state to read-only.

See Also

Read only state, READ WRITE, Record Locking.

Read only state

version 3

Read only state {(table)} → Boolean

Parameter	Type		Description
table	Table	→	Table for which to test read-only state, or Default table, if omitted
Function result	Boolean	←	Access to table is read-only (TRUE), or Access to table is read-write (FALSE)

Description

This function is used to test whether or not the state of table is read-only for the process in which it is called. Read only state returns TRUE if the state of table is read-only. Read only state returns FASLE if the state of table is read/ write.

Example

The following example tests the state of an [Invoice] table. If the state of the [Invoice] table is read-only, it is set to read/write, and then the current record is reloaded.

```
⇒   If (Read only state([Invoices]))
       READ WRITE([Invoices])
       LOAD RECORD([Invoices])
    End if
```

Note: The invoice record is reloaded to allow the user to modify it. A record that was previously loaded in a read-only state will remain locked until it is reloaded in a read/write state.

See Also

READ ONLY, READ WRITE, Record Locking.

LOAD RECORD

LOAD RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to load record, or Default table, if omitted

Description

LOAD RECORD loads the current record of table. If there is no current record, LOAD RECORD has no effect.

You can then use the Locked function to determine whether you can modify the record:

• If the table is in read-only state, the Locked function returns TRUE, and you cannot modify the record.
• If the table is in read/write state but the record was already locked, the record will be read-only, and you cannot modify the record.
• If the table is in read/write state and the record is not locked, you can modify the record in the current process. The Locked function returns TRUE for all other users and processes.

Usually, you do not need to use the LOAD RECORD command, because commands like QUERY, NEXT RECORD, PREVIOUS RECORD, etc., automatically load the current record.

In multi-user and multi-process environments, when you need to modify an existing record, you must access the table (to which the record belongs) in read/write mode. If a record is locked and not loaded, LOAD RECORD allows you to attempt to load the record again at a later time. By using LOAD RECORD in a loop, you can wait until the record becomes available in read/write mode.

See Also

Locked, Record Locking, UNLOAD RECORD.

UNLOAD RECORD

UNLOAD RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to unload record, or Default table, if omitted

Description

UNLOAD RECORD unloads the current record of table.

If the record is unlocked for the local user (locked for the other users), UNLOAD RECORD unlocks the record for the other users.

Although UNLOAD RECORD unloads it from memory, the record remains the current record. When another record is made the current record, the previous current record is automatically unloaded and therefore unlocked for other users. Always execute this command when you have finished modifying a record and want to make it available to other users, while retaining the record as your current record.

If a record has a large amount of data, picture fields, or external documents (such as 4D Write or 4D Draw documents), you may not want to keep the current record in memory unless you need to modify it. In this case, use the UNLOAD RECORD command to keep the current record without having it in memory. You free the memory occupied by the record, but you do not have access to its field values. If you later need access to the values of the record, use the LOAD RECORD command.

See Also

LOAD RECORD, Record Locking.

Locked

version 3

Locked {(table)} → Boolean

Parameter	Type		Description
table	Table	→	Table to check for locked current record, or Default table, if omitted
Function result	Boolean	←	Record is locked (TRUE), or Record is unlocked (FALSE)

Description

Locked tests whether or not the current record of table is locked. Use this function to find out whether or not the record is locked; then take appropriate action, such as giving the user the choice of waiting for the record to be free or skipping the operation.

If Locked returns TRUE, then the record is locked by another user or process and cannot be saved. In this case, use LOAD RECORD to reload the record until Locked returns FALSE.

If Locked returns FALSE, then the record is unlocked, meaning that the record is locked for all other users. Only the local user or current process can modify and save the record. A table must be in read/write state in order for you to modify the record.

If you try to load a record that has been deleted, Locked continues to return TRUE. To avoid waiting for a record that does not exist anymore, use the LOCKED ATTRIBUTES command. If the record has been deleted, the LOCKED ATTRIBUTES command returns -1 in the process parameter.

During transaction processing, LOAD RECORD and Locked are often used to test record availability. If a record is locked, it is common to cancel the transaction.

See Also

LOAD RECORD, LOCKED ATTRIBUTES, Record Locking.

LOCKED ATTRIBUTES

LOCKED ATTRIBUTES ({table; }process; user; machine; processName)

Parameter	Type		Description
table	Table	→	Table to check for record locked, or Default table, if omitted
process	Number	←	Process reference number
user	String	←	User name if multi-user
machine	String	←	Machine name if multi-user
processName	String	←	Process name

Description

LOCKED ATTRIBUTES returns information about the user and process that have locked a record. The process number, user name, machine name, and process name are returned in the process, user, machine, and processName variables. You can use this information in a custom dialog box to warn the user when a record is locked.

If the record is not locked, process returns 0 and user, machine, and processName return empty strings. If the record you try to load in read/write has been deleted, process returns -1 and user, machine, and processName return empty strings.

In single-user mode, this command returns values in process and processName only if a record is locked. The values returned in user and machine are empty strings.

The User parameter returned is the user name from the 4th Dimension password system, even if user name is blank. If there is no password system, “Manager” is returned.

The machine parameter returned is the owner name from the operating system file sharing setup. A name change does not take effect until you restart.

See Also

Locked, Record Locking.

38 Records

DISPLAY RECORD

version 3

DISPLAY RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table from which to display the current record, or Default table, if omitted

Description

The command DISPLAY RECORD displays the current record of table, using the current input form. The record is displayed only until an event redraws the window. Such an event might be the execution of an ADD RECORD command, returning to an input form, or returning to the menu bar. DISPLAY RECORD does nothing if there is no current record.

DISPLAY RECORD is often used to display custom progress messages. It can also be used to generate a free-running slide show.

If a form method exists, an On Load event will be generated.

WARNING: Do not call DISPLAY RECORD from within a Web connection process, because the command will be executed on the 4th Dimension Web server machine and not on the Web browser client machine.

Example

The following example displays a series of records as a slide show:

```
    ALL RECORDS([Demo])  ` Select all of the records
    INPUT FORM ([Demo]; "Display")  ` Set the form to use for display
    For ($vlRecord;1;Records in selection([Demo]))  ` Loop through all of the records
⇒       DISPLAY RECORD([Demo])  ` Display a record
        DELAY PROCESS (Current process; 180)  ` Pause for 3 seconds
        NEXT RECORD([Demo])  ` Move to the next record
    End for
```

See Also

MESSAGE.

CREATE RECORD

CREATE RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to create a new record, or Default table, if omitted

Description

CREATE RECORD creates a new empty record for table, but does not display the new record. Use ADD RECORD to create a new record and display it for data entry.

CREATE RECORD is used instead of ADD RECORD when data for the record is assigned with the language. The new record becomes the current record and the current selection (a one-record current selection).

The record exists in memory only until a SAVE RECORD command is executed for the table. If the current record is changed (for example, by a query) before the record is saved, the new record is lost.

Example

The following example archives records that are over 30 days old. It does does this by creating new records in an archival table. When the archiving is finished, the records that were archived are deleted from the [Accounts] table:

```
        ` Find records more than 30 days old
    QUERY ([Accounts]; [Accounts]Entered < (Current date – 30))
    For ($vlRecord;1; Records in selection([Accounts]))  ` Loop once for each record
⇒       CREATE RECORD ([Archive])  ` Create a new archive record
        [Archive]Number:=[Account]Number  ` Copy fields to the archive record
        [Archive]Entered:=[Account]Entered
        [Archive]Amount:=[Account]Amount
        SAVE RECORD([Archive])  ` Save the archive record
        NEXT RECORD([Accounts])  ` Move to the next account record
    End for
    DELETE SELECTION([Accounts])  ` Delete the account records
```

See Also

SAVE RECORD.

DUPLICATE RECORD

DUPLICATE RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to duplicate the current record, or Default table, if omitted

Description

DUPLICATE RECORD creates a new record for table that is a duplicate of the current record. The new record becomes the current record. If there is no current record, then DUPLICATE RECORD does nothing. You must use SAVE RECORD to save the new record.

DUPLICATE RECORD can be executed during data entry. This allows you to create a clone of the currently displayed record. Remember that you must first execute SAVE RECORD in order to save any changes made to the original record.

See Also

SAVE RECORD.

Modified record

version 3

Modified record {(table)} → Boolean

Parameter	Type		Description
table	Table	→	Table to test if current record has been modified, or Default table, if omitted
Function result	Boolean	←	Record has been modified (True), or Record has not been modified (False)

Description

Modified record returns True if the current record of table has been modified but not saved; otherwise it returns False. This function allows the designer to quickly test whether or not the record needs to be saved. It is especially valuable in input forms to check whether or not to save the current record before proceeding to the next one. This function always returns TRUE for a new record.

Example

The following example shows a typical use for Modified record:

```
⇒      If (Modified record ([Customers]))
          SAVE RECORD ([Customers])
       End if
```

See Also

Modified, Old, SAVE RECORD.

SAVE RECORD

SAVE RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to save the current record, or Default table, if omitted

Description

SAVE RECORD saves the current record of table in the current process. If there is no current record, then SAVE RECORD is ignored.

You use SAVE RECORD to save a record that you created or modified with code. A record that has been modified and validated by the user in a form does not need to be saved with SAVE RECORD. A record that has been modified by the user in a form, but has been canceled, can still be saved with SAVE RECORD.

Here are some cases where SAVE RECORD is required:
• To save a new record created with CREATE RECORD or DUPLICATE RECORD
• To save data from RECEIVE RECORD
• To save a record modified by a method
• To save a record that contains new or modified subrecord data following an ADD SUBRECORD CREATE SUBRECORD, or MODIFY SUBRECORD command
• During data entry to save the displayed record before using a command that changes the current record
• During data entry to save the current record

You should not execute a SAVE RECORD during the On Validate event for a form that has been accepted. If you do, the record will be saved twice.

Examples

The following example is part of a method that reads records from a document. The code segment receives a record, and then, if it is received properly, saves it:

```
RECEIVE RECORD ([Customers])  ` Receive record from disk
If (OK= 1)  ` If the record is received properly...
   SAVE RECORD ([Customers])  ` save it
End if
```

See Also

CREATE RECORD, Locked, Triggers.

DELETE RECORD

DELETE RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to delete the current record, or Default table, if omitted

Description

DELETE RECORD deletes the current record of table in the process. If there is no current record for table in the process, DELETE RECORD has no effect. In a form, you can create a Delete Record button instead of using this command. After the record is deleted, the current selection for table is empty.

Deleting records is a permanent operation and cannot be undone.

If a record is deleted, the record number will be reused when new records are created. Do not use the record number as the record identifier if you will ever delete records from the database.

Example

The following example deletes an employee record. The code asks the user what employee to delete, searches for the employee's record, and then deletes it:

```
vFind := Request ("Employee ID to delete:")  ` Get an employee ID
If (OK = 1)
   QUERY ([Employee]; [Employee]ID = vFind)  ` Find the employee
   DELETE RECORD ([Employee])  ` Delete the employee
End if
```

See Also

Locked, Triggers.

Records in table

Records in table {(table)} → Number

Parameter	Type		Description
table	Table	→	Table for which to return the number of records, or Default table, if omitted
Function result	Number	←	Total number of records in the table

Description

Records in table returns the total number of records in table. Records in selection returns the number of records in the current selection only. If Records in table is used within a transaction, records created during the transaction will be taken into account.

Example

The following example displays an alert that shows the number of records in a table:

⇒ **ALERT** ("There are "+**String**(**Records in table**([People]))+" records in the table.")

See Also

Records in selection.

Record number

Record number {(table)} → Number

Parameter	Type		Description
table	Table	→	Table for which to return the number of the current record, or Default table, if omitted
Function result	Number	←	Current record number

Description

Record number returns the physical record number for the current record of table. If there is no current record, such as when the record pointer is before or after the current selection, Record number returns –1. If the record is a new record that has not been saved, Record number returns –3.

Record numbers can change. The record numbers of deleted records are reused. Record numbers will also change if you compact the database or perform a recover by tags operation on the database using 4D Tools. During a transaction, a newly created record has a temporary record number. After the transaction has been accepted, the record is assigned a regular record number.

Example

The following example saves the current record number and then searches for any other records that have the same data:

```
⇒    $RecNum:=Record number([People])  ` Get the record number
     QUERY ([People]; [People]Last = [People]Last) ` Anyone else with the last name?
         ` Display an alert with the number of people with the same last name
     ALERT ("There are "+String (Records in selection([People])+" with that name.")
     GOTO RECORD ([People]; $RecNum)  ` Go back to the same record
```

See Also

About Record Numbers, GOTO RECORD, Selected record number, Sequence number.

GOTO RECORD

version 3

GOTO RECORD ({table; }record)

Parameter	Type		Description
table	Table	→	Table in which to go to the record, or Default table, if omitted
record	Number	→	Number returned by Record number

Description

GOTO RECORD selects the specified record of table as the current record. The record parameter is the number returned by the Record number function. After executing this command, the record is the only record in the selection.

If record is less than the smallest record number in the database or greater than the greatest record number in the database, 4th Dimension generates an error message stating that the record number is out of range. If record is equal to the record number of a deleted record, the selection becomes empty.

Note: With this command, you should not use temporary record numbers issued during transactions.

Example

See the example for Record number.

See Also

About Record Numbers, Record number.

Sequence number

version 3

Sequence number {(table)} → Number

Parameter	Type		Description
table	Table	→	Table for which to return the sequence number, or Default table, if omitted
Function result	Number	←	Sequence number

Description

Sequence number returns the next sequence number for table. The sequence number is unique for each table. It is a nonrepeating number that is incremented for each new record created for the table. The numbering starts at 1.

You should use the Sequence number function instead of the #N symbol if:
• You are creating records procedurally
• The sequence number needs to start at a number other than 1
• The sequence number needs an increment greater than 1
• The sequence number is part of a code, for example a part number code

To store the sequence number by means of a method, create a long integer field in the table and assign the sequence number to the field.

The sequence number is the same number assigned by using the #N symbol as the default value for a field in a form. For information on assigning default values, see the *4th Dimension Design Reference* .

If the sequence number needs to start at a number other than 1, just add the difference to Sequence number. For example, if the sequence number must start at 1000, you would use the following statement to assign the number:

⇒ [Table1]Seq Field := **Sequence number** ([Table1]) + 999

Example

The following example is part of a form method. It tests to see if this is a new record; i.e., if the invoice number is an empty string. If it is a new record, the method assigns an invoice number. The invoice number is formed from two pieces of information: the sequence number, and the operator’s ID, which was entered when the database was opened. The sequence number is formatted as a 5-character string:

```
      ` If this is a new part number, create a new invoice number
If ([Invoices]Invoice No = "")
         ` The invoice number is a string that ends with the operator’s ID.
   [Invoices]Invoice No:=String(Sequence number;"00000")+[Invoices]OpID
End if
```

See Also

About Record Numbers, Record number, Selected record number.

About Record Numbers

There are three numbers associated with a record:
• Record number
• Selected record number
• Sequence number

Record Number

The record number is the absolute/physical record number for a record. The record number is automatically assigned to each new record and remains constant for the record until the record is deleted or the file is permanently reordered using 4D Tools. Record numbers start at zero. Record numbers are not unique because record numbers of deleted records are reused for new records. Record numbers also change when the file is permanently reordered using 4D Tools or when the database is compacted or repaired. New records added in transaction are assigned temporary record numbers. They are assigned final record numbers when the transaction is validated.

Selected Record Number

The selected record number is the position of the record in the current selection, and so depends on the current selection. If the selection is changed or sorted, the selected record number will probably change. Numbering for the selected record number starts at one (1).

Sequence Number

The sequence number is a unique nonrepeating number that may be assigned to a field of a record. It is not automatically stored with each record. It starts at 1 and is incremented for each new record that is created. Unlike record numbers, a sequence number is not reused when a record is deleted or when a table is compacted, repaired, or permanently reordered using 4D Tools. Sequence numbers provide a way to have unique ID numbers for records. If a sequence number is incremented during a transaction, the number is not decremented if the transaction is canceled.

Record Number Examples

The following tables illustrate the numbers that are associated with records. Each line in the table represents information about a record. The order of the lines is the order in which records would be displayed in an output form.

• **Data column**: The data from a field in each record. For our example, it contains a person's name.
• **Record Number column**: The record's absolute record number. This is the number returned by the Record number function.
• **Selected Record Number column**: The record's position in the current selection. This is the number returned by the Selected record number function.
• **Sequence Number column**: The record's unique sequence number. This is the number returned by the Sequence number function when the record was created. This number is stored in a field.

After the Records Are Entered

The first table shows the records after they are entered.
• The default order for the records is by record number.
• The record number starts at 0.
• The selected record number and the sequence number start at 1.

Data	Record Number	Selected Record Number	Sequence Number
Tess	0	1	1
Terri	1	2	2
Sabra	2	3	3
Sam	3	4	4
Lisa	4	5	5

Note: The records remain in the default order after a command changes the current selection without reordering it; for example, after the Show All menu command is chosen in the User environment, or after the ALL RECORDS command is executed.

After the Records Are Sorted

The next table shows the same records sorted by name.
• The same record number remains associated with each record.
• The selected record numbers reflect the new position of the records in the sorted selection.
• The sequence numbers never change, since they were assigned when each record was created and are stored in the record.

Data	Record Number	Selected Record Number	Sequence Number
Lisa	4	1	5
Sabra	2	2	3
Sam	3	3	4
Terri	1	4	2
Tess	0	5	1

After a Record Is Deleted

The following table shows the records after Sam is deleted.
• Only the selected record numbers have changed. Selected record numbers reflect the order in which the records are displayed.

Data	Record Number	Selected Record Number	Sequence Number
Lisa	4	1	5
Sabra	2	2	3
Terri	1	3	2
Tess	0	4	1

After a Record Is Added

The next table shows the records after a new record has been added for Liz.
• A new record is added to the end of the current selection.
• Sam's record number is reused for the new record.
• The sequence number continues to increment.

Data	Record Number	Selected Record Number	Sequence Number
Tess	0	1	1
Terri	1	2	2
Sabra	2	3	3
Lisa	4	4	5
Liz	3	5	6

After the Selection is Changed and Sorted

The following table shows the records after the selection was reduced to three records and then sorted.
• Only the selected record number associated with each record changes.

Data	Record Number	Selected Record Number	Sequence Number
Sabra	2	1	3
Liz	3	2	6
Terri	1	3	2

See Also

Record number, Selected record number, Sequence number.

PUSH RECORD

PUSH RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to push record, or Default table, if omitted

Description

PUSH RECORD pushes the current record of table (and its subrecords, if any) onto the table's record stack. PUSH RECORD may be executed before a record is saved.

If you push a record that was unlocked, this record stays locked for all the other processes and users until you pop and unload it.

Example

The following example pushes the record for the customer onto the record stack:

```
⇒	PUSH RECORD ([Customer])  ` Push customer's record onto stack
```

See Also

POP RECORD, Using the Record Stack.

POP RECORD

POP RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to pop record, or Default table, if omitted

Description

POP RECORD pops a record belonging to table from the table's record stack, and makes the record the current record.

If you push a record, change the selection to not include the pushed record, and then pop the record, the current record is not in the current selection. To designate the popped record as the current selection, use ONE RECORD SELECT. If you use any commands that move the record pointer before saving the record, you will lose the copy in memory.

Example

The following example pops the record for the customer off the record stack:

```
⇒	POP RECORD ([Customers])  ` Pop customer's record onto stack
```

See Also

PUSH RECORD, Using the Record Stack.

Using the Record Stack

The commands PUSH RECORD and POP RECORD allow you to put (“push”) records onto the record stack, and to remove (“pop”) them from the stack.

Each process has its own record stack for each table. 4th Dimension maintains the record stacks for you. Each record stack is a last-in-first-out (LIFO) stack. Stack capacity is limited by memory.

PUSH RECORD and POP RECORD should be used with discretion. Each record that is pushed uses part of free memory. Pushing too many records can cause an out-of-memory or stack full condition.

4th Dimension clears the stack of any unpopped records when you return to the menu at the end of execution of your method.

PUSH RECORD and POP RECORD are useful when you want to examine records in the same file during data entry. To do this, you push the record, search and examine records in the file (copy fields into variables, for example), and finally pop the record to restore the record.

Note to version 3 users: While entering a record, if you have to check a multiple field unique value, use the new SET QUERY DESTINATION command. This will save you the calls to PUSH RECORD and POP RECORD that you were making before and after the call to QUERY in order to preserve the data entered in the current record. SET QUERY DESTINATION allows you to make a query that does not change the selection nor the current record.

See Also

POP RECORD, PUSH RECORD, SET QUERY DESTINATION.

39 Relations

Relations

Relations

version 6.0

The commands in this theme, in particular RELATE ONE and RELATE MANY, establish and manage the automatic and non-automatic relations between tables. Before using any of the commands in this theme, refer to the *4th Dimension Design Reference* manual for information about creating relations between tables.

Using Automatic Table Relations with Commands

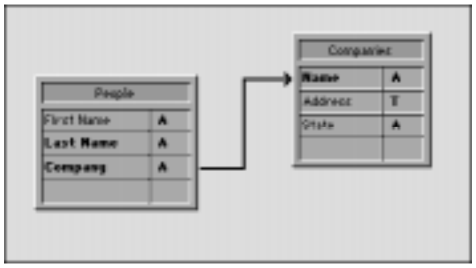
Two tables can be related with automatic table relations. In general, when an automatic table relation is established, it loads or selects the related records in a related table. Many operations cause the relation to be established.

These operations include:
• Data entry
• Listing records on the screen in output forms
• Reporting
• Operations on a selection of records, such as queries, sorts, and applying a formula

To optimize performance, when 4th Dimension establishes automatic relations, only one record becomes the current record for a table. For each of the operations listed above, the related record is loaded according to the following principles:

• If a relation selects only one record of a related table, that record is loaded from disk.
• If a relation selects more than one record of a related table, a new selection of records is created for that table, and the first record in that selection is loaded from disk.

For example, using the database structure displayed here, if a record for the [People] table is loaded and displayed for data entry, the related record from the [Companies] table is selected and is loaded. Similarly, if a record for the [Companies] table is loaded and displayed for data entry, the related records from the [People] table are selected.



In this database structure, the [People] table is referred to as the **Many table**, and the [Companies] table is referred to as the **One table**. To remember this concept, think of "there are many people related to one company" and "each company has many people."

Similarly, the Company field in the [People] table is referred to as the **Many field**, and the Name field in the [Companies] table is referred to as the **One field**.

It is not always possible to have the related field be unique. For example, the [Companies]Name field may have several company records containing the same value. This non-unique situation can be easily handled by creating a relation, which will always be unique, on another field in the related table. This field could be a company ID field.

The following table lists commands that use automatic relations to load related records during operation of the command. All of the commands will establish a Many-to-One relation automatically. Only those commands with Yes in the Many Established column will create a One-to-Many relation automatically.

Command	**One-to-Many established**
ADD RECORD	Yes
ADD SUBRECORD	No
APPLY TO SELECTION	No
DISPLAY SELECTION	No
EXPORT DIF	No
EXPORT SYLK	No
EXPORT TEXT	No
MODIFY RECORD	Yes
MODIFY SUBRECORD	No
MODIFY SELECTION	Yes (in data entry)
ORDER BY	No
ORDER BY FORMULA	No
QUERY BY FORMULA	Yes
QUERY SELECTION	Yes
QUERY	Yes
PRINT LABEL	No
PRINT SELECTION	Yes
REPORT	No
SELECTION TO ARRAY	No
SUBSELECTION TO ARRAY	No

Using Commands to Establish Table Relations

Automatic relations do not mean that the related record or records for a table will be selected simply because a command loads a record. In some cases, after using a command that loads a record, you must explicitly select the related records by using RELATE ONE or RELATE MANY if you need to access the related data.

Some of the commands listed in the previous table (such as the query commands) load a current record after the task is completed. In this case, the record that is loaded does not automatically select the records related to it. Again, if you need to access the related data, you must explicitly select the related records by using RELATE ONE or RELATE MANY.

See Also

AUTOMATIC RELATIONS, CREATE RELATED ONE, OLD RELATED MANY, OLD RELATED ONE, RELATE MANY, RELATE MANY SELECTION, RELATE ONE, RELATE ONE SELECTION, SAVE OLD RELATED ONE, SAVE RELATED ONE.

AUTOMATIC RELATIONS

AUTOMATIC RELATIONS (one; many)

Parameter	Type		Description
one	Boolean	→	Many-to-one relations
many	Boolean	→	One-to-many relations

Description

AUTOMATIC RELATIONS temporarily changes all manual relations into automatic relations for the entire database. The relations stay automatic unless a subsequent call to AUTOMATIC RELATIONS is made.

If one is true, then all manual Many-to-One relations will become automatic. If one is false, all previously changed Many-to-One relations will revert to manual relations.

The same is true for the many parameter, except that manual One-to-Many relations are affected.

Relations that are set as automatic in the Design environment are not affected by this command.
If all relations have been set as manual in the Design environment, this command makes it possible to change them to be automatic, just before executing operations that need the relation to be automatic (such as relational searches and sorts). After the operation is finished, the relation can be changed back to manual.

Example

The following example makes all manual Many-to-One relations automatic and reverts any previously changed One-to-Many relations:

```
⇒	AUTOMATIC RELATIONS (True; False)
```

See Also

Relations, SELECTION TO ARRAY, SUBSELECTION TO ARRAY.

RELATE ONE

RELATE ONE (manyTable | Field{; choiceField})

Parameter	Type		Description
manyTable | Field	Table | Field	→	Table for which to establish all automatic relations, or Field with manual relation to one table
choiceField	Field	→	Choice field from the one table

Description

RELATE ONE has two forms.

The first form, RELATE ONE(manyTable), establishes all automatic Many-to-One relations for manyTable in the current process. This means that for each field in manyTable that has an automatic Many-to-One relation, the command will select the related record in each related table. This changes the current record in the related tables for the process.

The second form, RELATE ONE(manyField{;choiceField), looks for the record related to manyField. The relation does not need to be automatic. If it exists, RELATE ONE loads the related record into memory, making it the current record and current selection for its table.

The optional choiceField can be specified only if manyField is an Alpha field. The choiceField must be a field in the related table. The choiceField must be an Alpha, Numeric, Date, Time, or Boolean field; it cannot be a text, picture, BLOB, or subtable field.

If choiceField is specified and more than one record is found in the related table, RELATE ONE displays a selection list of records that match the value in manyField. In the list, the left column displays related field values, and the right column displays choiceField values.

More than one record may be found if manyField ends with the wildcard character (@). If there is only one match, the list does not appear. Specifying choiceField is the same as specifying a wildcard choice when establishing the table relation. For information about specifying a wildcard choice, refer to the *4th Dimension Design Reference*.

RELATE ONE works with relations to subtables, but you must have a relation to the parent table and to the subtable's related field in order for the relation to be properly established. When using a relation to a subrecord, you must first use RELATE ONE to load the related record into memory, then use a second RELATE ONE command for the subtable.

Example

Let’s say you have an [Invoice] table related to a [Customers] table with two non-automatic relations. One relation is from [Invoice]Bill to to [Customers]ID, and the other relation is from [Invoice]Ship to to [Customers]ID.

Since both relations are to the same table, [Customers], you cannot obtain the billing and shipment information at the same time. Therefore, displaying both addresses in a form should be performed using variables and calls to RELATE ONE. If the [Customers] fields were displayed instead, data from only one of the relations would be displayed.

The following two methods are the object methods for the [Invoice]Bill to and [Invoice]Ship to fields. They are executed when the fields are entered.

Here is the object method for the [Invoice]Bill to field:

```
⇒      RELATE ONE ([Invoice]Bill to)
       vAddress1 := [Customers]Address
       vCity1 := [Customers]City
       vState1 := [Customers]State
       vZIP1 := [Customers]ZIP
```

Here is the object method for the [Invoice]Ship to field:

```
⇒      RELATE ONE ([Invoice]Ship to)
       vAddress2 := [Customers]Address
       vCity2 := [Customers]City
       vState2 := [Customers]State
       vZIP2 := [Customers]ZIP
```

See Also

OLD RELATED ONE, RELATE MANY.

RELATE MANY (oneTable | Field)

Parameter	Type		Description
oneTable | Field	Table | Field	→	Table to establish all one-to-many relations, or One Field

Description

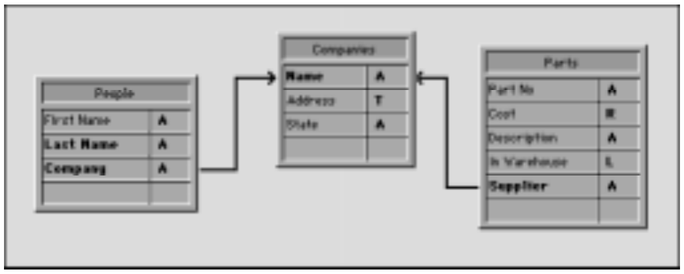
RELATE MANY has two forms.

The first form, RELATE MANY(oneTable), establishes all One-to-Many relations for oneTable. It changes the current selection for each table that has a One-to-Many relation to oneTable. The current selections in the Many tables depend on the current value of each related field in the One table. Each time this command is executed, the current selections of the Many tables will be regenerated.

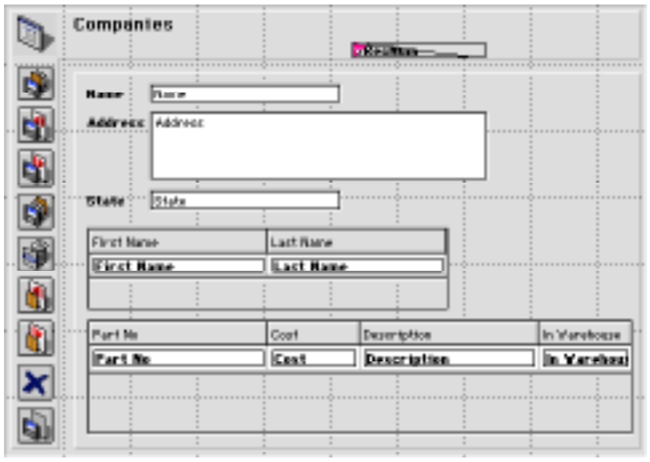
The second form, RELATE MANY(oneField), establishes the One-to-Many relation for oneField. It changes the current selection for only those tables that have relations with oneField. This means that the related records become the current selection for the Many table.

Example

In the following example, three tables are related with automatic relations. Both the [People] table and the [Parts] table have a Many-to-One relation to the [Companies] table.



This form for the [Companies] table will display related records from both the [People] and [Parts] tables.



When the People and Parts forms are displayed, the related records for both the [People] table and the [Parts] table are loaded and become the current selections in those tables.

On the other hand, the related records are not loaded if a record for the [Companies] table is selected programmatically. In this case, you must use the RELATE MANY command.

For example, the following method moves through each record of the [Companies] table. An alert box is displayed for each company. The alert box shows the number of people in the company (the number of related [People] records), and the number of parts they supply (the number of related [Parts] records). In the example, the argument to the ALERTcommand is printed on multiple lines for clarity.

Note that the RELATE MANY command is needed, even though the relations are automatic.

```
        ALL RECORDS ([Companies])  ` Select all records in the table
        ORDER BY ([Companies]; [Companies]Name)  ` Order records in alphabetical order
        For ($i; 1; Records in table ([Companies]))  ` Loop once for each record
⇒          RELATE MANY ([Companies]Name)  ` Select the related records
           ALERT ("Company: "+[Companies]Name+Char (13)+"People in company: "
                                          +String (Records in selection ([People]))+Char(13)+
                       "Number of parts they supply: "+ String (Records in selection ([Parts])))
           NEXT RECORD ([Companies])  ` Move to the next record
        End for
```

See Also

OLD RELATED MANY, RELATE ONE.

CREATE RELATED ONE

CREATE RELATED ONE (field)

Parameter	Type		Description
field	Field	→	Many field

Description

CREATE RELATED ONE performs two actions. If a related record does not exist for field (that is, if a match is not found for the current value of field), CREATE RELATED ONE creates a new related record.

To save a value in the appropriate field, assign values to the One field from the Many field. Call SAVE RELATED ONE to save the new record.

If a related record exists, CREATE RELATED ONE acts just like RELATE ONE and loads the related record into memory.

See Also

SAVE RELATED ONE.

SAVE RELATED ONE

SAVE RELATED ONE (field)

Parameter	Type		Description
field	Field	→	Many field

Description

SAVE RELATED ONE saves the record related to field. Execute a SAVE RELATED ONE command to update a record created with CREATE RELATED ONE, or to save modifications to a record loaded with RELATE ONE.

SAVE RELATED ONE does not apply to subtables, because saving the parent record automatically saves the subrecords.

SAVE RELATED ONE will not save a locked record. When using this command, you must first be sure that the record is unlocked. If the record is locked, the command is ignored, the record is not saved, and no error is returned.

See Also

CREATE RELATED ONE, Locked, RELATE ONE, Triggers.

OLD RELATED ONE

OLD RELATED ONE (field)

Parameter	Type		Description
field	Field	→	Many field

Description

OLD RELATED ONE operates the same way as RELATE ONE does, except that OLD RELATED ONE uses the old value of field to establish the relation.

Note: OLD RELATED ONE uses the old value of the Many field as returned by the Old function. For more information, see the description of the Old command.

OLD RELATED ONE loads the record previously related to the current record. The fields in that record can then be accessed. If you want to modify this old related record and save it, you must call SAVE OLD RELATED ONE. Note that there is no old related record for a newly created record.

See Also

Old, OLD RELATED MANY, RELATE ONE, SAVE OLD RELATED ONE.

SAVE OLD RELATED ONE

SAVE OLD RELATED ONE (field)

Parameter	Type		Description
field	Field	→	Many field

Description

SAVE OLD RELATED ONE operates the same way as SAVE RELATED ONE does, but uses the old relation to the field to save the old related record. Before you use SAVE OLD RELATED ONE, you must load the record with OLD RELATED ONE. Use SAVE OLD RELATED ONE when you want to save modifications to a record loaded with OLD RELATED ONE.

SAVE OLD RELATED ONE will not save a locked record. When using this command, you must first be sure that the record is unlocked. If the record is locked, the command is ignored, the record is not saved, and no error is returned.

See Also

Locked, OLD RELATED ONE, Triggers.

OLD RELATED MANY

OLD RELATED MANY (field)

Parameter	Type		Description
field	Field	→	One field

Description

OLD RELATED MANY operates the same way RELATE MANY does, except that OLD RELATED MANY uses the old value in the one field to establish the relation.

Note: OLD RELATED MANY uses the old value of the many field as returned by the Old function. For more information, see the description of the Old command.

OLD RELATED MANY changes the selection of the related table, and selects the first record of the selection as the current record.

See Also

OLD RELATED ONE, RELATE MANY.

RELATE ONE SELECTION

version 6.0 (Modified)

RELATE ONE SELECTION (manyTable; oneTable)

Parameter	Type		Description
manyTable	Table	→	Many table name (from which the relation starts)
oneTable	Table	→	One table name (to which the relation refers)

Description

The command RELATE ONE SELECTION creates a new selection of records for the table oneTable, based on the selection of records in the table manyTable.

This command can only be used if there is a relation from manyTable to oneTable. RELATE ONE SELECTION can work across several levels of relations. There can be several related tables between manyTable and oneTable. The relations can be manual or automatic.

Warning: Do not use this command inside a transaction.

Example

The following example finds all the clients whose invoices are due today.

Here is one way of creating a selection in the [Customers] table, given a selection of records in the [Invoices] table:

```
CREATE EMPTY SET([Customers];"Payment Due")
QUERY([Invoices];[Invoices]DueDate = Current date)
While(Not(End selection([Invoices])))
   RELATE ONE ([Invoices]CustID)
   ADD TO SET([Customers];"Payment Due")
   NEXT RECORD([Invoices])
End while
```

The following technique uses RELATE ONE SELECTION to accomplish the same result:

```
   QUERY([Invoices];[Invoices]DueDate = Current date)
⇒  RELATE ONE SELECTION([Invoices];[Customers])
```

See Also

QUERY, RELATE MANY SELECTION, RELATE ONE, Sets.

RELATE MANY SELECTION

version 6.0 (Modified)

RELATE MANY SELECTION (field)

Parameter	Type		Description
field	Field	→	Many table field (from which the relation starts)

Description

The command RELATE MANY SELECTION generates a selection of records in the Many table, based on a selection of records in the One table.

Note: RELATE MANY SELECTION changes the current record for the One table.

Warning: Do not use this command inside a transaction.

Example

This example selects all invoices made to the customers whose credit is greater than or equal to $1,000. The [Invoices] table field [Invoices]Customer ID relates to the [Customer] table field [Customers]ID Number.

```
    ` Select the Customers
QUERY ([Customers];[Customers]Credit>=1000)
    ` Find all invoices related to any of these customers
RELATE MANY SELECTION ([Invoices]Customer ID)
```

See Also

QUERY, RELATE ONE, RELATE ONE SELECTION.

40 Resources

Resources

version 6.0

A **resource** is data of any kind stored in a defined format in a **.RSR** Windows file or in the **resource fork** of a Macintosh file. Resources typically include data such as strings, pictures, icons and so on. As a matter of fact, you can create and use your own kinds of resources and store whatever data you want into them.

Data Fork and Resource Fork

On Macintosh, each file can have a **data fork** and a **resource fork**. The data fork of a Macintosh file is the equivalent of a file on Windows and UNIX. The resource fork of a Macintosh file contains the Macintosh-based resources of the file and has no direct equivalent on Windows or UNIX.

Windows-based resources are stored and mixed with the other data of the file. For example, in a Windows application, a .EXE file can contain both resource data and code. In order to maintain the platform independence of your 4D applications, 4th Dimension works with Macintosh-based resources on both the Macintosh and Windows platforms.

4D Transporter

Since resource forks do not exist on Windows, the **4D Transporter** utility program (delivered with the Macintosh version of 4th Dimension) enables you to transport a 4D database from Windows to Macintosh and vice-versa.

When you transport a 4D database from Windows to Macintosh, the .4DB and .RSR files of the database structure file are **merged** into one Macintosh file. The .4DB file becomes the data fork of the Macintosh structure file, and the .RSR file becomes the resource fork the Macintosh structure file. Conversly, when you transport a 4D database from Macintosh to Windows, the Macintosh structure file is **split** into two files. Its data fork becomes the .4DB file and the resource fork becomes the .RSR file.

In fact, this is the sole purpose of the 4D Transporter utility—splitting or merging data and resource forks of files. It does not translate or modify the actual data stored in the forks and files. For more information about transporting 4D databases between platforms, refer to the *4D Transporter Reference* manual.

Resource Files

No matter what platform you are using, a 4D database structure file is not the only type of file with resources. The 4D application itself contains resources. On Macintosh, the 4D resources are stored in the resource fork of the application. On Windows, they are stored in the 4D.RSR file, the resource part of the 4D application whose executable code is stored in the 4D.EXE file.

4D Plug-ins can also contain resources. For example, the 4D Write ACI plug-in contains resources. On Macintosh, these are stored in the resource fork of 4D Write 6.0. On Windows, they are stored in the 4DWRITE.RSR file.
The data file of a 4D database can also contains resources. For example, if you lock a data file for exclusive use with a particular structure file (using the **Customizer Plus** utility), this operation adds the same WEDD ("WEDD" for "wedding") resource into the structure and data files. On Macintosh, the resource is added to the resource fork of the data file. On Windows, the resource is stored in the .4DR file, the resource file for the data file.

Note: Customizer Plus is delivered with the Windows and Macintosh versions of 4th Dimension)

On Windows, with the exception of the data file's .4DR file, you usually detect standard 4D files containing Macintosh-based resources as files with the file extension .RSR. Note that the command Create resource files uses .RES as default file extension.

Creating Your Own Resource Files

In addition to the resource files provided by 4D, you can create and use your own resource files using the 4D commands Create resource file and Open resource file. These two commands return a **resource file reference number** that uniquely identifies the open resource file. The resource file reference number is the equivalent of the document reference number for regular files returned by System documents commands such as Open document. All the 4D Resources commands optionally expect a resource file reference number. After you have finished with a resource file, remember to close it using the command CLOSE RESOURCE FILE.

The Resource Files Chain

When you work with a 4D database, you can either work with **all the currently open resource files** or with a **specific resource file**.

Multiple resource files can be open at the same time. This is always the case from within a 4D database. The following files are open:

• On Macintosh, the System resource file.
• On Windows, the ASIPORT.RSR file (it contains part of the Macintosh system resources).
• The 4D application resource file.
• The database structure resource file.
• The database data file resource file may be optionally open.
• Finally, you can open your own resource file using the command Open resource file.

This list of open resource files is called the **resource files chain**. You can search for a given resource in two ways:

• If you pass a resource file reference number to a resource 4D command, the resource is searched for in that resource file only.
• If you do not pass a resource file reference number to a 4D Resource command, the resource is searched for in all currently open resource files, starting with the most recently opened file and ending with the first opened file. The resource files chain is thus browsed in the reverse order of opening—the last opened resource file is examined first.

Here is an example:

```
$vhResFile:=Create resource file("Just_a_file")
If (OK=1)
    ARRAY STRING(63;asSomeStrings;0)
    STRING LIST TO ARRAY(8;asSomeStrings;$vhResFile)
    ALERT("The size of the array is "+String(Size of array(asSomeStrings))+" element(s).")
    STRING LIST TO ARRAY(8;asSomeStrings)
    ALERT("The size of the array is "+String(Size of array(asSomeStrings))+" element(s).")
    CLOSE RESOURCE FILE($vhResFile)
End if
```

At execution of this method, the first alert will display “The size of the array is 0 element(s)” and the second alert will display “The size of the array is 634 element(s)”.

The first call:

```
    STRING LIST TO ARRAY(8;asSomeStrings;$vhResFile)
```

looks for the resource "STR#" ID=8 only in the resource file just created and open by the call to Create resource file. Because the file is new and therefore empty, the resource is not found.

The second call:

```
STRING LIST TO ARRAY(8;asSomeStrings)
```

looks for the resource "STR#" ID=8 in all the currently open resource files. Since the file just created and opened (by the call to Create resource file) does not contain that resource, STRING LIST TO ARRAY then looks for the resource in the database structure resource file. This resource file does not contain that resource either, so STRING LIST TO ARRAY then examines the 4D resource file, locates the resource in this file, and populates the array with it.

Conclusion: When working with resource files, if you want to access a specific file, make sure to pass the resource file reference number to a 4D Resources command. Otherwise, the command assumes that you do not care which file is the source of the resources.

Resource Type

A resource file is highly structured. In addition to the data of each resource, it contains a header and a map that fully describe its contents.

Resources are classified by **types**. A resource type is always denoted by a 4-character string. A resource type is both case sensitive and diacritical sensitive. For example, the resource types “Hi_!”, “hi_!” and “HI_!” are all different.

Important: Resource types with lowercase characters are reserved for use by the Operating System. Avoid designating your own resource types with lowercase characters.

The following is a list of some commonly-used resource types:

• A resource of type “STR#” is a resource containing a list of Pascal strings. This resource is called a **string list resource**.
• A resource of type “STR ” (note the space as fourth character) is a resource containing an individual Pascal string. This resource is called a **string resource**.
• A resource of type “TEXT” is a resource containing a text string without length. This resource is called a **text resource**.
• A resource of type “PICT” is a resource containing a Macintosh-based QuickDraw picture that you can use and display on both Macintosh and Windows with 4D. This resource is called a **picture resource**.

• A resource of type “cicn” is a resource containing a Macintosh-based color icon that you can use and display with 4D on both Macintosh and Windows. This resource is called a **color icon resource**. For example, a “cicn” resource can be associated with an item of a hierarchical list, using the command SET LIST ITEM PROPERTIES.

In addition to the standard resource types, you can create you own types. For example, you can decide to work with resources of type “MTYP” (for “My Type”).

To obtain the list of resource types currently present in all open resource files or in a particular resource file, use the command RESOURCE TYPE LIST. Then, to obtain the list of a specified type of resource present in all open resource files or in a particular resource file, use the command RESOURCE LIST. This command returns the IDs and Names (see next section) of all resources of a given type.

WARNING: Many applications rely on the resource type for working with its contents. For example, while accessing a “STR#” resource, applications expect to find a string list in the resource. Do NOT store inconsistent data in resources of standard types; this may lead to system errors in your 4D application or in other applications.

WARNING: A resource is a highly structured file—do NOT access the file with commands other than Resources commands. Note that nothing prevents you from passing a resource file reference number (formally a 4D time expression like the document reference number) to a command such as SEND PACKET. However, if you do so, you will probably damage the resource file.

WARNING: A resource file can contain about up to 2,700 individual resources. Do NOT attempt to exceed this limit. Note that nothing prevents you from doing so; however, this will damage the resource file and make it unusable.

Resource Name and Resource ID

A resource has a **resource name**. A resource name can be up to 255 characters, and is diacritical sensitive but not case sensitive. Resource names are useful for describing a resource, but you access a resource using its type and ID number. Resource names are not unique; several resources can have the same name.

A resource has a **resource ID number** (for short, resource ID or ID). This ID is unique within a resource type and a resource file. For example:

• One resource file can contain a resource “ABCD” ID=1 and a resource “EFGH” ID=1.
• Two resource files can contain a resource with the same type and ID.

When you access a resource using a 4D command, you indicate its type and ID. If you do not specify the resource file in which you are looking for this resource, the command returns the occurrence of the resource found in the first examined resource file. Remember that resource files are examined in the reverse order in which they have been opened.

The range of a resource ID is -32,768..32,767.

Important: Use the range 15,000..32,767 for your own resources. Do NOT use negative resource IDs; these are reserved for use by the Operating System. Do NOT use resource IDs in the range 0..14,999; this range is reserved for use by 4th Dimension.

To obtain the IDs and names of a given resource type, use the command RESOURCE LIST.

To obtain the name of an individual resource, use the command Get resource name.

To change the name of and individual resource, use the command SET RESOURCE NAME.

As each 4D command optionally accepts a resource file reference number, you can easily deal with resources having the same type and ID in two different resource files. The following example copies all the "PICT" resources from one resource file to another:

```
    ` Open an existing resource file
$vhResFileA:=Open resource file("")
If (OK=1)
        ` Create a new resource file
    $vhResFileB:=Create resource file("")
    If (OK=1)
            ` Get the ID and Name lists of all the resources of type "PICT"
            ` located in the resource file A
        RESOURCE LIST("PICT";$aiResID;$asResName;$vhResFileA)
            ` For each resource:
        For($vlElem;1;Size of array($aiResID))
            $viResID:=$aiResID{$vlElem}
                ` Load the resource from file A
            GET RESOURCE ("PICT";$viResID;vxResData;$vhResFileA)
                ` If the resource could be loaded
            If (OK=1)
                    ` Add and write the resource into file B
                SET RESOURCE ("PICT";$viResID;vxResData;$vhResFileB)
                    ` If the resource could be added and written
                If (OK=1)
                        ` Copy also the name of the resource
                    SET RESOURCE NAME("PICT;$viResID;$asResName{$vlElem}
                                                                    ;$vhResFileB)
                        ` As well as its properties
                        ` (see Resource Properties discussion further below)
                    $vlResAttr:=Get resource properties("PICT";$viResID;$vhResFileA)
                    SET RESOURCE PROPERTIES("PICT";$viResID;$vlResAttr;$vhResFileB)
                Else
                    ALERT("The resource PICT ID="+String($viResID)+
                                                        " could not be added.")
                End if
            Else
               ALERT("The resource PICT ID="+String($viResID)+" could not be loaded.")
            End if
         End for
         CLOSE RESOURCE FILE($vhResFileB)
      End if
      CLOSE RESOURCE FILE($vhResFileA)
   End if
```

Resource Properties

Besides its type, name and ID, a resource has additional **properties** (also called attributes). For example, a resource may or may not be purged. This attribute tells the Operating System whether or not a loaded resource can be purged from memory when free memory is required for allocating another object. As shown in the previous example, when creating or copying a resource, it can be important to not only copy the resource, but also its name and properties. For a complete explanation of resource properties, see the description of the commands Get resource properties and SET RESOURCE PROPERTIES.

Handling Resource Contents

To load a resource of any type into memory, call GET RESOURCE, which returns the contents of the resource in a BLOB.

To add or rewrite a resource on disk, call SET RESOURCE, which sets the contents of the resource to the contents of the BLOB you pass.

To delete an existing resource, use the command DELETE RESOURCE.

To simplify handling of standard resource types, 4D provides additional built-in commands that save you from having to parse a BLOB in order to extract the resource data:

• STRING LIST TO ARRAY populates a String or Text array with the strings contained in a string list resource.
• ARRAY TO STRING LIST creates or rewrites a string list resource with the elements of a String or Text array.
• Get indexed string returns a particular string from a string list resource.
• Get string resource returns the string from a string resource.
• SET STRING RESOURCE creates or rewrites a string resource.
• Get text resource returns the text of a text resource.
• SET TEXT RESOURCE creates or rewrites a text resource.
• GET PICTURE RESOURCE returns the picture of a picture resource.
• SET PICTURE RESOURCE creates or rewrites a picture resource.
• GET ICON RESOURCE returns a color icon resource as a picture.

Note that these commands are provided to simplify manipulation of standard resource types; however, they do not prevent you from using GET RESOURCE and SET RESOURCE using BLOBs. For example, this line of code:

```
ALERT(Get text resource(20000))
```

is the shorter equivalent of:

```
GET RESOURCE("TEXT";20000;vxData)
If (OK=1)
   $vlOffset:=0
   ALERT(BLOB to text(vxData;Text without length;$vlOffset;BLOB Size(vxData)))
End if
```

4D Commands and Resources

In addition to the Resources commands described in this chapter, there are other 4D commands that work with resources and resource files:

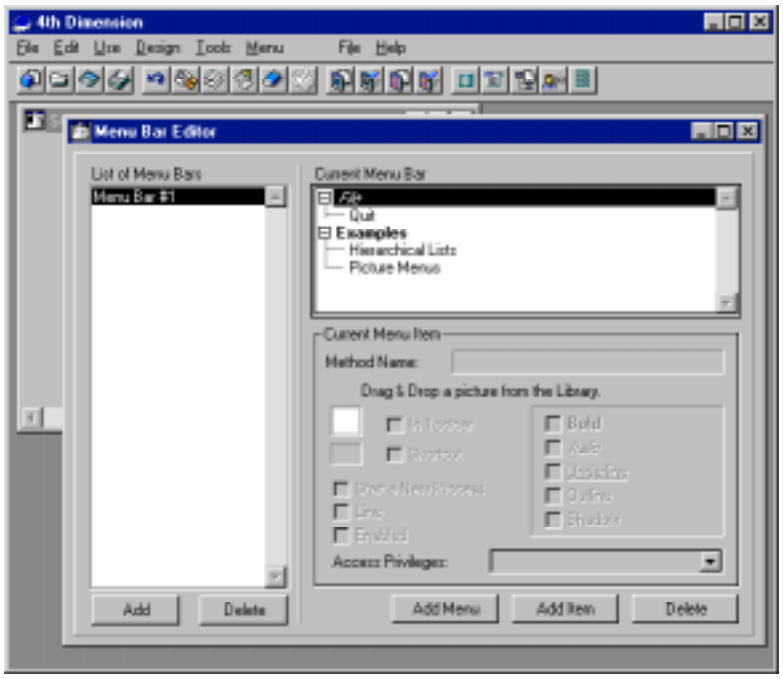
• On Macintosh, DOCUMENT TO BLOB and BLOB TO DOCUMENT can load and write the whole resource fork of a Macintosh file.
• Using the commands SET LIST ITEM PROPERTIES and SET LIST PROPERTIES, you can associate picture or color icon resources to the items of a list or use color icon resources as nodes of a list.
• The PLAY command plays “snd ” resources on both Macintosh and Windows.
• The SET CURSOR command changes the appearance of the mouse using “CURS” resources.

See Also

BLOB Commands, OS Resource Manager Errors, Resources and 4D Insider: an example.

Resources are very convenient way to deal with localization issues when developing and maintaining a 4D database in different languages for the international market.

Let’s look at an example. The following figure shows the menu bar of a database in English:

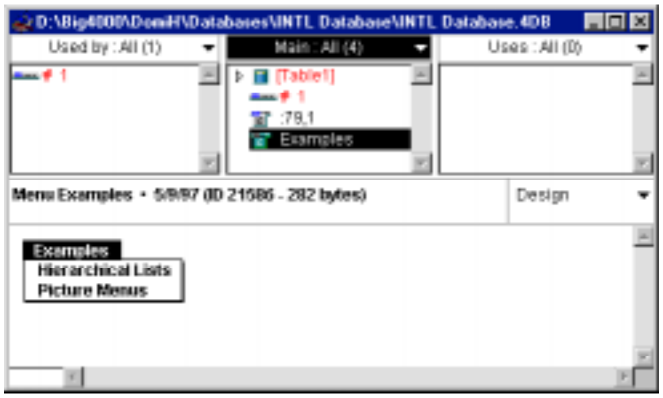


The title of the **File** menu already refers to a resource, while its menu item **Quit** does not. The **Examples** menu is composed of the menu items **Hierarchical Lists** and **Picture Menus**. This menu and its items do not refer to resources.

Using 4D Insider, it is possible to transform the literals of the menu bar into references to strings stored in STR# resources. Let’s see how to perform this operation.

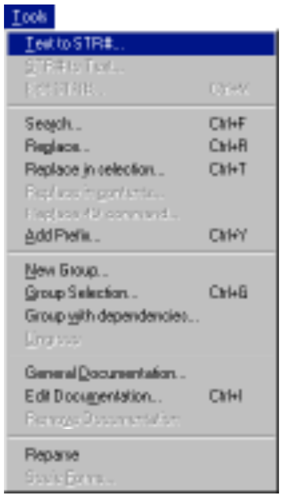
Note: 4D Insider is the 4D cross-reference and library management tool delivered with 4D Desktop.

1. Open the database using 4D Insider. The following figure shows the menu bar in the 4D Insider browser window:

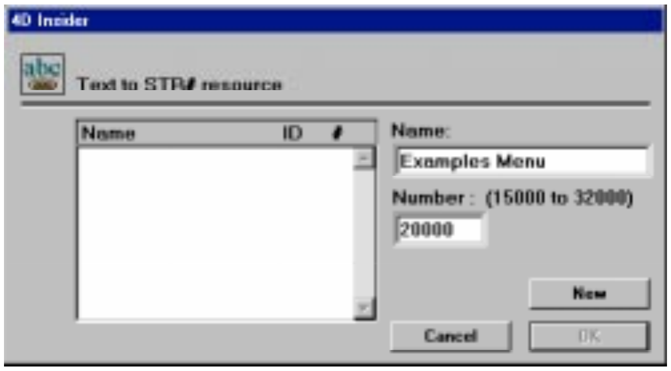


Menu Examples • 5/9/97 (ID 21586 - 282 bytes)

2. At this point, the menu bar can be transformed to refer to a STR# resource. To do so, select **Text to STR#** from the 4D Insider **Tools** Menu:



The **TEXT to STR# resource** dialog box appears:


4D Insider

Text to STR# resource

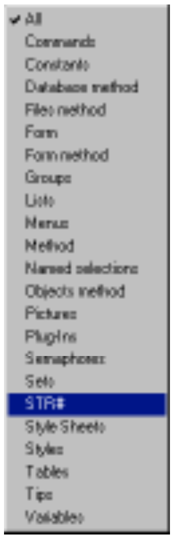
Name	ID	#

Name:
Examples Menu

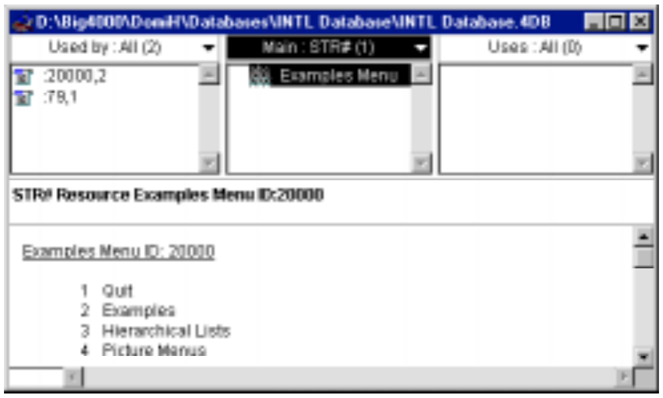
Number : (15000 to 32000)
20000

3. Enter the resource name and ID, then click **New**. For example, **Examples Menu** is the resource name and **20000** is the resource ID. The resource is created.

4. Select **STR#** in the popup menu of the browser window's main list:



5. Double-click the STR# 20000 list item to display its contents:



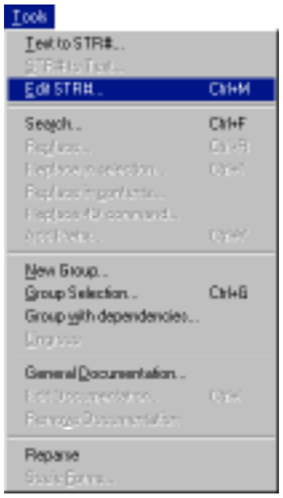
STR# Resource Examples Menu ID:20000

Examples Menu ID: 20000

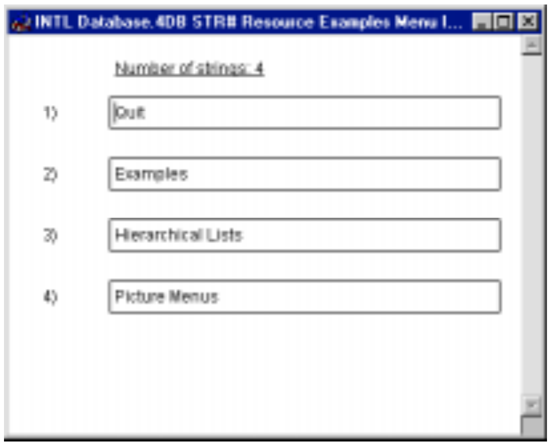
```
1  Quit
2  Examples
3  Hierarchical Lists
4  Picture Menus
```

Now that these strings are stored in a resource, it is possible to change their values without tampering with the logic of your database development.

6. To change the values, select **Edit STR#** from the 4D Insider **Tools** menu while the **Examples Menu** resource is selected in the main list of the browser window:



The STR# resource editing window appears:


INTL Database.4DB STR# Resource Examples Menu I...

Number of strings: 4

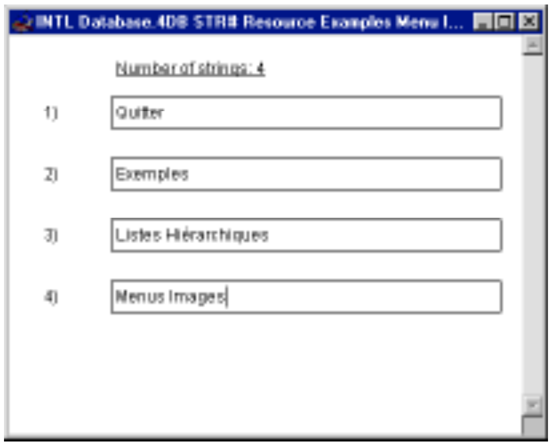
1) Quit

2) Examples

3) Hierarchical Lists

4) Picture Menus

7. Translate the strings to another language. In the following figure, the strings have been translated to French:


INTL Database.4DB STR# Resource Examples Menu I...

Number of strings: 4

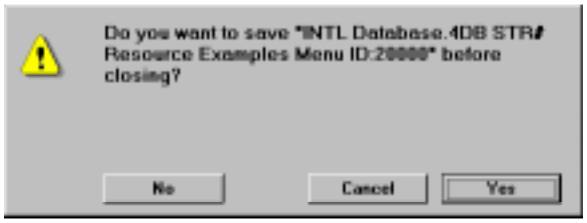
1) Quitter

2) Exemples

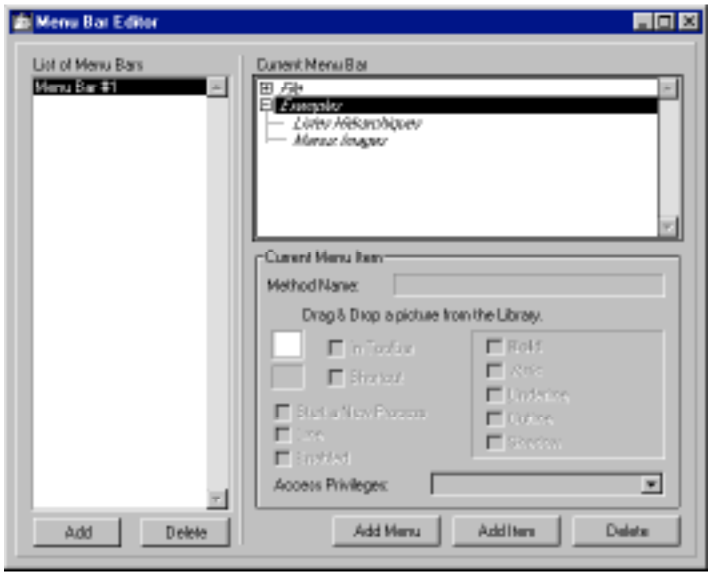
3) Listes Hiérarchiques

4) Menus Images

8. Once you have performed the translation, close the window. Click **Yes** in the confirm dialog box:

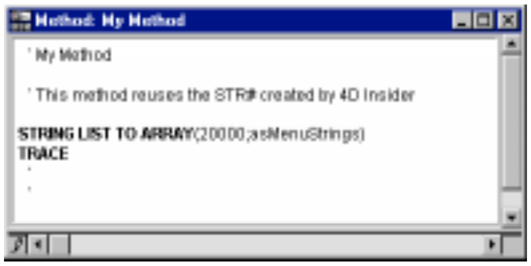

Do you want to save "INTL Database.4DB STR# Resource Examples Menu ID:20000" before closing?

9. At this point, quit 4D Insider and reopen the database with 4th Dimension. The 4D Design environment Menu Bar Editor now shows the menu bar with the references to the resources in French:



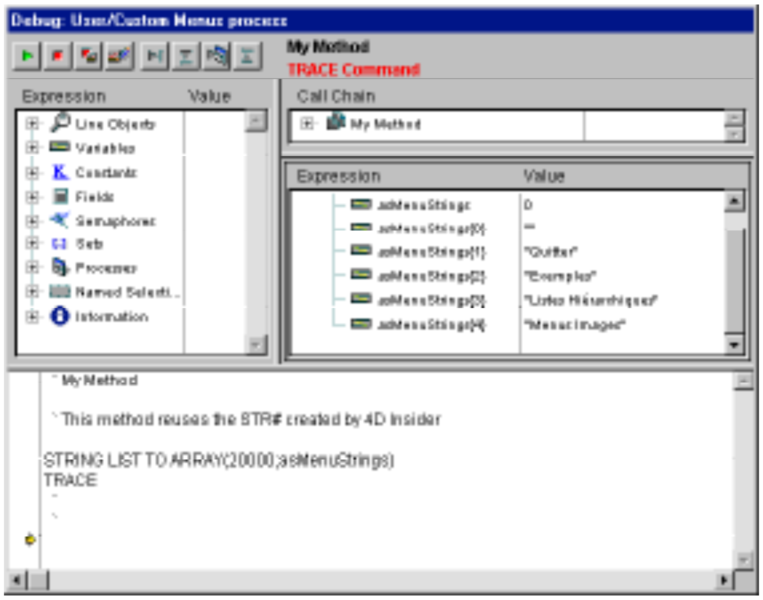
If you own the 4D desktop package, refer to the *4D Insider Reference* manual for more information about this process. In addition, for more information about using references to resources in menus bars as well as objects in your database forms, refer to the *4th Dimension Design Reference* manual.

The 4D Resources commands enable you to use the resources created by 4D Insider. The following method uses the STRING LIST TO ARRAY command to load the STR# resource (created using 4D Insider) into an array:



```
  ` My Method

  ` This method reuses the STR# created by 4D Insider

STRING LIST TO ARRAY(20000;asMenuStrings)
TRACE
  `
  `
```

In the Debugger window, you can see that the array is populated with the strings translated in 4D Insider:



Expression	Value
asMenuStrings	0
asMenuStrings{0}	""
asMenuStrings{1}	"Quitter"
asMenuStrings{2}	"Exemples"
asMenuStrings{3}	"Listes Hiérarchiques"
asMenuStrings{4}	"Menus Images"

```
  ` My Method

  ` This method reuses the STR# created by 4D Insider

STRING LIST TO ARRAY(20000;asMenuStrings)
TRACE
  `
  `
```

See Also

Resources.

Open resource file

Open resource file (resFilename{; fileType}) → DocRef

Parameter	Type		Description
resFilename	String	→	Short or long name of resource file, or Empty string for standard Open File dialog box
fileType	String	→	Mac OS file type (4-character string), or Windows file ext. (1- to 3-character string), or All files, if omitted
Function result	DocRef	←	Resource file reference number

Description

The command Open resource file opens the resource file whose name or pathname you pass in resFileName.

If you pass a filename, the file should be located in the same folder as the structure file of the database. Pass a pathname to open a resource file located in another folder.

If you pass an empty string in resFileName, the Open File dialog box is presented. You can then select the resource file to open. If you cancel the dialog, no resource file is open; Open resource file returns a null DocRef and sets the OK variable to 0.

If the resource file is opened correctly, Open resource file returns its resource file reference number and sets the OK variable to 1. If the resource file does not exist, or if the file you try to open is not a resource file, an error is generated.

On Macintosh, if you use the Open File dialog box, all files are presented by default. To show a particular type of file, specify the file type in the optional fileType parameter.

On Windows, if you use the Open File dialog box, all files are presented by default. To show a particular type of file, in fileType, pass a 1- to 3-character Windows file extension or a Macintosh file type mapped using the command MAP FILE TYPES.

Remember to call CLOSE RESOURCE FILE for the resource file. Note, however, that when you quit the application (or open another database), 4D automatically closes all the resource files you opened using Open resource file or Create resource file .

Unlike the Open document command, which opens a document (data fork on Macintosh) with exclusive read-write access, Open resource file does not prevent you from opening a resource file already open from within the 4D session. For example, if you try to open the same document twice using Open document, an I/O error will be returned at second attempt. On the other hand, if you try to open a resource file already open from within the 4D session, Open resource file will return the resource file reference number to the file already open. Even if you open a resource file several times, you need to call CLOSE RESOURCE FILE once in order to close that file. Note that this is permitted if the resource file is open from within the 4D session; if you try open a resource file already opened by another application, you will get an I/O error.

This multiple-opening capability enables you to easily obtain the reference numbers of the 4D application and database resource files without tampering with normal 4D operations (see examples 5 and 6).

WARNING

• Use caution when accessing the 4D application resource file. Do NOT modify the resources of the 4D application; you could inadvertently damage the program and provoke system errors. Also, remember that your database can be used in various environments (4D, Runtime, 4D Engine, 4D Server and 4D Client).
• If you access the database resource file and intend to programmatically add, delete or modify its resources, be sure to test the environment in which you are running. With 4D Server, this will probably lead to serious issues. For example, if you modify a resource on the server machine (via a database method or a stored procedure), you will definitely affect the built-in 4D Server administration service that distributes resources (transparently) to the workstations. Note that with 4D Client, you do not have direct access to the structure file; it is located on the server machine.
• For these reasons, if you use resources, store them in your own files.
• When working with your own resources, do NOT use negative resource IDs; they are reserved for use by the Operating System. Do NOT use resource IDs in the range 0..14,999; this range is reserved for use by 4th Dimension. Use the range 15,000..32,767 for your own resources. Remember that once you have opened a resource file, it will be the first file to be searched in the resource files chain. If you store a resource in that file with an ID in the range of system or 4D resources, this resource will be found by commands such as GET RESOURCE and also by internal routines of the 4D application. This may be the result you want to achieve, but if you are not sure, do NOT use these ranges, as they may lead to system errors.
• Resource files are highly structured files and cannot accept more than 2,700 resources per file. If you work with files containing a large number of resources, it is a good idea to test that number before adding new resources to a file. See the Count resources examples listed for the command RESOURCE TYPE LIST.

After you have opened a resource file, you can analyze the contents of the file using the commands RESOURCE TYPE LIST and RESOURCE LIST.

Examples

1. The following example tries to open, on Windows, the resource file “MyPrefs.res” located in the database folder:

```
⇒	$vhResFile:=Open resource file("MyPrefs";"res ")
```

On Macintosh, the example tries to open the file “MyPrefs”.

2. The following example tries to open, on Windows. the resource file “MyPrefs.rsr” located in the database folder:

```
⇒	$vhResFile:=Open resource file("MyPrefs";"rsr")
```

On Macintosh, the example tries to open the file “MyPrefs”.

3. The following example displays the Open file dialog box showing all types of files:

```
⇒	$vhResFile:=Open resource file("")
```

4. The following example displays the Open file dialog box showing files created by the Create resource file command, using the default file type:

```
⇒	$vhResFile:=Open resource file("";"res ")
	If (OK=1)
		ALERT("You just opened “"+Document+"”.")
		CLOSE RESOURCE FILE($vhResFile)
	End if
```

5. The following example returns in $vhStructureResFile the reference number to the database structure resource file:

```
	If (On Windows)
⇒		$vhStructureResFile:=Open resource file(Replace string(
								Structure file;".4DB";".RSR"))
	Else
⇒		$vhStructureResFile:=Open resource file(Structure file)
	End if
```

6. The following example returns in $vhApplResFile the reference number to the 4D application resource file:

```
        If (On Windows)
⇒          $vhApplResFile:=Open resource file(Replace string(Application file;".EXE";".RSR"))
        Else
⇒          $vhApplResFile:=Open resource file(Application file)
        End if
```

See Also

CLOSE RESOURCE FILE, Create resource file, Resources.

System Variables and Sets

If the resource file is successfully opened, the OK variable is set to 1. If the resource file could not be opened or if the user clicked Cancel in the Open file dialog box, the OK variable is set to 0 (zero).

If the resource file is successfully opened using the Open file dialog box, the Document variable is set to the pathname of the file.

Error Handling

If the resource file could not be opened due to a resource or I/O problem, an error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

Create resource file

version 6.0

Create resource file (resFilename{; fileType}) → DocRef

Parameter	Type		Description
resFilename	String	→	Short or long name of resource file, or empty string for standard Save File dialog box
fileType	String	→	Mac OS file type (4-character string), or Windows file ext. (1- to 3-character string), or Resource ("res " / .RES) document, if omitted
Function result	DocRef	←	Resource file reference number

Description

The command Create resource file creates and opens a new resource file whose name or pathname is passed in resFileName.

If you pass a filename, the file will be located in the same folder as the structure file of the database. Pass a pathname to create a resource file located in another folder.

If the file already exists and is not currently open, Create resource file overrides it with a new empty resource file. If the file is currently open, an I/O error is returned.

If you pass an empty string in resFileName, the Save File dialog box is presented. You can then choose the location and the name of the resource file to be created. If you cancel the dialog, no resource file is created; Create resource file returns a null DocRef and sets the OK variable to 0.

If the resource file is correctly created and opened, Create resource file returns its resource file reference number and sets the OK variable to 1. If the resource file cannot be created, an error is generated.

On Macintosh, the default file type for a file created with Create resource file is “res ”.
On Windows, the default file extension is “.res”.

To create a file of another type:

• On Macintosh, pass the file type in the optional parameter fileType.
• On Windows, in fileType, pass a 1- to 3-character Windows file extension or a Macintosh file type mapped using the command MAP FILE TYPES.

Remember to call CLOSE RESOURCE FILE for the resource file. Note, however, when you quit the application (or open another database), 4D automatically closes all the resource files you opened using Create resource file or Create resource file.

Examples

1. The following example tries to create and ope, on Windows, the resource file “MyPrefs.res” located in the database folder:

```
⇒	$vhResFile:=Create resource file("MyPrefs")
```

On Macintosh, the example tries to create and open the file “MyPrefs”.

2. The following example tries to create and open, on Windows, the resource file “MyPrefs.rsr” located in the database folder:

```
⇒	$vhResFile:=Create resource file("MyPrefs";"rsr")
```

On Macintosh, the example tries to create and open the file “MyPrefs”.

3. The following example displays the Save File dialog box:

```
⇒	$vhResFile:=Create resource file("")
	If (OK=1)
		ALERT("You just created “"+Document+"”.")
		CLOSE RESOURCE FILE($vhResFile)
	End if
```

See Also

CLOSE RESOURCE FILE, ON ERR CALL, Open resource file, Resources.

System Variables and Sets

If the resource file is successfully created and opened, the OK variable is set to 1. If the resource file could not be created or if the user clicked Cancel in the Save File dialog box, the OK variable is set to 0 (zero).

If the resource file is successfully created and opened through the Save File dialog box, the Document variable is set to the pathname of the file.

Error Handling

If the resource file could not be created or opened due to a resource or I/O problem, an error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

CLOSE RESOURCE FILE

version 6.0

CLOSE RESOURCE FILE (resFile)

Parameter	Type		Description
resFile	DocRef	→	Resource file reference number

Description

The command CLOSE RESOURCE FILE closes the resource file whose reference number is passed in resFile.

Even if you have opened the same resource file several times, you need to call CLOSE RESOURCE FILE only once in order to close that file.

If you apply CLOSE RESOURCE FILE to the 4D application or database resource files, the command detects it and does nothing.

If you pass an invalid resource file reference number, the command does nothing.

Remember to eventually call CLOSE RESOURCE FILE for a resource file that you have opened using Open Resource file or Create resource file. Note that when you quit the application (or open another database), 4D automatically closes all the resource files you opened.

Example

The following example creates a resource file, adds a string resource and closes the file:

```
      $vhDocRef:=Create resource file("Just a file")
      If (OK=1)
         SET STRING RESOURCE(20000;"Just a string";$vhDocRef)
⇒        CLOSE RESOURCE FILE($vhDocRef)
      End if
```

See Also

Create resource file, Open resource file.

System Variables and Sets

None is affected.

RESOURCE TYPE LIST

version 6.0

RESOURCE TYPE LIST (resTypes{; resFile})

Parameter	Type		Description
resTypes	String Array	←	List of available resource types
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted

Description

The command RESOURCE TYPE LIST populates the array resTypes with the resource types of the resources present in the resource files currently open.

If you pass a valid resource file reference number in the optional parameter resFile, only the resources from that file are listed. If you do not pass the parameter resFile, all the resources from the current open resource files are listed.

You can predeclare the array resTypes as a String or Text array before calling RESOURCE TYPE LIST. If you do not predeclare the array, the command creates resTypes as a Text array.

After the call, you can test the number of resource types found by applying the command Size of array to the array resTypes.

Examples

1. The following example populates the array atResType with the resource types of the resources present in all the resource files currently open:

```
⇒	RESOURCE TYPE LIST(atResType)
```

2. The following example tells you if the Macintosh 4D structure file you are using contains old 4D plug-ins that will need to be updated in order to use the database on Windows:

```
	$vhResFile:=Open resource file(Structure file)
⇒	RESOURCE TYPE LIST(atResType;$vhResFile)
	If (Find in array(atResType;"4DEX")>0)
	   ALERT("This database contains old model Mac OS 4D plug-ins."+(Char(13)*2)+
	                    "You will have to update them for using this database on Windows.")
	End if
```

Note: The structure file is not the only file where old version plug-ins can be stored. The database can also include a Proc.Ext file.

3. The following project method returns the number of resources present in a resource file:

```
     ` Count resources project method
     ` Count resources ( Time ) -> Long
     ` Count resources ( DocRef ) -> Number of resources

   C_LONGINT($0)
   C_TIME($1)

   $0:=0
⇒  RESOURCE TYPE LIST($atResType;$1)
   For ($vlElem;1;Size of array($atResType))
      RESOURCE LIST($atResType{$vlElem};$alResID;$atResName;$1)
      $0:=$0+Size of array($alResID)
   End for
```

Once this project method is implemented in a database, you can write:

```
   $vhResFile:=Open resource file("")
   If (OK=1)
      ALERT("The file “"+Document+"” contains "+String(Count resources ($vhResFile))
                                                                  +" resource(s).")
      CLOSE RESOURCE FILE($vhResFile)
   End if
```

See Also

RESOURCE TYPE LIST.

System Variables and Sets

None is affected.

RESOURCE LIST

RESOURCE LIST (resType; resIDs; resNames{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resIDs	LongInt Array	←	Resource ID numbers for resources of this type
resNames	String Array	←	Resource names for resources of this type
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted

Description

The command RESOURCE LIST populates the arrays resIDs and resNames with the resource IDs and names of the resources whose type is passed in resType.

Important: You must pass a 4-character string in resType.

If you pass a valid resource file reference number in the optional parameter resFile, only the resources from that file are listed. If you do not pass the parameter resFile, all resources from the current open resource files are listed.

If you predeclare the arrays before calling RESOURCE LIST, you must predeclare resIDs as a Longint array and resNames as a String or Text array. If you do not predeclare the arrays, the command creates resIDs as a Longint array and resNames as a Text array.

After the call, you can test the number of resources found by applying the command Size of array to the array resIDs or resNames.

Examples

1. The following example populates the arrays $alResID and $atResName with the IDs and names of the string list resources present in the structure file of the database:

```
      If (On Windows)
         $vhStructureResFile:=Open resource file(Replace string(Structure
file;".4DB";".RSR"))
      Else
         $vhStructureResFile:=Open resource file(Structure file)
      End if
      If (OK=1)
⇒        RESOURCE LIST("STR#";$alResID;$atResName;$vhStructureResFile)
      End if
```

2. The following example copies the picture resources present in all currently open resource files into the Picture Library of the database:

```
⇒   RESOURCE LIST("PICT";$alResID;$atResName)
    Open window(50;50;550;120;5;"Copying PICT resources...")
    For ($vlElem;1;Size of array($alResID))
        GET PICTURE RESOURCE($alResID{$vlElem};$vgPicture)
        If (OK=1)
            $vsName:=$atResName{$vlElem}
            If ($vsName="")
                $vsName:="PICT resID="+String($alResID{$vlElem})
            End if
            ERASE WINDOW
            GOTO XY(2;1)
            MESSAGE("Adding picture “"+$vsName+"” to the DB Picture library.")
            SET PICTURE TO LIBRARY($vgPicture;$alResID{$vlElem};$vsName)
        End if
    End for
    CLOSE WINDOW
```

See Also

RESOURCE TYPE LIST.

System Variables and Sets

None is affected.

STRING LIST TO ARRAY

version 6.0

STRING LIST TO ARRAY (resID; strings{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
strings	String array	→	String or Text array to receive the strings
		←	Strings from the STR# resource
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted

Description

The command STRING LIST TO ARRAY populates the array strings with the strings stored in the string list ("STR#") resource whose ID is passed in resID.

If the resource is not found, the array strings is left unchanged and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Before calling STRING LIST TO ARRAY, you can predeclare the array strings as a String or Text array. If you do not predeclare the array, the command creates strings as a Text array.

Note: Each string of a string list resource can contain up to 255 characters.

Tip: Limit your use of string list resources to those up to 32K in total size, and a maximum of a few hundred strings per resource.

Example

See example for the command ARRAY TO STRING LIST.

See Also

ARRAY TO STRING LIST, Get indexed string, Get string resource, Get text resource.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

ARRAY TO STRING LIST

version 6.0

ARRAY TO STRING LIST (strings; resID{; resFile})

Parameter	Type		Description
strings	String array	→	String or Text array (new contents for the STR# resource)
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command ARRAY TO STRING LIST creates or rewrites the string list (“STR#”) resource whose ID is passed in resID. The contents of the resource are created from the strings passed in the array strings. The array can be a String or Text array.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top the resource files chain (the last resource file opened).

Note: Each string of a string list resource can contain up to 255 characters.

Tip: Limit your use of string list resources to resources no more than 32K in total size, and a maximum of a few hundred strings maximum per resource.

Example

Your database relies on a given set of fonts.

In the On Exit Database Method, you write:

```
     ` On Exit Database Method
  If (◊vbFontsAreOK)
     FONT LIST($atFont)
     $vhResFile:=Open resource file("FontSet")
     If (OK=1)
⇒       ARRAY TO STRING LIST($atFont;15000;$vhResFile)
        CLOSE RESOURCE FILE($vhResFile)
     End if
  End if
```

In the On Startup Database Method, you write:

```
	` On Startup Database Method
◊vbFontsAreOK:=False
FONT LIST($atNewFont)
If (Test path name("FontSet")#Is a document)
	$vhResFile:=Create resource file("FontSet")
Else
	$vhResFile:=Open resource file("FontSet")
End if
If (OK=1)
	STRING LIST TO ARRAY(15000;$atOldFont;$vhResFile)
	If (OK=1)
		◊vbFontsAreOK:=True
		For($vlElem;1;Size of array($atNewFont))
			If ($atNewFont{$vlElem}#$atOldFont{$vlElem}))
				$vlElem:=MAXLONG
				◊vbFontsAreOK:=False
			End if
		End for
	Else
		◊vbFontsAreOK:=True
	End if
	CLOSE RESOURCE FILE($vhResFile)
End if
If(Not(◊vbFontsAreOK))
	CONFIRM("You are not using the same font set, OK?")
	If(OK=1)
		◊vbFontsAreOK:=True
	Else
		QUIT 4D
	End if
End if
```

See Also

SET STRING RESOURCE, SET TEXT RESOURCE, STRING LIST TO ARRAY.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

Get indexed string

Get indexed string (resID; strID{; resFile}) → String

Parameter	Type		Description
resID	Number	→	Resource ID number
strID	Number	→	String number
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted
Function result	String	←	Value of the indexed string

Description

The command Get indexed string returns one of the strings stored in the string list ("STR#") resource whose ID is passed in resID.

You pass the number of the string in strID. The strings of a string list resource are numbered from 1 to N. To get all the strings (and their numbers) of a string list resource, use the command STRING LIST TO ARRAY.

If the resource or the string within the resource is not found, an empty string is returned and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Note: A string of a string list resource can contain up to 255 characters.

Example

See example for the command Month of.

See Also

Get string resource, Get text resource, STRING LIST TO ARRAY.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

Get string resource

Get string resource (resID{; resFile}) → String

Parameter	Type		Description
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted
Function result	String	←	Contents of the STR resource

Description

The command Get string resource returns the string stored in the string ("STR ") resource whose ID is passed in resID.

If the resource is not found, an empty string is returned and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Note: A string resource can contain up to 255 characters.

Example

The following example displays the contents of the string resource ID=20911, which must be located in at least one of the currently open resource files:

```
⇒	ALERT (Get string resource(20911))
```

See Also

Get indexed string, Get text resource, SET STRING RESOURCE, STRING LIST TO ARRAY.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

SET STRING RESOURCE

version 6.0

SET STRING RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	String	→	New contents for the STR resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command SET STRING RESOURCE creates or rewrites the string (“STR ”) resource whose ID is passed in resID with the string passed in resData.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top the resource files chain (the last resource file opened).

Note: A string resource can contain up to 255 characters.

See Also

Get string resource, SET TEXT RESOURCE.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

Get text resource

Get text resource (resID{; resFile}) → Text

Parameter	Type		Description
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted
Function result	Text	←	Contents of the TEXT resource

Description

The command Get text resource returns the text stored in the text (“TEXT”) resource whose ID is passed in resID.

If the resource is not found, empty text is returned, and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Note: A text resource can contain up to 32,000 characters.

Example

The following example displays the contents of the text resource ID=20800, which must be located in at least one of the currently open resource files:

```
⇒	ALERT (Get text resource(20800))
```

See Also

Get indexed string, Get string resource, SET TEXT RESOURCE, STRING LIST TO ARRAY.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

version 6.0

SET TEXT RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	String	→	New contents for the TEXT resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command SET TEXT RESOURCE creates or rewrites the text (“TEXT”) resource whose ID is passed in resID with the text or string passed in resData.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top the resource files chain (the last resource file opened).

Note: A text resource can contain up to 32,000 characters.

See Also

Get text resource, SET STRING RESOURCE.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

GET PICTURE RESOURCE

GET PICTURE RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	Field or Var.	→	Picture field or variable to receive the picture
		←	Contents of the PICT resource
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted

Description

The command GET PICTURE RESOURCE returns in the picture field or variable resData the picture stored in the picture ("PICT") resource whose ID is passed in resID.

If the resource is not found, the resData parameter is left unchanged, and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Note: A picture resource can be at least several megabytes in size.

Example

See example for the command RESOURCE LIST.

See Also

GET ICON RESOURCE, ON ERR CALL, SET PICTURE RESOURCE.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

Error Handling

If there is not enough memory to load the picture, an error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

SET PICTURE RESOURCE

version 6.0

SET PICTURE RESOURCE (resID; resData{; resFile})

Parameter	Type		Description
resID	Number	→	Resource ID number
resData	Picture	→	New contents for the PICT resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command SET PICTURE RESOURCE creates or rewrites the picture (“PICT”) resource whose ID is passed in resID with the picture passed in resData.

If the resource cannot be added, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top of the resource files chain (the last resource file opened).

Note: A picture resource can be several megabytes in size and even more.

See Also

GET PICTURE RESOURCE.

System Variables and Sets

If the resource has been written, OK is set to 1. Otherwise, it is set to 0 (zero).

GET ICON RESOURCE

version 6.0

GET ICON RESOURCE (resID; resData{; fileRef})

Parameter	Type		Description
resID	Number	→	Icon resource ID number
resData	Picture	→	Picture field or variable to receive the picture
		←	Contents of the cicn resource
fileRef	Number	→	Resource file reference number, or all open resource files, if omitted

Description

The command GET ICON RESOURCE returns, in the picture field or variable resData, the icon stored in the color icon (“cicn”) resource whose ID is passed in resID.

If the resource is not found, the resData parameter is left unchanged and the OK variable is set to 0 (zero).

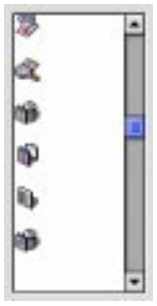
If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Example

The following example loads, in a Picture array, the color icons located in the active 4D application:

```
     If (On Windows)
        $vh4DResFile:=Open resource file(Replace string(Application file;".EXE";".RSR"))
     Else
        $vh4DResFile:=Open resource file(Application file)
     End if
     RESOURCE LIST("cicn";$alResID;$asResName;$vh4DResFile)
     $vlNbIcons:=Size of array($alResID)
     ARRAY PICTURE(ag4DIcon;$vlNbIcons)
     For ($vlElem;1;$vlNbIcons)
⇒       GET ICON RESOURCE($alResID{$vlElem};ag4DIcon{$vlElem};$vh4DResFile)
     End for
```

After this code has been executed, the array looks like this when displayed in a form:



See Also

GET PICTURE RESOURCE.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

GET RESOURCE

version 6.0

GET RESOURCE (resType; resID; resData{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resData	BLOB	→	BLOB field or variable to receive the data
		←	Contents of the resource
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted

Description

The command GET RESOURCE returns in the BLOB field or variable resData the contents of the resource whose type and ID is passed in resType and resID.

Important: You must pass a 4-character string in resType.

If the resource is not found, the resData parameter is left unchanged and the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is searched for in that file only. If you do not pass resFile, the first occurrence of the resource found in the resource files chain is returned.

Note: A resource can be at least several megabytes in size.

Platform independence: Remember that you are working with MacOS-based resources. No matter what the platform, internal resource data such as Long Integer is stored using Macintosh byte ordering. On Windows, the data for standard resources (such as string list and pictures resources) is automatically byte swapped when necessary. On the other hand, if you create and use your own internal data structures, it is up to you to byte swap the data you extract from the BLOB (i.e., passing Macintosh byte ordering to a command such BLOB to longint).

Example

See the example for the command SET RESOURCE.

See Also

BLOB Commands, Resources, SET RESOURCE.

System Variables and Sets

If the resource is found, OK is set to 1. Otherwise, it is set to 0 (zero).

Error Handling

If there is not enough memory to load the resource, an error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

SET RESOURCE

version 6.0

SET RESOURCE (resType; resID; resData{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resData	BLOB	→	New contents for the resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command SET RESOURCE creates or rewrites the resource whose type and ID is passed in resType and resID with the data passed in the BLOB resData.

Important: You must pass a 4-character string in resType.

If the resource cannot be written, the OK variable is set to 0 (zero).

If you pass a valid resource file reference number in resFile, the resource is added to that file. If you do not pass resFile, the resource is added to the file at the top of the resource files chain (the last resource file opened).

Note: A resource can be at least several megabytes in size.

Platform independence: Remember that you are working with MacOS-based resources. No matter what the platform, internal resource data such as Long Integer is stored using Macintosh byte ordering. On Windows, the data for standard resources (such as string list and pictures resources) is automatically byte swapped when necessary. On the other hand, if you create and use your own internal data structures, it it up to you to byte swap the data you write into the BLOB (i.e., passing Macintosh byte ordering to a command such LONGINT TO BLOB).

Example

During a 4D session you maintain some user preferences in interprocess variables. To save these preferences from session to session, you can:

1. Use the commands SAVE VARIABLES and LOAD VARIABLES to store and retrieve the variables in variable documents on disk.

2. Use the commands VARIABLE TO BLOB, BLOB TO DOCUMENT, DOCUMENT TO BLOB and BLOB TO VARIABLE to store and retrieve the variables in BLOB documents on disk.

3. Use the commands VARIABLE TO BLOB, SET RESOURCE, GET RESOURCE and BLOB TO VARIABLE to to store and retrieve the variables in resource files on disk.

The following is an example of the third method. In the On Exit Database Method you write:

```
    ` On Exit Database Method
If (Test path name("DB_Prefs")#Is a document)
    $vhResFile:=Create resource file("DB_Prefs")
Else
    $vhResFile:=Open resource file("DB_Prefs")
End if
If (OK=1)
    VARIABLE TO BLOB(◊vbAutoRepeat;$vxPrefData)
    VARIABLE TO BLOB(◊vlCurTable;$vxPrefData;*)
    VARIABLE TO BLOB(◊asDfltOption;$vxPrefData;*)
        ` and so on...
⇒   SET RESOURCE("PREF";26500;$vxPrefData;$vhResFile)
    CLOSE RESOURCE FILE($vhResFile)
End if
```

In the On Startup Database Method you write:

```
    ` On Startup Database Method
C_BOOLEAN(◊vbAutoRepeat)
C_LONGINT(◊vlCurTable)
$vbDone:=False
$vhResFile:=Open resource file("DB_Prefs")
If (OK=1)
⇒   GET RESOURCE("PREF";26500;$vxPrefData;$vhResFile)
    If (OK=1)
        $vlOffset:=0
        BLOB TO VARIABLE($vxPrefData;◊vbAutoRepeat;$vlOffset)
        BLOB TO VARIABLE($vxPrefData;◊vlCurTable;$vlOffset)
        BLOB TO VARIABLE($vxPrefData;◊asDfltOption;$vlOffset)
            ` and so on...
        $vbDone:=False
    End if
    CLOSE RESOURCE FILE($vhResFile)
End if
If(Not($vbDone))
    ◊vbAutoRepeat:=False
    ◊vlCurTable:=0
    ARRY STRING(127;◊asDfltOption;0)
End if
```

See Also

BLOB Commands, GET RESOURCE.

System Variables and Sets

If the resource is written, OK is set to 1. Otherwise, it is set to 0 (zero).

Get resource name

version 6.0

Get resource name (resType; resID{; resFile}) → String

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted
Function result	String	←	Name of the resource

Description

The command Get resource name returns the name of the resource whose type is passed in resType and whose ID number is passed in resID.

If you pass a valid resource file reference number in the parameter resFile, the resource is searched for within that file only. If you do not pass the parameter resFile, the resource is searched for within the current open resource files.

If the resource does not exist, Get resource name returns an empty string and sets the OK variable to 0 (zero).

Example

The following project method copies a resource, and its resource name and attributes, from one resource file to another:

```
      ` COPY RESOURCE Project Method
      ` COPY RESOURCE ( String ; Long ; Time ; Time )
      ` COPY RESOURCE ( resType ; resID ; srcResFile ; dstResFile )
   C_STRING (4;$1)
   C_LONGINT ($2)
   C_TIME ($3;$4)
   C_BLOB ($vxResData)
   GET RESOURCE ($1;$2;$vxData;$3)
   If (OK=1)
      SET RESOURCE ($1;$2;$vxData;$4)
      If (OK=1)
⇒        SET RESOURCE NAME ($1;$2; Get resource name ($1;$2;$3);$4)
         SET RESOURCE PROPERTIES ($1;$2; Get resource properties ($1;$2;$3);$4)
      End if
   End if
```

Once this project method is present in your application, you can write:

```
		` Copy the resource 'DATA' ID = 15000 from file A to file B
	COPY RESOURCE ("DATA";15000;$vhResFileA;$vhResFileB)
```

See Also

SET RESOURCE PROPERTIES.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist; otherwise, it is set to 1.

SET RESOURCE NAME

version 6.0

SET RESOURCE NAME (resType; resID; resName{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resName	String	→	New name for the resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command SET RESOURCE NAME changes the name of the resource whose type is passed in resType and whose ID number is passed in resID.

If you pass a valid resource file reference number in the parameter resFile, the resource is searched for within that file only. If you do not pass the parameter resFile, the resource is searched for within the current open resource files.

If the resource does not exist, SET RESOURCE NAME does nothing and sets the OK variable to 0 (zero).

WARNING: DO NOT change the names of resources that belong to 4D or to any System files. If you do so, you may provoke undesired system errors.

Note: Resource names can be up to 255 characters in length. They are not case sensitive, but are diacritical sensitive.

Example

See example for the command Get resource name.

See Also

SET RESOURCE PROPERTIES.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist, otherwise it is set to 1.

Get resource properties

version 6.0

Get resource properties (resType; resID{; resFile}) → Number

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or all open resource files, if omitted
Function result	Number	←	Resource attributes

Description

The command Get resource properties returns the attributes of the resource whose type is passed in resType and whose ID number is passed in resID.

If you pass a valid resource file reference number in the parameter resFile, the resource is searched for within that file only. If you do not pass the parameter resFile, the resource is searched for within the current open resource files.

If the resource does not exist, Get resource properties returns 0 (zero) and sets the OK variable to 0 (zero).

The numeric value returned by Get resource properties must be seen as a bit field value whose bits have special meaning. For a description of the resource attributes and their effects, please refer to the command SET RESOURCE PROPERTIES.

Example

See example for the command Get resource name.

See Also

SET RESOURCE NAME.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist; otherwise, it is set to 1.

SET RESOURCE PROPERTIES

version 6.0

SET RESOURCE PROPERTIES (resType; resID; resAttr{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resAttr	Number	→	New attributes for the resource
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command SET RESOURCE PROPERTIES changes the attributes of the resource whose type is passed in resType and whose ID number is passed in resID.

If you pass a valid resource file reference number in the parameter resFile, the resource is searched for within that file only. If you do not pass the parameter resFile, the resource is searched for within the current open resource files.

If the resource does not exist, SET RESOURCE PROPERTIES does nothing and sets the OK variable to 0 (zero).

The numeric value you pass in resAttr must be seen as a bit field value whose bits have special meaning. The following predefined constants are provided by 4th Dimension:

Constant	Type	Value
System heap resource mask	Long Integer	64
System heap resource bit	Long Integer	6
Purgeable resource mask	Long Integer	32
Purgeable resource bit	Long Integer	5
Locked resource mask	Long Integer	16
Locked resource bit	Long Integer	4
Protected resource mask	Long Integer	8
Protected resource bit	Long Integer	3
Preloaded resource mask	Long Integer	4
Preloaded resource bit	Long Integer	2
Changed resource mask	Long Integer	2
Changed resource bit	Long Integer	1

Using these constants, you can build any resource attributes value. See examples below.

Resource Attributes and Their Effects

• System heap
If this attribute is set, the resource will be loaded into the system memory rather than into 4D memory. You should not use this attribute, unless you **really** know what you are doing.

• Purgeable
If this attribute is set, after the resource has been loaded, you can purge it from memory if space is required for allocation of other data. Since you load resources into 4D BLOBs, it is a good idea to have all your own resources purgeable in order to reduce memory usage. However, if you frequently access this resource during a working session, you might want to make it non-purgeable in order to reduce disk access due to frequent reloading of a purged resource.

• Locked
If this attribute is set, you will not be able to relocate the resource (unmovable) after it is loaded into memory. A locked resource cannot be purged even if it is purgeable. Locking a resource has the undesirable effect of fragmenting the memory space. DO NOT use this attribute, unless you **really** know what you are doing.

• Protected
If this attribute is set, you can no longer change the name, ID number or the contents of a the resource. You can no longer delete this resource. However, you can call SET RESOURCE PROPERTIES to clear this attribute; then you can again modify or delete the resource. Most of the time, you will not use this attribute. Note: This attribute has no effect on Windows.

• Preloaded
If this attribute is set, the resource is automatically loaded into memory if the resource file where it is located is open. This attribute is useful for optimizing resource loading when a resource file is opened. Most of the time, you will not use this attribute.

• Changed
If this attribute is set, the resource is marked as “must be saved on disk” when the resource file where it is located is closed. Since the 4D command SET RESOURCE handles the writing and rewriting of resources internally, you should not use this attribute, unless you **really** know what you are doing.

You will usually use the attribute purgeable and, more rarely, Preloaded and Protected.

WARNING: DO NOTchange the attributes of resources that belong to 4D or to any System files. If you do so, you may provoke undesired system errors.

Examples

1. See example for the command Get resource name.

2. The following example makes the resource 'STR#' ID=17000 purgeable, but leaves the other attributes unchanged:

```
$vlResAttr:=Get resource properties ('STR#';17000;$vhResFile)
SET RESOURCE PROPERTIES('STR#';17000;
                              $vlResAttr ?+ Purgeable resource bit;$vhMyResFile)
```

3. The following example makes the resource 'STR#' ID=17000 preloaded and non purgeable:

```
SET RESOURCE PROPERTIES('STR#';17000;Preloaded resource mask;$vhResFile)
```

4. The following example makes the resource 'STR#' ID=17000 preloaded but purgeable:

```
SET RESOURCE PROPERTIES('STR#';17000;
                    Preloaded resource mask+Purgeable resource mask;$vhResFile)
```

See Also

SET RESOURCE NAME.

System Variables or Sets

The OK variable is set to 0 if the resource does not exist; otherwise, it is set to 1.

DELETE RESOURCE

version 6.0

DELETE RESOURCE (resType; resID{; resFile})

Parameter	Type		Description
resType	String	→	4-character resource type
resID	Number	→	Resource ID number
resFile	DocRef	→	Resource file reference number, or current resource file, if omitted

Description

The command DELETE RESOURCE deletes the resource whose type is passed in resType and whose ID number is passed in resID.

If you pass a valid resource file reference number in the parameter resFile, the resource is searched for within that file only. If you do not pass resFile, the resource is searched for within the current open resource files.

If the resource does not exist, DELETE RESOURCE does nothing and sets the OK variable to 0 (zero). If the resource is found and deleted, the OK variable is set to 1.

WARNING: DO NOT delete resources that belong to 4D or to any System files. If you do so, you may provoke undesired system errors.

Examples

1. The following example deletes the resource "STR#" ID=20000:

```
          ` Note that this example will delete the first "STR#" ID=20000 resource
          ` found in any resource file currently open:
⇒     DELETE RESOURCE ("STR#";20000)
```

2. The following example deletes the resource "STR#" ID=20000 if it is found in a specified resource file:

```
          ` Note that this example will delete the resource "STR#" ID=20000
          ` only if it is present in the resource file specified by $vhResFile:
⇒     DELETE RESOURCE ("STR#";20000;$vhResFile)
          ` Note also that if there is such a resource in a currently open
          ` resource file other than that specified by $vhResFile, this resource
          ` is left untouched
```

3. The project method DELETE RESOURCES OF TYPE deletes all the resources of the type specified (as the second parameter) from the resource file specified (as the first parameter):

```
     ` DELETE RESOURCES OF TYPE Project Method
     ` DELETE RESOURCES OF TYPE ( Time ; String )
     ` DELETE RESOURCES OF TYPE ( resFile ; resType )

   C_TIME($1)
   C_STRING(4;$2)

   RESOURCE LIST($2;$aiResID;$asResName;$1)
   If(OK=1)
      For($vlElem;1;Size of array($aiResID))
⇒        DELETE RESOURCE($2;$aiResID{$vlElem};$1)
      End for
   End if
```

After this project method is present in a database, you can write:

```
     ` Delete all the resource of type "PREF" from the resource file $vhResFile
   DELETE RESOURCES OF TYPE ($vhResFile;"PREF")
```

4. The project method DELETE RESOURCE BY NAME deletes a resource (of a specific type) whose name is known:

```
     ` DELETE RESOURCE BY NAME Project Method
     ` DELETE RESOURCE BY NAME ( Time ; String ; String )
     ` DELETE RESOURCE BY NAME ( resFile ; resType ; resName )

   C_TIME($1)
   C_STRING(4;$2)
   C_STRING(255;$3)

   RESOURCE LIST($2;$aiResID;$asResName;$1)
   If(OK=1)
      $vlElem:=Find in array($asResName;$3)
      If($vlElem>0)
⇒        DELETE RESOURCE($2;$aiResID{$vlElem};$1)
      End for
   End if
```

After this project method is present in a database, you can write:

```
		` Delete, from the resource file $vhResFile, the resource "PREF" whose name is
“Standard Settings”:
	DELETE RESOURCE BY NAME ($vhResFile;"PREF";"Standard Settings")
```

See Also

RESOURCE LIST, SET RESOURCE PROPERTIES.

System Variables and Sets

The OK variable is set to 0 if the resource does not exist. If the resource has been deleted, the OK variable is set to 1.

41 Selection

ALL RECORDS

ALL RECORDS {(table)}

Parameter	Type		Description
table	Table	→	Table for which to select all records, or Default table, if omitted

Description

ALL RECORDS selects all the records of table for the current process. ALL RECORDS makes the first record the current record and loads the record from disk. ALL RECORDS returns the records to the default record order, which is the order in which the records are stored on disk.

Example

The following example displays all the records from the [People] table:

```
⇒	ALL RECORDS ([People])  ` Select all the records in the table
	DISPLAY SELECTION ([People])  ` Display records in output form
```

See Also

DISPLAY SELECTION, MODIFY SELECTION, ORDER BY, QUERY, Records in selection, Records in table.

Records in selection

Records in selection {(table)} → Number

Parameter	Type		Description
table	Table	→	Table for which to return number of selected records, or Default table, if omitted
Function result	Number	←	Records in selection of table

Description

Records in selection returns the number of records in the current selection of table. In contrast, Records in table returns the total number of records in the table.

Example

The following example shows a loop technique commonly used to move through all the records in a selection. The same action can also be accomplished with the APPLY TO SELECTION command:

```
   FIRST RECORD ([People])   ` Start at first record in the selection
⇒  For ($vlRecord; 1; Records in selection ([People]))   ` Loop once for each record
      Do Something   ` Do something with the record
      NEXT RECORD ([People])   ` Move to the next record
   End for
```

See Also

Records in table.

DELETE SELECTION

Selection

version 3

DELETE SELECTION {(table)}

Parameter	Type		Description
table	Table	→	Table for which to delete the current selection, or Default table, if omitted

Description

DELETE SELECTION deletes the current selection of records from table. If the current selection is empty, DELETE SELECTION has no effect. After the records are deleted, the current selection is empty. Records that are deleted during a transaction are locked to other users and other processes until the transaction is validated or canceled.

Warning: Deleting a selection of records is a permanent operation, and cannot be undone.

The Completely Delete option in the Table Properties dialog box allows you to increase the speed of deletions when DELETE SELECTION is used.

Examples

1. The following example displays all the records from the [People] table and allows the user to select which ones to delete. The example has two sections. The first is a method to display the records. The second is an object method for a Delete button. Here is the first method:

```
ALL RECORDS ([People])  ` Select all records
OUTPUT FORM ([People]; "Listing") ` Set the form to list the records
DISPLAY SELECTION ([People])  ` Display all records
```

The following is the object method for the Delete button, which appears in the Footer area of the output form. The object method uses the records the user selected (the UserSet) to delete the selection. Note that if the user did not select any records, DELETE SELECTION has no effect.

```
      ` Confirm that the user really wants to delete the records
   CONFIRM("You selected "+String(Records in set ("UserSet"))+" people to delete."
                                             +Char(13)+"Click OK to Delete them.")
   If (OK=1)
      USE SET ("UserSet") ` Use the records chosen by the user
⇒     DELETE SELECTION([People]) ` Delete the selection of records
   End if
   ALL RECORDS ([People]) ` Select all records
```

2. If a locked record is encountered during the execution of DELETE SELECTION, that record is not deleted. Any locked records are put into a set called LockedSet. After DELETE SELECTION has executed, you can test the LockedSet to see if any records were locked. The following loop will execute until all the records have been deleted:

```
        Repeat  ` Repeat for any locked records
⇒           DELETE SELECTION([ThisTable])
            USE SET ("LockedSet") ` Select only the locked records
        Until (Records in set("LockedSet")=0) ` Until there are no more locked records
```

See Also

DISPLAY SELECTION, MODIFY SELECTION, Record Locking, Sets.

Selected record number

Selection

version 3

Selected record number {(table)} → Number

Parameter	Type		Description
table	Table	→	Table for which to return the selected record number or Default table, if omitted
Function result	Number	←	Selected record number of current record

Description

Selected record number returns the position of the current record within the current selection of table.

If the selection not is empty and if the current record is within the selection, Selected record number returns a value between 1 and Records in selection. If the selection is empty, of if there is no current record, it returns 0 (zero).

The selected record number is not the same as the number returned by Record number, which returns the physical record number in the table. The selected record number depends on the current selection and the current record.

Example

The following example saves the current selected record number in a variable:

```
⇒    CurSelRecNum:=Selected record number([People]) ` Get the selected record number
```

See Also

About Record Numbers, GOTO SELECTED RECORD, Records in selection.

GOTO SELECTED RECORD

GOTO SELECTED RECORD ({table; }record)

Parameter	Type		Description
table	Table	→	Table in which to go to the selected record, or Default table, if omitted
record	Number	→	Position of record in the selection

Description

GOTO SELECTED RECORD moves to the specified record in the current selection of table and makes that record the current record. The current selection does not change. The record parameter is not the same as the number returned by Record number; it represents the record's position in the current selection. The record's position depends on how the selection is made and whether or not the selection is sorted.

If there are no records in the current selection, or the number is not in the selection, then GOTO SELECTED RECORD does nothing.

Example

The following example loads data from the field [People]Last Name into the atNames. An array of long integers, called alRecNum, is filled with numbers that will represent the selected record numbers. Both arrays are then sorted:

```
    ` Make any selection for the [People] table here
    ` ...
    ` Get the names
SELECTION TO ARRAY ([People]Last Name;atNames)
    ` Create an array for the selected record numbers
$vlNbRecords:=Size of array (atNames)
ARRAY LONGINT (alRecNum;$vlNbRecords)
For ($vlRecord; 1; $vlNbRecords)
    alRecNum{$vlRecord}:=$vlRecord
End for
    ` Sort the arrays in alphabetical order
SORT ARRAY (atNames; alRecNum; >)
```

If the array atNames is displayed in a scrollable area, the user can click one of the items. Since the sorting of the two arrays is synchronized, any element in alRecNum provides the selected record number for the record whose name is stored in the corresponding element in atNames.

The following object method for atNames selects the correct record in the [People] selection, according to the name chosen in the scrollable area:

```
        Case of
            : (Form event=On Clicked)
                If (atNames#0)
⇒                   GOTO SELECTED RECORD (alRecNum{atNames})
                End if
        End case
```

See Also

Selected record number.

FIRST RECORD

Selection

version 3

FIRST RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to move to the first selected record, or Default table, if omitted

Description

FIRST RECORD makes the first record of the current selection of table the current record, and loads the record from disk. All query, selection, and sorting commands also set the current record to the first record. If the current selection is empty, FIRST RECORD has no effect.

This command is most often used after the USE SET command to begin looping through a selection of records from the first record. However, you can also call it from a subroutine if you are not sure whether or not the current record is actually the first.

Example

The following example makes the first record of the [Customers] table the first record:

⇒ **FIRST RECORD** ([Customers])

See Also

Before selection, End selection, LAST RECORD, NEXT RECORD, PREVIOUS RECORD.

NEXT RECORD

Selection

version 3

NEXT RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to move to the next selected record, or Default table, if omitted

Description

NEXT RECORD moves the current record pointer to the next record in the current selection of table for the current process. If the current selection is empty, or if Before selection or End selection is TRUE, NEXT RECORD has no effect.

If NEXT RECORD moves the current record pointer past the end of the current selection, End selection returns TRUE, and there is no current record. If End selection returns TRUE, use FIRST RECORD, LAST RECORD, or GOTO SELECTED RECORD to move the current record pointer back into the current selection.

Example

See the example for DISPLAY RECORD.

See Also

Before selection, End selection, FIRST RECORD, LAST RECORD, PREVIOUS RECORD.

LAST RECORD

LAST RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to move to the last selected record, or Default table, if omitted

Description

LAST RECORD makes the last record of the current selection of table the current record and loads the record from disk. If the current selection is empty, LAST RECORD has no effect.

Example

The following example makes the last record of the [People] table the current record:

```
⇒	LAST RECORD ([People])
```

See Also

Before selection, End selection, FIRST RECORD, NEXT RECORD, PREVIOUS RECORD.

PREVIOUS RECORD

PREVIOUS RECORD {(table)}

Parameter	Type		Description
table	Table	→	Table for which to move to the previous selected record, or Default table, if omitted

Description

PREVIOUS RECORD moves the current record pointer to the previous record in the current selection of table for the current process. If the current selection is empty, or if Before selection or End selection is TRUE, PREVIOUS RECORD has no effect.

If PREVIOUS RECORD moves the current record pointer before the current selection, Before selection returns TRUE, and there is no current record. If Before selection returns TRUE, use FIRST RECORD, LAST RECORD, or GOTO SELECTED RECORD to move the current record pointer back into the current selection.

See Also

Before selection, End selection, FIRST RECORD, LAST RECORD, NEXT RECORD.

Before selection

version 3

Before selection {(table)} → Boolean

Parameter	Type		Description
table	Table	→	Table for which to test if record pointer is before the first selected record, or Default table, if omitted
Function result	Boolean	←	Yes (TRUE) or No (FALSE)

Description

Before selection returns TRUE when the current record pointer is before the first record of the current selection of table. Before selection is commonly used to check whether or not PREVIOUS RECORD has moved the current record pointer before the first record. If the current selection is empty, Before selection returns TRUE.

To move the current record pointer back into the selection, use LAST RECORD, FIRST RECORD, or GOTO SELECTED RECORD. NEXT RECORD does not move the pointer back into the selection.

Before selection also returns TRUE in the first header when a report is being printed with PRINT SELECTION or from the Print menu. You can use the following code to test for the first header and print a special header for the first page:

```
	` Method of a form being used as output form for a summary report
$vpFormTable:=Current form table
Case of
		` ...
	: (Form event=On Printing Header)
			` A header area is about to be printed
		Case of
⇒			: (Before selection($vpFormTable->))
				` Code for the first break header goes here
				` ...
		End case
End case
```

Example

This form method is used during the printing of a report. It sets a variable, vTitle, to print in the Header area on the first page:

```
      ` [Finances];"Summary" Form Method
    Case of
        ` ...
      : (Form event=On Printing Header)
        Case of
⇒           : (Before selection([Finances]))
              vTitle := "Corporate Report 1997"  ` Set the title for the first page
            Else
              vTitle := ""  ` Clear the title for all other pages
        End case
    End case
```

See Also

End selection, FIRST RECORD, Form event, PREVIOUS RECORD, PRINT SELECTION.

End selection

End selection {(table)} → Boolean

Parameter	Type		Description
table	Table	→	Table for which to test if record pointer is beyond the last selected record, or Default table, if omitted
Function result	Boolean	←	Yes (TRUE) or No (FALSE)

Description

End selection returns TRUE when the current record pointer is beyond the last record of the current selection of table. End selection is commonly used to check whether or not NEXT RECORD has moved the current record pointer past the last record. If the current selection is empty, End selection returns TRUE.

To move the current record pointer back into the selection, use LAST RECORD, FIRST RECORD, or GOTO SELECTED RECORD. PREVIOUS RECORD does not move the pointer back into the selection.

End selection also returns TRUE in the last footer when a report is being printed with PRINT SELECTION or from the Print menu. You can use the following code to test for the last footer and print a special footer for the last page:

```
        ` Method of a form being used as output form for a summary report
    $vpFormTable:=Current form table
    Case of
            ` ...
⇒       : (Form event=On Printing Footer)
                ` A footer is about to be printed
            If(End selection($vpFormTable->))
                ` Code for the last footer goes here
            Else
                ` Code for a footer goes here
            End if
    End case
```

Example

This form method is used during the printing of a report. It sets the variable vFooter to print in the Footer area on the last page:

```
          ` [Finances];"Summary" Form Method
      Case of
             ` ...
⇒         : (Form event=On Printing Footer)
             If(End selection([Finances]))
                vFooter := "©1997 Acme Corp."  ` Set the footer for the last page
             Else
                vFooter := ""  ` Clear the footer for all other pages
             End if
      End case
```

See Also

Before selection, Form event, LAST RECORD, NEXT RECORD, PRINT SELECTION.

DISPLAY SELECTION

DISPLAY SELECTION ({table}{; *}{; *})

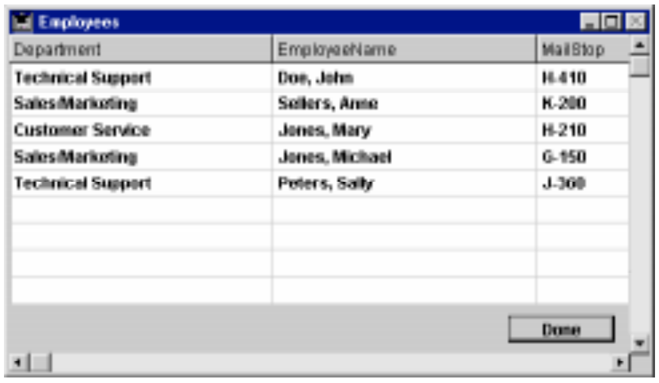
Parameter	Type		Description
table	Table	→	Table to display, or Default table, if omitted
*		→	Use output form for one record selection and hide scroll bars in the input form
*		→	Show scroll bars in the input form (overrides second option of first optional *)

Description

DISPLAY SELECTION displays the selection of table, using the output form. The records are displayed in a scrollable list similar to the User environment's list. If the user double-clicks a record, the record is displayed in the input form. The list is displayed in the frontmost window.

To display a selection and also modify a record after you have double-clicked on it (as you do in the User environment window), use MODIFY SELECTION instead of DISPLAY SELECTION.
All of the following information applies to both commands, except for the information on modifying records.

The following figure shows an output form displayed by the DISPLAY SELECTION command.



Department	EmployeeName	MailStop
Technical Support	Doe, John	H-410
Sales/Marketing	Sellers, Anne	K-200
Customer Service	Jones, Mary	H-210
Sales/Marketing	Jones, Michael	G-150
Technical Support	Peters, Sally	J-360

After DISPLAY SELECTION is executed, there may not be a current record. Use a command such as FIRST RECORD or LAST RECORD to select one.

DISPLAY SELECTION does not allow the user to modify a record when in the input form. MODIFY SELECTION does.

Some rules regarding the optional * parameter:
- If the selection contains only one record and the first optional * is not used, the record appears in the input form instead of the output form.
- If the first optional * is specified, a one-record selection is displayed, using the output form.
- If the first optional * is specified and the user displays the record in the input form by double-clicking on it, the scroll bars will be hidden in the input form. To reverse this effect, pass the second optional *.

A button labeled Done is automatically included at the bottom of the list. Adding any variable or active object on the form removes the Done button. Clicking this button exits the command. Custom buttons may be used instead; you can put these buttons in the Footer area of the output form. You can use automatic Accept or Cancel buttons to exit, or use an object method that calls ACCEPT or CANCEL.

The user can scroll through the selection and click a record to select it. If the user clicks a different record, the first record is deselected and the second record is selected. A user can select a group of contiguous records by clicking the first record and Shift+clicking (Windows or Macintosh) the last record. To select records that are not adjacent, the user can Ctrl+click (Windows) or Command-click (Macintosh) each record.

During and after execution of DISPLAY SELECTION, the records that the user highlighted (selected) are kept in a set named UserSet. The UserSet is available within the selection display for object methods when a button is clicked or a menu item is chosen. It is also available to the project method that called DISPLAY SELECTION after the command completed.

Examples

1. The following example selects all the records in the [People] table. It then uses DISPLAY SELECTION to display the records, and allows the user to select the records to print. Finally, it selects the records with USE SET, and prints them with PRINT SELECTION:

```
	ALL RECORDS([People])  ` Select all records
⇒	DISPLAY SELECTION ([People]; *)  ` Display the records
	USE SET ("UserSet")  ` Use only records picked by user
	PRINT SELECTION ([People])  ` Print the records that the user picked
```

2. See example #6 for the command Form event. This example shows all the tests you may need to check in order to fully monitor the events that occur during a DISPLAY SELECTION.

3. To reproduce the functionality provided by, for example, the Queries menu of the User environment when you use DISPLAY SELECTION or MODIFY SELECTION in the Custom Menus environment, proceed as follows:

a. In the Design environment, create a menu bar with the menu commands you want, for example, Show All, Query and Order By.

b. Associate this menu bar (using a negative menu bar number) with the output form used with DISPLAY SELECTION or MODIFY SELECTION.

c. Associate the following project methods to your menu commands:

```
   ` M_SHOW_ALL (attached to menu item Show All)
$vpCurTable:=Current form table
ALL RECORDS($vpCurTable->)

   ` M_QUERY (attached to menu item Query)
$vpCurTable:=Current form table
QUERY($vpCurTable->)

   ` M_ORDDER_BY (attached to menu item Order By)
$vpCurTable:=Current form table
ORDER BY($vpCurTable->)
```

You can also use other commands, such as PRINT SELECTION, REPORT, and so on, to provide all the "standard" menu options you may want each time you display or modify a selection in the Custom Menus environment. Thanks to the command Current form table, these methods are generic, and the menu bar they support can be attached to any output form of any table.

See Also

Form event, MODIFY SELECTION, Sets.

MODIFY SELECTION

MODIFY SELECTION ({table}{; *}{; *})

Parameter	Type		Description
table	Table	→	Table to display and modify, or Default table, if omitted
*		→	Use output form for one record selection and hide scroll bars in the input form
*		→	Show scroll bars in the input form (overrides second option of first optional *)

Description

MODIFY SELECTION does almost the same thing as DISPLAY SELECTION. Refer to the description of DISPLAY SELECTION for details. The differences between the two commands are:

1. DISPLAY SELECTION enables you to display the current selected records in list mode, or in the input form when you double-click on a record. Using MODIFY SELECTION, you can modify a record when you double-click on it, if it is not already in use by another process or user.

2. DISPLAY SELECTION automatically switches the table to read-only. MODIFY SELECTION automatically switches the table to read-write. Both commands restore the table state after they have completed execution.

See Also

DISPLAY SELECTION, Form event, Sets.

APPLY TO SELECTION

Selection

version 3

APPLY TO SELECTION ({table; }statement)

Parameter	Type		Description
table	Table	→	Table for which to apply statement, or Default table, if omitted
statement	Statement	→	One line of code or a method

Description

APPLY TO SELECTION applies statement to each record in the current selection of table. The statement can be a statement or a method. If statement modifies a record of table, the modified record is saved. If statement does not modify a record, the record is not saved. If the current selection is empty, APPLY TO SELECTION has no effect. If the relation is automatic, the statement can contain a field from a related table.

APPLY TO SELECTION can be used to gather information from the selection of records (for example, a total), or to modify a selection (for example, changing the first letter of a field to uppercase). If this command is used within a transaction, all changes can be undone if the transaction is canceled.

4D Server: The server does not execute any of the commands that may be passed in statement. Every record in the selection will be sent back to the local workstation to be modified.

The progress thermometer is displayed while APPLY TO SELECTION is executing. To hide it, use MESSAGES OFF prior to the call to APPLY TO SELECTION. If the progress thermometer is displayed, the user can cancel the operation.

Examples

1. The following example changes all the names in the table [Employees] to uppercase:

```
⇒	APPLY TO SELECTION([Employees];
				[Employees]Last Name:=Uppercase([Employees]Last Name))
```

2. If a record is locked during execution of APPLY TO SELECTION and that record is modified, the record will not be saved. Any locked records that are encountered are put in a set called LockedSet. After APPLY TO SELECTION has executed, test LockedSet to see if any records were locked. The following loop will execute until all records have been modified:

```
        Repeat
⇒           APPLY TO SELECTION([Employees];
                                        [Employees]Last Name:=Uppercase([Employees]Last Name))
            USE SET ("LockedSet")  ` Select only locked records
        Until (Records in set ("LockedSet") = 0)  ` Done when there are no locked records
```

System Variables or Sets

If the user clicks the Stop button in the progress thermometer, the OK system variable is set to 0. Otherwise, the OK system variable is set to 1.

See Also

Sets.

REDUCE SELECTION

Selection

version 3

REDUCE SELECTION ({table; }number)

Parameter	Type		Description
table	Table	→	Table for which to reduce the selection, or Default table, if omitted
number	Number	→	Number of records to keep selected

Description

REDUCE SELECTION creates a new selection of records for table. The command returns the first Number of records from the current selection table. REDUCE SELECTION is applied to the current selection of table in the current process. It changes the current selection of table for the current process; the first record of the new selection is the current record.

Example

The following example first finds the correct statistics for a worldwide contest among the dealers in over 20 countries. For each country, the 3 best dealers who have sold product worth more than $50,000 and who are among the 100 best dealers in the world are awarded a prize. With a few lines of code, this complex request can be executed by using indexed searches:

```
CREATE EMPTY SET([Dealers];"Winners")  ` Create an empty set
SCAN INDEX([Dealers]Sales amount;100;<)  ` Scan from the end of the index
CREATE SET([Dealers];"100 best Dealers")  ` Put the selected records in a set
For ($Country;1;Records in table([Countries]))  ` For each Country
        ` Search for the dealers in this country who sold for more than $50000
    QUERY([Dealers];[Dealers]Country=[Countries]Name;*)
    QUERY(&;[Dealers];[Dealers]Sales amount>=50000)
    CREATE SET([Dealers];"WinnerDealers")  ` Put them in a set
        ` They should be in the group of 100 best dealers
    INTERSECTION("WinnerDealers";"100 best Dealers";"WinnerDealers")
    USE SET("WinnerDealers") ` Potential winners for the country
        ` Sort them by the results in descending order
    ORDER BY([Dealers];[Dealers]Sales amount;<)
⇒   REDUCE SELECTION([Dealers];3)  ` Take the 3 best Dealers
    CREATE SET([Dealers];"WinnerDealers")  ` The winners for the country
    ` Put them in the worldwide winners list
    UNION("WinnerDealers";"TheWinners";"TheWinners")
End for
```

```
CLEAR SET("100 best Dealers")  ` Don't need this set anymore
CLEAR SET("WinnerDealers")  ` Don't need this set anymore
USE SET("The Winners")  ` Here you have the Winners
CLEAR SET("The Winners")  ` Don't need this set anymore
OUTPUT FORM([Dealers];"Prize letter")  ` Select the letter
PRINT SELECTION([Dealers]) ` Print the letters
```

See Also

ORDER BY, QUERY, SCAN INDEX, Sets.

SCAN INDEX

Selection

version 3

SCAN INDEX (field; number; > or <)

Parameter	Type		Description
field	Field	→	Indexed field on which to scan index
number	Number	→	Number of records to return
> or <		→	> from beginning of index < from end of index

Description

SCAN INDEX returns a selection of number records for table. If you pass <, SCAN INDEX returns the number of records from the end of the index (high values). If you pass >, SCAN INDEX returns the number of records for table from the beginning of the index (low values). This command is very efficient because it uses the index to perform the operation.

SCAN INDEX works only on indexed fields. This command changes the current selection of the table for the current process, but there is no current record.

If you specify more records than exist in the table, SCAN INDEX will return all records.

Example

The following example mails letters to 50 of the worst customers and then to 50 of the best customers:

```
⇒   SCAN INDEX([Customers]TotalDue;50;<)  ` Get the 50 worst customers
    ORDER BY([Customers]Zipcode;>)  ` Sort by Zip codes
    OUTPUT FORM([Customers];"ThreateningMail")
    PRINT SELECTION([Customers])  ` Print the letters
⇒   SCAN INDEX([Customers]TotalDue;50;>)  ` Get the 50 best customers
    ORDER BY([Customers]Zipcode;>)  ` Sort by Zip codes
    OUTPUT FORM([Customers];"Thanks Letter")
    PRINT SELECTION([Customers])  ` Print the letters
```

See Also

ORDER BY, QUERY, REDUCE SELECTION.

ONE RECORD SELECT

ONE RECORD SELECT {(table)}

Parameter	Type		Description
table	Table	→	Table for which to reduce the selection to the current record, or Default table, if omitted

Description

ONE RECORD SELECT reduces the current selection of table to the current record. If no current record exists, ONE RECORD SELECT has no effect.

Historical Note: This command was useful to "get back" into the selection a record that had been pushed and popped from the record stack while the selection for the table was changed. In version 6, SET QUERY DESTINATION allows you to make a query without changing the selection or the current record of a table; therefore, you no longer need to push and pop a current record in order to query its table. Consequently, ONE RECORD SELECT is less useful, unless you actually want to reduce the selection of a table to the current record.

42 Sets

version 6.0 (Modified)

Sets offer you a powerful, swift means for manipulating record selections. Besides the ability to create sets, relate them to the current selection, and store, load, and clear sets, 4th Dimension offers three standard set operations:
• Intersection
• Union
• Difference

Sets and the Current Selection

A set is a compact representation of a selection of records. The idea of sets is closely bound to the idea of the current selection. Sets are generally used for the following purposes:
• To save and later restore a selection when the order does not matter
• To access the selection a user made on screen (the UserSet)
• To perform a logical operation between selections

The current selection is a list of references that points to each record that is currently selected. The list exists in memory. Only currenly selected records are in the list. A selection doesn't actually contain the records, but only a list of references to the records. Each reference to a record takes 4 bytes in memory. When you work on a table, you always work with the records in the current selection. When a selection is sorted, only the list of references is rearranged. There is only one current selection for each table inside a process.

Like a current selection, a set represents a selection of records. A set does this by using a very compact representation for each record. Each record is represented by one bit (one-eighth of a byte). Operations using sets are very fast, because computers can perform operations on bits very quickly. A set contains one bit for every record in the table, whether the record is included in the set or not. In fact, each bit is equal to 1 or 0, depending on whether the record is in the set or not.

Sets are very economical in terms of RAM space. The size of a set, in bytes, is always equal to the total number of records in the table divided by 8. For example, if you create a set for a table containing 10,000 records, the set takes up 1,250 bytes, which is about 1.2K in RAM.

There can be many sets for each table. In fact, sets can be saved to disk separately from the database. To change a record belonging to a set, first you must use the set as the current selection, then modify the record or records. The name of an interprocess set must be unique in the database.

A set is never in a sorted order—the records are simply indicated as belonging to the set or not. On the other hand, a named selection is in sorted order, but it requires more memory in most cases. For more information about named selections, see the section Named Selections.

A set "remembers" which record was the current record at the time the set was created. The following table compares the concepts of the current selection and of sets:

Comparison	Current Selection	Sets
Number per table	1	0 to many
Sortable	Yes	No
Can be saved on disk	No	Yes
RAM per record(in bytes)	Number of selected records * 4	Total number of records/8
Combinable	No	Yes
Contains current record	Yes	Yes, as of the time the set was created

When you create a set, it belongs to the table from which you created it. The set operations can be performed only between sets belonging to the same table.

Sets are independent from the data. This means that after changes are made to a file, a set may no longer be accurate. There are many operations that can cause a set to be inaccurate. For example, if you create a set of all the people from New York City, and then change the data in one of those records to "Boston" the set would not change, because the set is just a representation of a selection of records. Deleting records and replacing them with new ones also changes a set. Sets can be guaranteed to be accurate only as long as the data in the original selection has not been changed.

Process and Interprocess Sets

You can have the following three types of sets:

• **Process sets**: A process set can only be accessed by the process in which it has been created. UserSet and LockedSet are process sets. Process sets are cleared as soon as the process method ends. Process sets do not need any special prefix in the name.
• **Interprocess sets**: A set is an interprocess set if the name of the set is preceded symbols (<>) — a "less than" sign followed by a "greater than" sign. **Note**: This syntax can be used on both Windows and Macintosh. Also, on Macintosh only, you can use the diamond (Option-Shift-V on a US keyboard).
• **Local Sets/Client Sets**: Version 6 introduces local/client sets. The name of a local/client set is preceded by the dollar sign ($).

Sets and Transactions

A set can be created inside a transaction. It is possible to create a set of the records created inside a transaction and a set of records created or modified outside of a transaction. When the transaction ends, the set created during the transaction should be cleared, because it may not be an accurate representation of the records, especially if the transaction was canceled.

Set Example

The following example deletes duplicate records from a table which contains information about people. A For...End for loop moves through all the records, comparing the current record to the previous record. If the name, address, and zip code are the same, then the record is added to a set. At the end of the loop, the set is made the current selection and the (old) current selection is deleted:

```
CREATE EMPTY SET([People];"Duplicates")
   ` Create an empty set for duplicate records
ALL RECORDS([People])
   ` Select all records
   ` Sort the records by ZIP, address, and name so
   ` that the duplicates will be next to each other
ORDER BY ([People];[People]ZIP;>;[People]Address;>;[People]Name;>)
   ` Initialize variables that hold the fields from the previous record
$Name:=[People]Name
$Address:=[People]Address
$ZIP:=[People]ZIP
   ` Go to second record to compare to first
NEXT RECORD ([People])
For ($i; 2; Records in table ([People]))
      ` Loop through records starting at 2
      ` If the name, address, and ZIP are the same as the
      ` previous record then it is a duplicate record.
   If (([People]Name=$Name) & ([People]Address=$Address) & ([People]ZIP=$ZIP))
         ` Add current record (the duplicate) to set
      ADD TO SET ([People]; "Duplicates")
   Else
         ` Save this record's name, address, and ZIP
         ` for comparison with the next record
      $Name:=[People]Name
      $Address:=[People]Address
      $ZIP:=[People]ZIP
   End if
      ` Move to the next record
   NEXT RECORD ([People])
End for
```

```
    ` Use duplicate records that were found
USE SET ("Duplicates")
    ` Delete the duplicate records
DELETE SELECTION ([People])
    ` Remove the set from memory
CLEAR SET ("Duplicates")
```

As an alternative to immediately deleting records at the end of the method, you could display them on screen or print them, so that a more detailed comparison can be made.

The UserSet System Set

4th Dimension maintains a system set named UserSet. UserSet automatically stores the most recent selection of records highlighted on screen by the user. Thus, you can display a group of records with MODIFY SELECTION or DISPLAY SELECTION, ask the user to select from among them, and turn the results of that manual selection into a selection or into a set that you name.

There is only one UserSet for a process. Each table does not have its own UserSet. UserSet becomes “owned” by a table when a selection of records is displayed for the table.

The following method illustrates how you can display records, allow the user to select some, and then use UserSet to display the selected records:

```
    ` Display all records and allow user to select any number of them.
    ` Then display this selection by using UserSet to change the current selection.
OUTPUT FORM ([People]; "Display") ` Set the output layout
ALL RECORDS ([People]) ` Select all people
ALERT ("Press Ctrl or Command and Click to select the people required.")
DISPLAY SELECTION ([People]) ` Display the people
USE SET ("UserSet") ` Use the people that were selected
ALERT ("You chose the following people.")
DISPLAY SELECTION ([People]) ` Display the selected people
```

Note: You must execute either MODIFY SELECTION or DISPLAY SELECTION to retrieve the UserSet.

The LockedSet System Set

The command APPLY TO SELECTION, ARRAY TO SELECTION and DELETE SELECTION create a set named LockedSet when used in multi-processing environment. LockedSet indicates which records were locked during the execution of the command.

See Also

ADD TO SET, CLEAR SET, COPY SET, CREATE EMPTY SET, CREATE SET, DIFFERENCE, INTERSECTION, Is in set, LOAD SET, Records in set, REMOVE FROM SET, SAVE SET, UNION, USE SET.

CREATE EMPTY SET

CREATE EMPTY SET ({table; }set)

Parameter	Type		Description
table	Table	→	Table for which to create an empty set, or Default table, if omitted
set	String	→	Name of the new empty set

Description

CREATE EMPTY SET creates a new empty set, set, for table. You can add records to this set with the ADD TO SET command. If a set with the same name already exists, the existing set is cleared by the new set.

Note: You do not need to use CREATE EMPTY SET before using CREATE SET.

Example

This example creates a new set and then “merges” the UserSet with it (with the UNION command), in order to save the UserSet:

```
          ` Create a new set
⇒    CREATE EMPTY SET ([People]; "KeepUserSet")  ` Merge the two sets together
     UNION ("UserSet"; "KeepUserSet"; "KeepUserSet")
```

See Also

CLEAR SET, CREATE SET.

CREATE SET

CREATE SET ({table; }set)

Parameter	Type		Description
table	Table	→	Table for which to create a set from the selection, or Default table, if omitted
set	String	→	Name of the new set

Description

CREATE SET creates a new set, set, for table, and places the current selection in set. The current record pointer for the table is saved with set. If set is used with USE SET, the current selection and current record are restored. As with all sets, there is no sorted order; when set is used, the default order is used. If a set with the same name already exists, the existing set is cleared by the new set.

Example

The following example creates a set after doing a search, in order to save the set to disk:

```
     QUERY ([People])  ` Let the user do a search
⇒    CREATE SET ([People]; "SearchSet")  ` Create a new set
     SAVE SET ("SearchSet"; "MySearch")  ` Save the set on disk
```

See Also

CLEAR SET, CREATE EMPTY SET.

USE SET

version 3

USE SET (set)

Parameter	Type		Description
set	String	→	Name of the set to use

Description

USE SET makes the records in set the current selection for the table to which the set belongs.

When you create a set, the current record is “remembered” by the set. USE SET retrieves the position of this record and makes the it the new current record. If you delete this record before you execute USE SET, 4th Dimension selects the first record in the set as the current record. The set commands INTERSECTION, UNION, DIFFERENCE, and ADD TO SET reset the current record. Also, if you create a set that does not contain the position of the current record, USE SET selects the first record in the set as the current record.

WARNING: Remember that a set is a representation of a selection of records at the moment that the set is created. If the records represented by the set do change, the set may no longer be accurate. Therefore, a set saved to disk should represent a group of records that does not change frequently. A number of things can invalidate a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined the set.

Example

The following example uses LOAD SET to load a set of the Acme locations in New York. It then uses USE SET to make the loaded set the current selection:

```
      LOAD SET ([Companies]; "NY Acme"; "NYAcmeSt")   ` Load the set into memory
⇒     USE SET ("NY Acme")   ` Change current selection to NY Acme
      CLEAR SET ("NY Acme")   ` Clear the set from memory
```

See Also

CLEAR SET, LOAD SET.

ADD TO SET

ADD TO SET ({table; }set)

Parameter	Type		Description
table	Table	→	Current record's table, or Default table, if omitted
set	String	→	Name of the set to which to add the current record

Description

ADD TO SET adds the current record of table to set. The set must already exist; if it does not, an error occurs. If a current record does not exist for Table, ADD TO SET has no effect.

See Also

REMOVE FROM SET.

REMOVE FROM SET

version 6.0

REMOVE FROM SET ({table; }set)

Parameter	Type		Description
table	Table	→	Current record's table, or Default table, if omitted
set	String	→	Name of the set from which to remove the current record

Description

REMOVE FROM SET removes the current record of table from set. The set must already exist; if it does not, an error occurs. If a current record does not exist for Table, REMOVE FROM SET has no effect.

See Also

ADD TO SET.

CLEAR SET

version 3

CLEAR SET (set)

Parameter	Type		Description
set	String	→	Name of the set to clear from memory

Description

CLEAR SET clears set from memory and frees the memory used by set. CLEAR SET does not affect tables, selections, or records. To save a set before clearing it, use the SAVE SET command. Since sets use memory, it is good practice to clear them when they are no longer needed.

Example

See the example for USE SET.

See Also

CREATE EMPTY SET, CREATE SET, LOAD SET.

Is in set

Is in set (set) → Boolean

Parameter	Type		Description
set	String	→	Name of the set to test
Function result	Boolean	←	Current record of set's table is in set (True) or Current record of set's table is not in set (False)

Description

Is in set tests whether or not the current record for the table is in set. The Is in set function returns TRUE if the current record of the table is in set, and returns FALSE if the current record of the table is not in set.

Example

The following example is a button object method. It tests to see whether or not the currently displayed record is in the set of best customers:

```
⇒	If (Is in set ("Best"))  ` Check if it is a good customer
		ALERT ("They are one of our best customers.")
	Else
		ALERT ("They are not one of our best customers.")
	End if
```

See Also

ADD TO SET, REMOVE FROM SET.

Records in set

version 3

Records in set (set) → Number

Parameter	Type		Description
set	String	→	Name of the set to test
Function result	Number	←	Number of records in test

Description

Records in set returns the number of records in set. If set does not exist, or if there are no records in set, Records in set returns 0.

Example

The following example displays an alert telling what percentage of the customers are rated as the best:

```
          ` First calculate the percentage
⇒      $Percent := (Records in set ("Best") / Records in table ([Customers])) * 100
          ` Display an alert with the percentage
       ALERT (String ($Percent; "##0%") + " of our customers are the best.")
```

See Also

Records in selection, Records in table.

SAVE SET

version 3

SAVE SET (set; document)

Parameter	Type		Description
set	String	→	Name of the set to save
document	String	→	Name of the disk file to which to save the set

Description

SAVE SET saves Set to document, a document on disk.

The document need not have the same name as the set. If you supply an empty string for document, a Create File dialog box appears so that the user can enter the name of the document. You can load a saved set with the LOAD SET command.

If the user clicks Cancel in the Save File dialog box, or if there is an error during the save operation, the OK system variable is set to 0. Otherwise, it is set to 1.

SAVE SET is often used to save to disk the results of a time-consuming search.

WARNING: Remember that a set is a representation of a selection of records at the moment that the set is created. If the records represented by the set change, the set may no longer be accurate. Therefore, a set saved to disk should represent a group of records that does not change frequently. A number of things can invalidate a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined the set. Also remember that sets do not save field values.

Example

The following example displays the Save File dialog box, wihch the user can enter the name of the document that contains the set:

```
⇒	SAVE SET ("SomeSet"; "")
```

System Variables or Sets

If the user clicks Cancel in the Save File dialog box, or if there is an error during the load operation, the OK system variable is set to 0. Otherwise, it is set to 1.

See Also

LOAD SET.

LOAD SET

LOAD SET ({table; }set; document)

Parameter	Type		Description
table	Table	→	Table to which the set belongs, or Default table, if omitted
set	String	→	Name of the set to be created in memory
document	String	→	Document holding the set

Description

LOAD SET loads a set from document that was saved with the SAVE SET command.

The set that is stored in document must be from table. The set created in memory is overwritten if it already exists.

The document parameter is the name of the disk document containing the set. The document need not have the same name as the set. If you supply an empty string for document, an Open File dialog box appears so that the user can choose the set to load.

Remember that a set is a representation of a selection of records at the moment that the set is created. If the records represented by the set change, the set may no longer be accurate. Therefore, a set loaded from disk should represent a group of records that does not change frequently. A number of things can make a set invalid: modifying a record of the set, deleting a record of the set, or changing the criteria that determined a set.

Example

The following example uses LOAD SET to load a set of the Acme locations in New York:

```
⇒  LOAD SET ([Companies]; "NY Acme"; "NYAcmeSt")   ` Load the set into memory
   USE SET ("NY Acme")   ` Change current selection to NY Acme
   CLEAR SET ("NY Acme")   ` Clear the set from memory
```

System Variables or Sets

If the user clicks Cancel in the Open File dialog box, or there is an error during the load operation, the OK system variable is set to 0. Otherwise, it is set to 1.

See Also

SAVE SET.

DIFFERENCE

DIFFERENCE (set; subtractSet; resultSet)

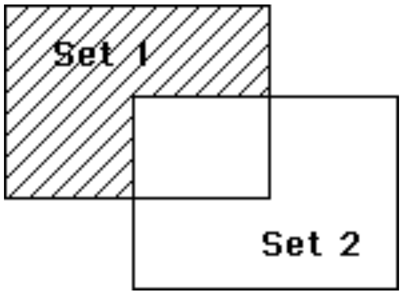
Parameter	Type		Description
set	String	→	Set
subtractSet	String	→	Set to subtract
resultSet	String	→	Resulting set

Description

DIFFERENCE compares set1 and set2 and excludes all records that are in set2 from the resultSet. In other words, a record is included in the resultSet only if it is in set1, but not in set2. The following table shows all possible results of a set Difference operation.

Set1	Set2	Result Set
Yes	No	Yes
Yes	Yes	No
No	Yes	No
No	No	No

The result of a Difference operation is depicted here. The shaded area is the result set.



The resultSet is created by DIFFERENCE. The resultSet replaces any existing set having the same name, including set1 and set2. Both set1 and set2 must be from the same table. The resultSet belongs to the same table as set1 and set2.

4D Server: In Client/Server, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. DIFFERENCE requires the three sets to be on the same machine. Consequently, all or none of the sets must be local. See the discussion 4D Server and Sets in the *4D Server Reference* manual for more information.

Example

The following example excludes the records that a user selects from a displayed selection. The records are displayed on screen with the following line:

```
	DISPLAY SELECTION ([Customers]) ` Display the customers in a list
```

At the bottom of the list of records is a button with an object method. The object method excludes the records that the user has selected (the set named “UserSet”), and displays the reduced selection:

```
	CREATE SET ([Customers]; "Current")  ` Create a set of current selection
⇒	DIFFERENCE ("Current";"UserSet";"Current")  ` Exclude selected records
	USE SET ("Current")  ` Use the new set
	CLEAR SET ("Current")  ` Clear the set
```

See Also

INTERSECTION, UNION.

INTERSECTION

INTERSECTION (set1; set2; resultSet)

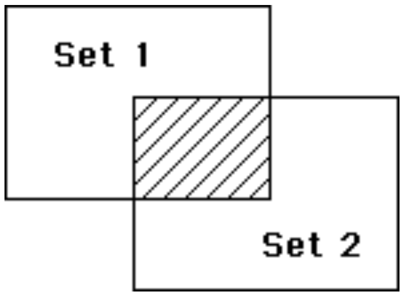
Parameter	Type		Description
set1	String	→	First set
set2	String	→	Second set
resultSet	String	→	Resulting set

Description

INTERSECTION compares set1 and set2 and selects only the records that are in both. The following table lists all possible results of a set Intersection operation.

Set1	Set2	Result Set
Yes	No	No
Yes	Yes	Yes
No	Yes	No
No	No	No

The graphical result of an Intersection operation is displayed here. The shaded area is the result set.



The resultSet is created by INTERSECTION. The resultSet replaces any existing set having the same name, including set1 and set2. Both set1 and set2 must be from the same table. The resultSet belongs to the same table as set1 and set2.

4D Server: In Client/Server, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. INTERSECTION requires the three sets to be on the same machine. Consequently, all or none of the sets must be local. See the discussion 4D Server and Sets in the *4D Server Reference* manual for more information.

Example

The following example finds the customers who are served by two sales representatives, Joe and Abby. Each sales representative has a set that represents his or her customers. The customers that are in both sets are represented by both Joe and Abby:

```
⇒	INTERSECTION ("Joe"; "Abby"; "Both")  ` Put customers in both sets in Both
	USE SET ("Both")  ` Use the set
	CLEAR SET ("Both")  ` Clear this set but save the others
	DISPLAY SELECTION ([Customers])  ` Display customers served by both
```

See Also

DIFFERENCE, UNION.

UNION

version 3

UNION (set1; set2; resultSet)

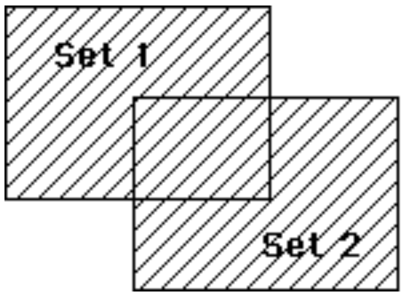
Parameter	Type		Description
set1	String	→	First set
set2	String	→	Second set
resultSet	String	→	Resulting set

Description

UNION creates a set that contains all records from set1 and set2. The following table shows all possible results of a set Union operation.

Set1	Set2	Result Set
Yes	No	Yes
Yes	Yes	Yes
No	Yes	Yes
No	No	No

The result of a Union operation is depicted here. The shaded area is the result set.



The resultSet is created by UNION. The resultSet replaces any existing set having the same name, including set1 and set2. Both set1 and set2 must be from the same table. The resultSet belongs to the same table as set1 and set2. The current record for the resultSet is the current record from Set1.

4D Server: In Client/Server, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. UNION requires the three sets to be on the same machine. Consequently, all or none of the sets must be local. See the discussion 4D Server and Sets in the *4D Server Reference* manual for more information.

Example

The following example adds records to a set of best customers. The records are displayed on screen with the first line. After the records are displayed, a set of the best customers is loaded from disk, and any records that the user selected (the set named “UserSet”) are added to the set. Finally, the new set is saved on disk:

```
        ALL RECORDS ([Customers])  ` Select all the customers
        DISPLAY SELECTION ([Customers])  ` Display all the customers in a list
        LOAD SET ("Best"; "SaveBest")  ` Load the set of best customers
⇒       UNION ("Best"; "UserSet"; "Best")  ` Add any selected to the set
        SAVE SET ("Best"; "SaveBest")  ` Save the set of best customers
```

See Also

DIFFERENCE, INTERSECTION.

COPY SET

version 6.0

COPY SET (srcSet; dstSet)

Parameter	Type		Description
srcSet	String	→	Source set name
dstSet	String	→	Destination set name

Description

The command COPY SET copies the contents of the set srcSet into the set dstSet.

Both sets can be process, interprocess or local sets.

4D Server: In Client/Server, interprocess and process sets are maintained on the server machine, while local sets are maintained on the client machines. COPY SET allows you to copy sets between the two machines. See the discussion 4D Server and Sets in the *4D Server Reference* manual for more information.

Examples

1. The following example, in Client/Server, copies the local set "$SetA", maintained on the client machine, to the process set "SetB", maintained on the server machine:

```
⇒	COPY SET("$SetA";"SetB")
```

(1) The following example, in Client/Server, copies the process set "SetA", maintained on the server machine, to the local process set "$SetB", maintained on the client machine:

```
⇒	COPY SET("SetA";"$SetB")
```

See Also

Sets.

43 String

String

String (expression{; format}) → String

Parameter	Type		Description
expression		→	Expression for which to return the string form (can be Real, Integer, Long Integer, Date, or Time)
format	String | Number	→	Display format
Function result	String	←	String form of the expression

Description

The command String returns the string form of the numeric, Date, or Time expression you pass in expression.

If you do not pass the optional format parameter, the string is returned with the appropriate default format. If you pass format, you can force the result string to be of a specific format.

Numeric Expressions
If expression is a numeric expression (Real, Integer, Long Integer), you can pass an optional string format. Following are some examples:

Example	Result
String(2^15) ` Use default format	32768 (Default format used here)
String(2^15;"###,##0 Inhabitants")	32,768 Inhabitants
String(1/3;"##0.00000")	0.33333
String(1/3) ` Use default format	0.3333333333333333 (Default format used here)
String(Arctan(1)*4)	3.1415926535897931 (Default format used here)
String(Arctan(1)*4;"##0.00")	3.14
String(-1;"&x")	0xFFFFFFFF
String(-1;"&$")	$FFFFFFFF
String(0 ?+ 7;"&x")	0x80
String(0 ?+ 7;"&$")	$80
String(0 ?+ 14;"&x")	0x4000
String(0 ?+ 14;"&$")	$4000
String(Num(1=1);"True;;False")	True
String(Num(1=2);"True;;False")	False

The format is specified in the same way as it would be for a number field on a form. See the *4th Dimension Design Reference* for more information about formatting numbers. You can also pass the name of a custom style in format. The name of the custom style must be preceded by the "|" character.

Date Expressions
If expression is a Date expression, the string is returned using the default country format (i.e., MM/DD/YY for the U.S. English langauge version).
You can pass an optional numeric format from the following table:

Format	**Name**	**Example**
1	Short	12/29/96
2	Abbreviated	Sun, Dec 29, 1996
3	Long	Sunday, December 29, 1996
4	MM/DD/YYYY	12/29/96 or 12/29/1896 or 12/29/2096
5	Month Date, Year	December 29, 1996
6	Abbr: Month Date, Year	Dec 29, 1996
7	MM/DD/YYYY Forced	12/29/1996

4D provides the following predefined constants:

Constant	**Type**	**Value**
Short	Long Integer	1
Abbreviated	Long Integer	2
Long	Long Integer	3
MM DD YYYY	Long Integer	4
Month Date Year	Long Integer	5
Abbr Month Date	Long Integer	6
MM DD YYYY Forced	Long Integer	7

These examples assume that the current date is 12/29/96):

```
   ` $vsResult gets "12/29/96"
$vsResult:=String(Current date)
   ` $vsResult gets "December 29, 1996"
$vsResult:=String(Current date;Month Date Year)
```

Time Expressions
If expression is a Time expression, the string is returned using the default HH:MM:SS format. You can pass an optional numeric format from the following table:

Format	**Name**	**Example**
1	HH:MM:SS	01:02:03
2	HH:MM	01:02
3	hour min sec	1 hour 2 minutes 3 seconds
4	hour min	1 hour 2 minutes
5	H:MM AM/PM	1:02 AM

4D provides the following predefined constants:

Constant	Type	Value
HH MM SS	Long Integer	1
HH MM	Long Integer	2
Hour Min Sec	Long Integer	3
Hour Min	Long Integer	4
HH MM AM PM	Long Integer	5

These examples assume that the current time is 5:30 PM and 45 seconds):

```
	$vsResult:=String(Current time)   ` $vsResult gets "17:30:45"
	$vsResult:=String(Current time;Hour Min Sec)   ` $vsResult gets "17 hours 30 minutes 
45 seconds"
```

See Also

Date, Num, Time string.

Num

String

version 3

Num (expression) → Number

Parameter	Type		Description
expression	String | Boolean	→	String for which to return the numeric form, or Boolean to return 0 or 1
Function result	Number	←	Numeric form of the string or Boolean

Description

The command Num returns the numeric form of the String or Boolean expression you pass in expression.

String Expressions

If string consists only of one or more alphabetic characters, Num returns a zero. If string includes alphabetic and numeric characters, Num ignores the alphabetic characters. Thus, Num transforms the string "a1b2c3" into the number 123.

Note: Only the first 32 characters of string are evaluated.

There are three reserved characters that Num treats specially: the decimal separator in the US English version (i.e., the period “.”) , the hyphen “-”, and “e” or “E”. These characters are interpreted as numeric format characters.

• The decimal separator is interpreted as a decimal place and must appear embedded in a numeric string.
• The hyphen causes the number or exponent to be negative. The hyphen must appear before any negative numeric characters or after the “e” for an exponent. If a hyphen is embedded in a numeric string, the string is considered invalid and the Num function returns a zero (0). For example, Num(123-456) returns 0, but Num(-9) returns -9.
• The e or E causes any numeric characters to its right to be interpreted as the power of an exponent. The “e” must be embedded in a numeric string. Thus, Num("123e–2") returns 1.23.

Boolean Expressions
If you pass a Boolean expression, Num returns 1 if the expression is True; otherwise, it returns 0 (zero).

Examples

1. The following example illustrates how Num works when passed a string argument. Each line assigns a number to the vResult variable. The comments describe the results:

```
⇒	vResult := Num ("ABCD")  ` vResult gets 0
⇒	vResult := Num ("A1B2C3")  ` vResult gets 123
⇒	vResult := Num ("123")  ` vResult gets 123
⇒	vResult := Num ("123.4")  ` vResult gets 123.4
⇒	vResult := Num ("–123")  ` vResult gets –123
⇒	vResult := Num ("–123e2")  ` vResult gets –12300
```

2. Here, [Client]Debt is compared with $1000. The Num command applied to these comparisons returns 1 or 0. Multiplying 1 or 0 with a string repeats the string once or returns the empty string. As a result, [Client]Risk gets either "Good" or "Bad":

```
		` If client owes less than 1000, a good risk.
		` If client owes more than 1000, a bad risk.
⇒	[Client]Risk:=("Good"*Num ([Client]Debt<1000))+("Bad"*Num([Client]Debt>=1000))
```

See Also

Logical Operators, String, String Operators.

Position

String

version 3

Position (find; string) → Number

Parameter	Type		Description
find	String	→	String to find
string	String	→	String in which to search
Function result	Number	←	Position of first occurrence

Description

Position returns the position of the first occurrence of find in string.

If string does not contain find, it returns a zero (0).

If Position locates an occurrence of find, it returns the position of the first character of the occurrence in string.

If you ask for the position of an empty string within an empty string, Position returns zero (0).

Warning: You cannot use the @ wildcard character with Position. For example, if you pass "abc@" in find, the command will actually look for "abc@" and not for "abc" plus any character.

Examples

1. This example illustrates the use of Position. The results, described in the comments, are assigned to the variable vlResult.

```
⇒	vlResult := Position ("ll"; "Willow")  ` vlResult gets 3
⇒	vlResult := Position (vtText1; vtText2)  ` Returns first occurrence of vtText1 in vtText2
```

2. See example for the command Substring.

See Also

Comparison Operators, Substring.

Substring

Substring (source; firstChar{; numChars}) → String

Parameter	Type		Description
source	String	→	String from which to get substring
firstChar	Number	→	Position of first character
numChars	Number	→	Number of characters to get
Function result	String	←	Substring of source

Description

The command Substring returns the portion of source defined by firstChar and numChars.

The firstChar parameter points to the first character in the string to return, and numChars specifies how many characters to return.

If firstChar plus numChars is greater than the number of characters in the string, or if numChars is not specified, Substring returns the last character(s) in the string, starting with the character specified by firstChar. If firstChar is greater than the number of characters in the string, Substring returns an empty string ("").

Examples

1. This example illustrates the use of Substring. The results, described in the comments, are assigned to the variable vsResult.

```
⇒      vsResult := Substring ("08/04/62"; 4; 2)  ` vsResult gets "04"
⇒      vsResult := Substring ("Emergency"; 1; 6)  ` vsResult gets "Emerge"
⇒      vsResult := Substring (var; 2)  ` vsResult gets all characters except ` the first
```

2. The following project method appends the paragraphs found in the text (passed as first parameter) to a string or text array (the pointer of which is passed as second parameter):

```
          ` EXTRACT PARAGRAPHS
          ` EXTRACT PARAGRAPHS ( text ; Pointer )
          ` EXTRACT PARAGRAPHS ( Text to parse ; -> Array of ¶s )

      C_TEXT ($1)
      C_POINTER ($2)

      $vlElem:=Size of array($2->)
      Repeat
          $vlElem:=$vlElem+1
          INSERT ELEMENT($2->;$vlElem)
          $vlPos:=Position(Char(Carriage return);$1)
          If ($vlPos>0)
⇒             $2->{$vlElem}:=Substring($1;1;$vlPos-1)
⇒             $1:=Substring($1;$vlPos+1)
          Else
              $2->{$vlElem}:=$1
          End if
      Until ($1="")
```

See Also

Position.

Length

String

version 3

Length (string) → Number

Parameter	Type		Description
string	String	→	String for which to return length
Function result	Number	←	Length of string

Description

Length is used to find the length of string. Length returns the number of characters that are in string.

Note: The test If (vtAnyText="") is equivalent to the test If(Length(vtAnyText)=0).

Examples

This example illustrates the use of Length. The results, described in the comments, are assigned to the variable vlResult.

```
⇒    vlResult := Length ("Topaz")  ` vlResult gets 5
⇒    vlResult := Length ("Citizen")  ` vlResult gets 7
```

Ascii

String

version 3

Ascii (character) → Number

Parameter	Type		Description
character	String	→	Character to return as an ASCII code
Function result	Number	←	ASCII code for the character

Description

The command Ascii returns the ASCII code of character.

If there is more than one character in the string, Ascii returns the code of the first character.

The Char function is the counterpart of Ascii. It returns the character that an ASCII code represents.

Important: Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are MacOS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Examples

1. Uppercase and lowercase characters are considered equal within a comparison. You can use Ascii to differentiate between uppercase and lowercase characters. Thus, this line returns True:

```
("A" = "a")
```

On the other hand, this line returns False:

```
⇒   (Ascii("A")=Ascii("a"))
```

2. This example returns the ASCII value of the first character of the string "ABC":

```
⇒   vlAscii:=Ascii("ABC")  ` vlAscii gets 65, the ASCII code of A
```

3. The following example tests for carriage returns and tabs:

```
For($vlChar;1;Length(vtText))
   Case of
      : (vtText[[$vlChar]]=Char(Carriage return))
         ` Do something
      : (vtText[[$vlChar]]=Char(Tab))
         ` Do something else
      : (...)
         ` ...
   End case
End for
```

When executed multiple times on large texts, this test will run faster when compiled if it is written this way:

```
      For($vlChar;1;Length(vtText))
⇒        $vlAscii:=Ascii(vtText[[$vlChar]])
         Case of
            : ($vlAscii=Carriage return)
               ` Do something
            : ($vlAscii=Tab)
               ` Do something else
            : (...)
               ` ...
         End case
      End for
```

The second piece of code runs faster for two reasons: it does only one character reference by iteration and uses LongInt comparisons instead of string comparisons to test for carriage returns and tabs. Use this technique when working with common ASCII codes such as CR and TAB.

See Also

ASCII Codes, Char, Character Reference Symbols.

Char

String

version 3

Char (asciiCode) → String

Parameter	Type		Description
asciiCode	Number	→	ASCII code from 0 to 255
Function result	String	←	Character represented by the ASCII code

Description

The command Char returns the character whose ASCII code is asciiCode.

Tip: In editing a method, the command Char is commonly used to specify characters that cannot be entered from the keyboard or that would be interpreted as an editing command in the Method editor.

Important: Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are MacOS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Example

The following example uses Char to insert a carriage return within the text of an alert message:

```
⇒      ALERT("Employees: "+String(Records in table([Employees]))+
                                        Char(13)+"Press OK to continue.")
```

See Also

Ascii, ASCII Codes, Character Reference Symbols.

Introduction

The character reference symbols:

```
[[...]]  On Windows
≤...≥ or [[...]] On Macintosh
```

are used to refer to a single character within a string. This syntax allows you to individually address the characters of a text variable, string variable, or field.

Note: On Macintosh, you obtain the first two symbols by typing Option+"<" and Option+">".

If the character reference symbols appear on the left side of the assignment operator (:=), a character is assigned to the referenced position in the string. For example, if vsName is not an empty string, the following line sets the first character of vsName to uppercase:

```
If (vsName#"")
   vsName[[1]]:=Uppercase(vsName[[1]])
End if
```

Otherwise, if the character reference symbols appear within an expression, they return the character (to which they refer) as a 1-character string. For example:

```
   ` The following example tests if the last character of vtText is an At sign "@"
If (vtText # "")
   If (Ascii(Substring(vtText;Length(vtText);1))=At Sign)
      ` ...
   End if
End if

   ` Using the character reference syntax, you would write in a simpler manner:
If (vtText # "")
   If (Ascii(vtText[[Length(vtText)]])=At Sign)
      ` ...
   End if
End if
```

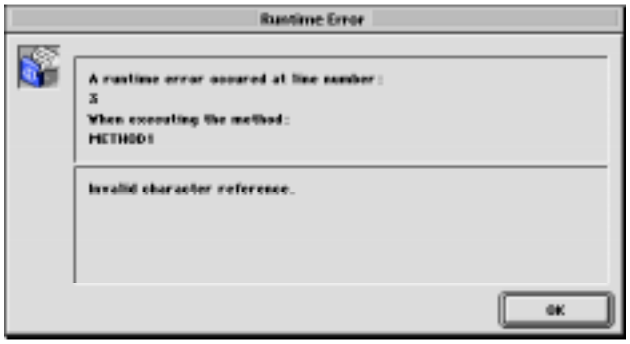
Advanced note about invalid character reference

When you use the character reference symbols, you must address existing characters in the string in the same way you address existing elements of an array. For example if you address the 20th character of a string variable, this variable MUST contain at least 20 characters.

• Failing to do so, in interpreted mode, does not cause a syntax error.
• Failing to do so, in compiled mode (with no options), may lead to memory corruption, if, for instance, you write a character beyond the end of a string or a text.
• Failing to do so, in compiled mode, causes an error with the option Range Checking On. For example, executing the following code:

```
vsAnyText:=""
vsAnyText[[1]]:="A" ` Very bad and nasty thing to do, booh!
```

will trigger the Runtime Error shown here:



Runtime Error

A runtime error occured at line number :
3
When executing the method :
METHOD1

Invalid character reference.

Example

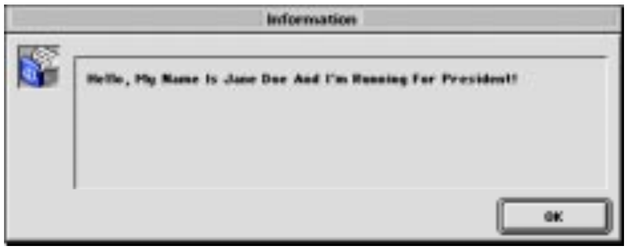
The following project method capitalizes the first character of each word of the text received as parameter and returns the resulting capitalized text:

```
    ` Capitalize text project method
    ` Capitalize text ( Text ) -> Text
    ` Capitalize text ( Source text ) -> Capitalized text
$0:=$1
$vlLen:=Length($0)
If ($vlLen>0)
    $0[[1]]:=Uppercase($0[[1]])
    For ($vlChar;1;$vlLen-1)
        If (Position($0[[$vlChar]];" !&()-{}:;<>?/,.=+*")>0)
            $0[[$vlChar+1]]:=Uppercase($0[[$vlChar+1]])
        End if
    End for
End if
```

For example, the line:

ALERT(*Capitalize text* ("hello, my name is jane doe and i'm running for president!"))

displays the alert shown here:



Hello, My Name Is Jane Doe And I'm Running For President!

See Also
Ascii, ASCII Codes, Char.

Uppercase

String

version 3

Uppercase (chaîne) → String

Parameter	Type		Description
chaîne	Alpha	→	Chaîne à convertir en majuscules
Function result	String	←	String in uppercase

Description

Uppercase takes string and returns the string with all alphabetic characters in uppercase.

Examples

See the example for Lowercase.

See Also

Lowercase.

Lowercase

String

version 3

Lowercase (string) → String

Parameter	Type		Description
string	String	→	String to convert to lowercase
Function result	String	←	String in lowercase

Description

Lowercase takes string and returns the string with all alphabetic characters in lowercase.

Example

The following project method capitalizes the string or text received as parameter. For instance, Caps ("john") would return "John".

```
     ` Caps project method
     ` Caps ( String ) -> String
     ` Caps ( Any text or string ) -> Capitalized text

⇒   $0:=Lowercase($1)
    If (Length($0)>0)
       $0[[1]]:=Uppercase($0[[1]])
    End if
```

See Also

Uppercase.

Change string

Change string (source; newChars; where) → String

Parameter	Type		Description
source	String	→	Original string
newChars	String	→	New characters
where	Number	→	Where to start the changes
Function result	String	←	Resulting string

Description

Change string changes a group of characters in source and returns the resulting string. Change string overlays source, with the characters in newChars, at the character described by where.

If newChars is an empty string (""), Change string returns source unchanged. Change string always returns a string of the same length as source. If where is less than one or greater than the length of source, Change string returns source.

Change string is different from Insert string in that it overwrites characters instead of inserting them.

Example

The following example illustrates the use of Change string. The results are assigned to the variable vtResult.

```
⇒	vtResult := Change string ("Acme"; "CME"; 2)  ` vtResult gets "ACME"
⇒	vtResult := Change string ("November";"Dec"; 1)  ` vtResult gets "December"
```

See Also

Delete string, Insert string, Replace string.

Insert string

Insert string (source; what; where) → String

Parameter	Type		Description
source	String	→	String in which to insert the other string
what	String	→	String to insert
where	Number	→	Where to insert
Function result	String	←	Resulting string

Description

Insert string inserts a string into source and returns the resulting string. Insert string inserts the string what before the character at position where.

If what is an empty string (""), Insert string returns source unchanged.

If where is greater than the length of source, then what is appended to source. If where is less than one (1), then what is inserted before source.

Insert string is different from Change string in that it inserts characters instead of overwriting them.

Example

The following example illustrates the use of Insert string. The results are assigned to the variable vtResult.

```
⇒    vtResult := Insert string ("The tree"; " green"; 4)  ` vtResult gets "The green tree"
⇒    vtResult := Insert string ("Shut"; "o"; 3)  ` vtResult gets "Shout"
⇒    vtResult := Insert string ("Indention"; "ta"; 6)  ` vtResult gets "Indentation"
```

See Also

Change string, Delete string, Replace string.

Delete string

String

version 3

Delete string (source; where; numChars) → String

Parameter	Type		Description
source	String	→	String from which to delete characters
where	Number	→	First character to delete
numChars	Number	→	Number of characters to delete
Function result	String	←	Resulting string

Description

Delete string deletes numChars from source, starting at where, and returns the resulting string.

Delete string returns the same string as source when:
- source is an empty string
- where is greater than the length of Source
- numChars is zero (0)

If where is less than one, the characters are deleted from the beginning of the string.

If where plus numChars is equal to or greater than the length of source, the characters are deleted from where to the end of source.

Example

The following example illustrates the use of Delete string. The results are assigned to the variable vtResult.

```
⇒	vtResult:=Delete string("Lamborghini"; 6; 6)  ` vtResult gets "Lambo"
⇒	vtResult:=Delete string("Indentation"; 6; 2)  ` vtResult gets "Indention"
⇒	vtResult:=Delete string(vtOtherVar;3;32000)  ` vtResult gets the first two characters of vtOtherVar
```

See Also

Change string, Insert string, Replace string.

Replace string

Replace string (source; oldString; newString{; howMany}) → String

Parameter	Type		Description
source	String	→	Original string
oldString	String	→	Characters to replace
newString	String	→	Replacement string (if empty string, occurrences are deleted)
howMany	Number	→	How many times to replace If omitted, all occurrences are replaced
Function result	String	←	Resulting string

Description

Replace string replaces howMany occurrences of oldString in source with newString.

If newString is an empty string (""), Replace string deletes each occurrence of oldString in source.

If howMany is specified, Replace string will replace only the number of occurrences of oldString specified, starting at the first character of source. If howMany is not specified, then all occurrences of oldString are replaced.

If oldString is an empty string, Replace string returns the unchanged source.

Examples

1. The following example illustrates the use of Replace string. The results, described in the comments, are assigned to the variable vtResult.

```
⇒	vtResult:=Replace string("Willow"; " ll"; "d")  ` Result gets "Widow"
⇒	vtResult:=Replace string("Shout"; "o ";"")  ` Result gets "Shut"
⇒	vtResult:=Replace string(vtOtherVar;Char(9);",")  ` Replaces all tabs in vtOtherVar with
commas
```

2. The following example eliminates CRs and TABs from the text in vtResult:

```
⇒	vtResult:=Replace string(Replace string(vtResult;Char(13);"");Char(9);"")
```

See Also

Change string, Delete string, Insert string.

Mac to Win

String

version 6.0

Mac to Win (text) → String

Parameter	Type		Description
text	String	→	Text expressed using MacOS ASCII map
Function result	String	←	Text expressed using Windows ANSI map

Description

The command Mac to Win returns the text, expressed using the Windows ANSI map, equivalent to the text you pass in Text, expressed using the MacOS ASCII map.

This command expects a text parameter expressed using the MacOS ASCII map.

Generally, when running on Windows, you do not need to use this command to convert ASCII codes. When you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, you need to explicitly invoke ASCII conversions. This is the main purpose of the Mac to Win command.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are MacOS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Example

On Windows, when you write characters into a document using SEND PACKET, if you do not use an output ASCII map for filtering characters from MacOS to Windows (see USE ASCII MAP), you need to convert the text from MacOS to Windows yourself. You can do it this way:

```
	` ...
⇒	SEND PACKET ($vhDocRef;Mac to Win(vtSomeText))
	` ...
```

See Also

ASCII Codes, SEND PACKET, USE ASCII MAP, Win to Mac.

Win to Mac

String

version 6.0

Win to Mac (text) → String

Parameter	Type		Description
text	String	→	Text expressed using Windows ANSI map
Function result	String	←	Text expressed using Macintosh ASCII map

Description

The command Win to Mac returns text, expressed using the MacOS ASCII map, equivalent to the text you pass in Text, expressed using the Windows ANSI map.

This command expects a text parameter expressed using the Windows ANSI map.

Generally, when running on Windows, you do not need to use this command to convert ASCII codes. When you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, you need to explicitly invoke ASCII conversions. This is the main purpose of the Win to Mac command.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are MacOS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

Example

On Windows, when you read characters from a document using RECEIVE PACKET, if you do not use an input ASCII map for filtering characters from Windows to MacOS (see USE ASCII MAP), you need to convert the text from Windows to MacOS yourself. You can do it this way:

```
	` ...
	RECEIVE PACKET ($vhDocRef;vtSomeText;16*1024)
⇒	vtSomeText:=Win to Mac(vtSomeText)
	` ...
```

See Also

ASCII Codes, Mac to Win, RECEIVE PACKET, USE ASCII MAP.

Mac to ISO

String

version 6.0

Mac to ISO (text) → String

Parameter	Type		Description
text	String	→	Text expressed using MacOS ASCII map
Function result	String	←	Text expressed using ISO Latin-1 character map

Description

The command Mac to ISO returns text, expressed using the ISO Latin-1 character map, equivalent to the text you pass in text, expressed using the MacOS ASCII map.

You will generally not need to use this command.

This command expects a text parameter expressed using the MacOS ASCII map.

4D converts characters received from and sent to a Web Browser. As a result, the text values you deal with, inside a Web Connection process, are always expressed using the MacOS ASCII map.

Generally, when running on Windows, you do not need to convert ASCII codes. When you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, you need to explicitly invoke ASCII conversions.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are MacOS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

On Windows, it is necessary that, in this case, you do not filter the characters using an output filter ASCII map.

Consequently, no matter what the platform, if you want to write ISO Latin-1 HTML documents on disk, you just need to convert the text using Mac to ISO. This is the main purpose of the Mac to ISO command.

Examples

1. The following line of code converts the (assumed) MacOS encoded text stored in vtSomeText into an ISO-Latin 1 encoded text:

```
⇒      vtSomeText:=Mac to ISO(vtSomeText)
```

2. While developing a 4D Web Server application, you build HTML documents that you later send over Intranet or Internet, using the command SEND HTML FILE. Some of these documents have references or links to other documents.
The following project method calculates an HTML-based pathname from the Windows or Macintosh pathname received as parameter:

```
          ` HTML Pathname project method
          ` HTML Pathname ( Text ) -> Text
          ` HTML Pathname ( Native File Manager Pathname ) -> HTML Pathname

       C_TEXT($0;$1)
       C_LONGINT($vlChar;$vlAscii)
       C_STRING(31;$vsChar)

       $0:=""
       If (On Windows )
          $1:=Replace string($1;"\";"/")
       Else
          $1:=Replace string($1;":";"/")
       End if
⇒      $1:=Mac to ISO($1)
       For ($vlChar;1;Length($1))
          $vlAscii:=Ascii($1[[$vlChar]])
          Case of
             : ($vlAscii>=127)
                $vsChar:="%"+Substring(String($vlAscii;"&$");2)
             : (Position(Char($vlAscii);":<>&%= "+Char(34))>0)
                $vsChar:="%"+Substring(String($vlAscii;"&$");2)
          Else
             $vsChar:=Char($vlAscii)
          End case
          $0:=$0+$vsChar
       End for
```

Note: The project method On Windows is listed in the section System Documents.

Once this project method is present in your database, if you want to include a list of FTP links to documents present in a particular directory, you can write:

```
	` Interprocess variables set, for instance, in the On Startup database method
<>vsFTPURL:="ftp://123.4.56.78/Spiders/"
<>vsFTPDirectory:="APS500:Spiders:" ` Here, a MacOS File Manager pathname
	` ...

	` ...
ARRAY STRING(31;$asDocuments;0)
DOCUMENT LIST(...;$asDocuments)
$vlNbDocuments:=Size of array($asDocuments)
jsHandler:=...
For ($vlDocument;1;$vlNbDocuments)
	vtHTMLCode:=vtHTMLCode+"<P><A HREF="+Char(34)+<>vsFTPURL
			+HTML Pathname
(Substring($1+$asDocuments{$vlDocument};Length(<>vsFTPDirectory)+1))
			+Char(34)+jsHandler+">
"+$asDocuments{$vlDocument}+"</A></P>"+Char(13)
End for
	` ...
```

See Also

ASCII Codes, ISO to Mac, SEND HTML FILE, SEND PACKET, USE ASCII MAP.

ISO to Mac

String

version 6.0

ISO to Mac (text) → String

Parameter	Type		Description
text	String	→	Text expressed using ISO Latin-1 character map
Function result	String	←	Text expressed using MacOS ASCII map

Description

The command ISO to Mac returns text, expressed using the MacOS ASCII map, equivalent to the text you pass in text, expressed using the ISO Latin-1 character map.

You will generally not need to use this command.

This command expects a text parameter expressed using the ISO Latin-1 character map.

4D converts characters received from and sent to a Web Browser. As a result, the text values you deal with, inside a Web Connection process, are always expressed using the MacOS ASCII map.

Generally, when running on Windows, you do not need to convert ASCII codes. When you copy or paste text between 4D and Windows or when you import or export data, 4D automatically performs these conversions. However, when you use disk read/write commands such as SEND PACKET or RECEIVE PACKET, 4D does not perform any ASCII code conversion.

Within 4D, all the text values, fields, or variables that you have not yet converted to another ASCII map are MacOS-encoded on both Macintosh and Windows. For more information, see the section ASCII Codes.

On Windows, it is necessary that, in this case, you do not filter the characters using an output filter ASCII map.

Consequently, no matter what the platform, if you want to use RECEIVE PACKET to read ISO Latin-1 HTML documents from the disk, you just need to convert the text using ISO to Mac. This is the main purpose of the ISO to Mac command.

Example

The following line of code converts the (assumed) ISO Latin-1 encoded text stored in vtSomeText into a MacOS encoded text:

```
	RECEIVE PACKET ($vhDocRef;vtSomeText;16*1024) ` Read some text from an ISO
Latin-1 HTML document
⇒	vtSomeText:=ISO to Mac(vtSomeText)
```

See Also

ASCII Codes, Mac to ISO, RECEIVE PACKET, USE ASCII MAP.

44 Structure Access

Structure Access

The commands in this theme return a description of the database structure. They return the number of tables, the number of fields in each table, the names of the tables and fields, and the type and properties of each field.

Determining the database structure is extremely useful when you are developing and using groups of project methods and forms that can be copied into different databases.

The ability to read the database structure allows you to develop and use portable code.

See Also

Count fields, Count tables, Field, GET FIELD PROPERTIES, Pointers, SET INDEX, Table, Table name.

Count tables

Count tables → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of tables in the database

Description

Count tables returns the number of tables in the database. Tables are numbered in the order in which they are created.

Example

The following example builds an array, named asTables, with the names of tables defined in the database. This array can be used as a drop-down list (or tab control, scrollable area, and so on) to display the list of the tables, within a form:

```
⇒   ARRAY STRING (31;asTables;Count tables)
    For ($vlTable; 1; Size of array(asTables))
        asTables {$vlTable}:=Table name ($vlTable)
    End for
```

See Also

Arrays, Count fields, Table name.

Count fields

Count fields (tableNum | tablePtr) → Number

Parameter	Type		Description
tableNum | tablePtr	Number | Pointer	→	Table number or Pointer to table
Function result	Number	←	Number of fields in table

Description

The command Count fields returns the number of fields in the table whose number or pointer you pass in TableNum or TablePtr.

Fields are numbered in the order in which they are created.

Example

The following project method builds the array asFields, consisting of the field names, for the table whose pointer is received as first parameter:

```
   $vlTable:=Table($1)
⇒  ARRAY STRING(31;asFields;Count fields($vlTable))
   For ($vlField;1;Size of array(asFields))
      asFields{$vlTable}:=Field name($vlTable;$vlField)
   End for
```

See Also

Arrays, Count tables, Field name, GET FIELD PROPERTIES.

Table name

Structure Access

version 3

Table name (tableNum | tablePtr) → String

Parameter	Type		Description
tableNum | tablePtr	Number | Pointer	→	Table number or Table pointer
Function result	String	←	Name of the table

Description

The command Table name returns the name of the table whose number of pointer you pass in tableNum or tablePtr.

Example

The following is an example of a generic method that displays the records of a table. The reference to the table is passed as a pointer to the table. The Table name command is used to include the name of the table in the title bar for the window:

```
	` SHOW CURRENT SELECTION Project method
	` SHOW CURRENT SELECTION ( Pointer )
	` SHOW CURRENT SELECTION (->[Table])

⇒	SET WINDOW TITLE("Records for "+Table name($1))  ` Sets the window title
	DISPLAY SELECTION($1->)  ` Displays the selection
```

See Also

Count tables, Field name, Table.

Field name

Field name ({tableNum; }fieldNum | fieldPtr) → String

Parameter	Type		Description
tableNum	Number	→	Table number
fieldNum | fieldPtr	Number	→	Field number if Table number is passed, or Field pointer
Function result	String	←	Name of the field

Description

The command Field name returns the name of the field whose pointer you pass in fieldPtr or whose table and field number you pass in tableNum and fieldNum.

Examples

1. This example sets the second element of the array FieldArray{1} to the name of the second field in the first table. FieldArray is a two-dimensional array:

```
⇒	FieldArray{1}{2}:=Field name(1;2)
```

2. This example sets the second element of the array FieldArray{1} to the name of the field [MyTable]MyField. FieldArray is a two-dimensional array:

```
⇒	FieldArray{1}{2}:=Field name(->[MyTable]MyField)
```

3. This example displays an alert. This method passes a pointer to a field:

```
⇒	ALERT("The ID number for the field "+Field name($1)+" in the table "
			+Table name(Table($1))+" has to be longer than five characters.")
```

See Also

Count fields, Field, Table name.

Table

Table (tableNum | aPtr) → Pointer | Number

Parameter	Type		Description
tableNum | aPtr	Number | Pointer	→	Table number, or Table pointer, or Field pointer
		←	
Function result	Pointer | Number	←	Table pointer, if a Table number is passed Table number, if a Table pointer is passed Table number, if a Field pointer is passed

Description

The command Table has three forms:
• If you pass a table number in tableNum, Table returns a pointer to the table.
• If you pass a table pointer in aPtr, Table returns the table number of the table.
• If you pass a field pointer in aPtr, Table returns the table number of the field.

Examples

1. This example sets the tablePtr variable to a pointer to the third table of the database:

⇒ TablePtr:=**Table**(3)

2. Passing tablePtr (a pointer to the third table) to Table returns the number 3. The following line sets TableNum to 3:

⇒ TableNum:=**Table**(TablePtr)

3. This example sets the tableNum variable to the table number of [Table3]:

⇒ TableNum:=**Table**(->[Table3])

4. This example sets the tableNum variable to the table number of the table to which the [Table3]Field1 field belongs:

⇒ TableNum:=**Table** (->[Table3]Field1)

See Also

Count tables, Field, Pointers, Table name.

Field

version 3

Field (tableNum | fieldPtr{; fieldNum}) → Number | Pointer

Parameter	Type		Description
tableNum | fieldPtr	Number | Pointer	→	Table number or Field pointer
fieldNum	Number	→	Field number, if Table number is passed
Function result	Number | Pointer	←	Field number, if Field pointer is passed Field pointer, if Table and Field numbers are passed

Description

The command Field has two forms:
• If you pass a table number in tableNum and a field number in fieldNum, Field returns a pointer to the field.
• If you pass a field pointer in fieldPtr, Field returns the field number of the field.

Examples

1. The following example sets the fieldPtr variable to a pointer to the second field in the third table:

```
⇒	FieldPtr:=Field(3; 2)
```

2. Passing fieldPtr (a pointer to the second field of a table) to Field returns the number 2. The following line sets FieldNum to 2:

```
⇒	FieldNum:=Field(FieldPtr)
```

3. The following example sets the FieldNum variable to the field number of [Table3]Field2:

```
⇒	FieldNum:=Field(->[Table3]Field2)
```

See Also

Count fields, Field name, GET FIELD PROPERTIES, Table.

GET FIELD PROPERTIES

GET FIELD PROPERTIES ({tableNum; }fieldNum | fieldPtr; fieldType{; fieldLen{; indexed}})

Parameter	Type		Description
tableNum	Number	→	Table number
fieldNum | fieldPtr	Number | Pointer	→	Field number, if Table number is passed, or Field pointer
fieldType	Number	←	Type of field
fieldLen	Number	←	Length of field, if Alphanumeric
indexed	Boolean	←	TRUE = Indexed, FALSE = Non indexed

Description

The command GET FIELD PROPERTIES returns information about the field specified by fieldPtr or by tableNum and fieldNum.

You either pass:
• the table and field numbers in tableNum and fieldNum, or
• a pointer to the field in fieldPtr.

After the call:

• The fieldType parameter returns the type of the field. The fieldType variable parameter can take a value provided by the following predefined constants:

Constant	Type	Value
Is Alpha Field	Long Integer	0
Is Text	Long Integer	2
Is Real	Long Integer	1
Is Integer	Long Integer	8
Is LongInt	Long Integer	9
Is Date	Long Integer	4
Is Time	Long Integer	11
Is Boolean	Long Integer	6
Is Picture	Long Integer	3
Is Subtable	Long Integer	7
Is BLOB	Long Integer	30

• The fieldLen parameter returns the length of the field, if the field is Alphanumeric (i.e., fieldType=Is Alpha Field). The value of fieldLen is meaningless for the other field types.

• The indexed parameter returns TRUE is the field is indexed, and FALSE if not. The value of indexed is meaningful only for Alphanumeric, Integer, Long Integer, Real, Date, Time, and Boolean fields.

Examples

1. This example sets the variables vType, vLength, and vIndex to the properties for the third field of the first table:

```
⇒	GET FIELD PROPERTIES(1; 3;vType;vLength;vIndex)
```

2. This example sets the variables vType, vLength, and vIndex to the properties for the field named [Table3]Field2:

```
⇒	GET FIELD PROPERTIES(->[File3]Field2;vType;vLength;vIndex)
```

See Also

Field, Field name, SET INDEX.

SET INDEX

version 3

SET INDEX (field; index{; *})

Parameter	Type		Description
field	Field or Subfield	→	Field for which to create or delete the index
index	Boolean	→	Create index (TRUE) or Delete index (FALSE)
*		→	Asynchronous indexing if * is passed

Description

The command SET INDEX creates or removes the index for the field or subfield you pass in field.

To index the field or subfield, pass TRUE in index. If the index already exists, the call has no effect. To delete the index, pass FALSE. If the index does not exist, the call has no effect.

Since indexing is done in a separate process, the database remains available for use during this time. If an operation that uses the index is executed while the index is being built, the index will not be used. To determine if a field has been indexed, use the GET FIELD PROPERTIES command.

SET INDEX will not index locked records; it will wait until the record becomes unlocked.

The optional * parameter indicates an asynchronous (simultaneous) indexing. Asynchronous indexing allows the execution of the calling method to continue immediately, whether or not indexing is completed. However, execution will halt at any command that requires the index.

Example

The following example indexes the field [Customers]ID:

```
	UNLOAD RECORD([Customers])
⇒	SET INDEX ([Customers]ID; True)
```

See Also

GET FIELD PROPERTIES, ORDER BY, QUERY.

45 Subrecords

CREATE SUBRECORD

CREATE SUBRECORD (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable for which to create a new subrecord

Description

CREATE SUBRECORD creates a new subrecord for subtable and makes the new subrecord the current subrecord. The new subrecord is saved only when the parent record is saved. The parent record can be saved by a command such as SAVE RECORD or by the user accepting the record. If there is no current record, CREATE SUBRECORD has no effect. To add a new subrecord through a subrecord input form, use ADD SUBRECORD.

Example

The following example is an object method for a button. When it is executed (that is, when the button is clicked), it creates new subrecords for children in the [People] table. The Repeat loop lets the user add children until the Cancel button is clicked. The form displays the children in an subform, but will not allow direct data entry into the subtable because the Enterable option has been turned off:

```
      Repeat
              ` Get the child's name
          vChild := Request("Name (cancel when done):")
              ` If the user clicked OK
          If (OK = 1)
                  ` Add a new subrecord for a child
⇒             CREATE SUBRECORD([People]Children)
                  ` Assign child's name to the subfield
              [People]Children'Name:=vChild
          End if
      Until (OK=0)
```

See Also

ADD SUBRECORD, DELETE SUBRECORD, SAVE RECORD.

DELETE SUBRECORD

version 3

DELETE SUBRECORD (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable from which to delete the current subrecord

Description

DELETE SUBRECORD deletes the current subrecord of subtable. If there is no current subrecord, DELETE SUBRECORD has no effect. After the subrecord is deleted, the current subselection for subtable is empty. As a result, DELETE SUBRECORD can't be used to scan through a subselection and delete selected subrecords.

The deletion of subrecords is not permanent until the parent record is saved. Deleting a parent record automatically deletes all its subrecords.

To delete a subselection, create the subselection you want to delete, delete the first subrecord, create the subselection again, delete the first subrecord, and so on.

Examples

1. The following example deletes all the subrecords of a subtable:

```
        ALL SUBRECORDS([People]Children)
        While (Records in subselection([People]Children)>0)
⇒           DELETE SUBRECORD([People]Children)
            ALL SUBRECORDS([People]Children)
        End while
```

2. The following example deletes the subrecords in which the age of the child is greater than or equal to 12, from the [People]Children subtable :

```
        ALL RECORDS([People])  ` Select all the records
        For ($vlRecord;1;Records in selection([People]))  ` For all the records in the table
                ` Query all records that have subrecords with the criteria
            QUERY SUBRECORDS([People]Children;[People]Children'Age>=12)
                ` Loop until no subrecords are left by the query
            While (Records in subselection([People]Children)>0)
                    ` Delete the subrecord
⇒               DELETE SUBRECORD([People]Children)
                    ` Query again
                QUERY SUBRECORDS([People]Children;[People]Children'Age>=12)
            End while
            SAVE RECORD([People])  ` Save the parent record
            NEXT RECORD([People])
        End for
```

See Also

ALL SUBRECORDS, QUERY SUBRECORDS, Records in subselection, SAVE RECORD.

ALL SUBRECORDS

ALL SUBRECORDS (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable in which to select all subrecords

Description

ALL SUBRECORDS makes all the subrecords of subtable the current subselection. If a current parent record does not exist, ALL SUBRECORDS has no effect. When a parent record is first loaded, the subselection contains all subrecords. A subselection may not contain all subrecords after ADD SUBRECORD, QUERY SUBRECORDS, or DELETE SUBRECORD is executed.

Example

The following example selects all subrecords to ensure that they are all included in the sum:

```
⇒	ALL SUBRECORDS ([Stats]Sales)
	TotalSales := Sum ([Stats]Sales'Dollars)
```

See Also

QUERY SUBRECORDS, Records in subselection.

Records in subselection

Records in subselection (subtable) → Number

Parameter	Type		Description
subtable	Subtable	→	Subtable for which to count the number of selected subrecords
Function result	Number	←	Number of subrecords in current subselection

Description

Records in subselection returns the number of subrecords in the current subselection of subtable. Records in subselection applies only to subrecords in the current record. It is the subrecord equivalent of Records in selection. The result is undefined if no parent record exists.

Example

The following example selects all the subrecords and displays the number of children for the parent record:

```
	` Select all children, then display how many
	ALL SUBRECORDS ([People]Children)
⇒	ALERT ("Number of children: "+String(Records in subselection ([People]Children)))
```

See Also

ALL SUBRECORDS, QUERY SUBRECORDS.

APPLY TO SUBSELECTION

version 3

APPLY TO SUBSELECTION (subtable; statement)

Parameter	Type		Description
subtable	Subtable	→	Subtable to which to apply the formula
statement	Statement	→	One line of code or a method

Description

APPLY TO SUBSELECTION applies statement to each subrecord in the current subselection of subtable. The statement may be a statement or a method. If the statement modifies a subrecord, the modified subrecord is written to disk only when the parent record is written. If the subselection is empty, APPLY TO SUBSELECTION has no effect.

APPLY TO SUBSELECTION can be used to gather information from the subselection or to modify the subselection.

Example

The following example capitalizes the first names in [People]Children:

```
        ALL SUBRECORDS ([People]Children)
⇒       APPLY TO SUBSELECTION([People]Children;[People]Children'Name:=
                                            Uppercase(Substring([People]Children'Name;1;1))
                                            +Lowercase(Substring([People]Children'Name;2)))
```

Note: The statement has been put on several lines for clarity in documentation only.

See Also

ALL SUBRECORDS, QUERY SUBRECORDS, SAVE RECORD.

FIRST SUBRECORD

version 3

FIRST SUBRECORD (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable in which to move to the first selected subrecord

Description

FIRST SUBRECORD makes the first subrecord of the current subselection of subtable the current subrecord. All query, selection, and order by commands also set the current subrecord to the first subrecord. If the current subselection is empty, FIRST SUBRECORD has no effect.

Example

The following example concatenates the first and last names in child records stored in a subtable. It copies the names into the array atNames:

```
          ` Create an array to hold the names
      ARRAY TEXT (atNames; Records in subselection ([People]Children))
⇒     FIRST SUBRECORD ([People]Children)  ` Start at the first subrecord and loop once for
each child
      For ($vlSub; 1; Records in subselection ([People]Children))
          atNames{$vlSub} := [People]Children'First Name+" "+ [People]Children'Last Name
          NEXT SUBRECORD ([People]Children)
      End for
```

See Also

LAST SUBRECORD, NEXT SUBRECORD, PREVIOUS SUBRECORD.

LAST SUBRECORD

LAST SUBRECORD (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable in which to move to the last selected subrecord

Description

LAST SUBRECORD makes the last subrecord of the current subselection of subtable the current subrecord. If the current subselection is empty, LAST SUBRECORD has no effect.

Example

The following example concatenates the first and last names in child records stored in a subtable. It copies the names into an array, called atNames. It is the same as the example for FIRST SUBRECORD except that it moves through the subrecords from last to first:

```
          ` Create an array to hold the names
      ARRAY TEXT (atNames; Records in subselection ([People]Children))
          ` Start at the last subrecord and loop once for each child
⇒     LAST SUBRECORD ([People]Children)
      For ($vlSub;1;Records in subselection ([People]Children))
          atNames{$vlSub}:=[People]Children'First Name + " " + [People]Children'Last Name
          PREVIOUS SUBRECORD ([People]Children)
      End for
```

See Also

FIRST SUBRECORD, NEXT SUBRECORD, PREVIOUS SUBRECORD.

NEXT SUBRECORD

version 3

NEXT SUBRECORD (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable in which to move to the next selected subrecord

Description

NEXT SUBRECORD moves the current subrecord pointer to the next subrecord in the current subselection of subtable. If NEXT SUBRECORD moves the pointer past the last subrecord, End subselection returns TRUE, and there is no current subrecord. If End subselection returns TRUE, use FIRST SUBRECORD or LAST SUBRECORD to move the pointer back into the current subselection. If the current subselection is empty, or Before subselection returns TRUE, NEXT SUBRECORD has no effect.

Example

See the example for FIRST SUBRECORD.

See Also

FIRST SUBRECORD, LAST SUBRECORD, PREVIOUS SUBRECORD.

PREVIOUS SUBRECORD

version 3

PREVIOUS SUBRECORD (subtable)

Parameter	Type		Description
subtable	Subtable	→	Subtable in which to move to the previous selected subrecord

Description

PREVIOUS SUBRECORD moves the current subrecord pointer to the previous subrecord in the current subselection of subtable. If PREVIOUS SUBRECORD moves the pointer before the first subrecord, Before subselection returns TRUE, and there is no current subrecord. If Before subselection returns TRUE, use FIRST SUBRECORD or LAST SUBRECORD to move the pointer back into the current subselection. If the current subselection is empty, or End subselection returns TRUE, PREVIOUS SUBRECORD has no effect.

Example

See the example for LAST SUBRECORD.

See Also

FIRST SUBRECORD, LAST SUBRECORD, NEXT SUBRECORD.

Before subselection

Before subselection (subtable) → Boolean

Parameter	Type		Description
subtable	Subtable	→	Subtable for which to test if subrecord pointer is before the first selected subrecord
Function result	Boolean	←	Yes (TRUE) or No (FALSE)

Description

Before subselection returns True when the current subrecord pointer is before the first subrecord of subtable. Before subselection is used to check whether or not PREVIOUS SUBRECORD has moved the pointer before the first subrecord. If the current subselection is empty, Before subselection returns True.

Example

The following example is an object method for a button. When the button is clicked, the pointer moves to the previous subrecord. If the pointer is before the first subrecord, it moves to the last subrecord:

```
   PREVIOUS SUBRECORD ([People]Children)   ` Move to the previous subrecord
⇒  If (Before subselection ([People]Children)   ` If we have gone too far...
      LAST SUBRECORD ([People]Children)   ` move to the last subrecord
   End if
```

See Also

PREVIOUS SUBRECORD.

End subselection

End subselection (subtable) → Boolean

Parameter	Type		Description
subtable	Subtable	→	Subtable for which to test if subrecord pointer is after the last selected subrecord
Function result	Boolean	←	Yes (TRUE) or No (FALSE)

Description

End subselection returns True when the current subrecord pointer is after the end of the current subselection of subtable. End subselection is used to check whether or not NEXT SUBRECORD has moved the pointer after the last subrecord. If the current subselection is empty, End subselection returns True.

Example

The following example is an object method for a button. When the button is clicked, the pointer moves to the next subrecord. If the pointer is after the last subrecord, it moves to the first subrecord:

```
   NEXT SUBRECORD ([People]Children)   ` Move to the next subrecord
⇒  If (End subselection ([People]Children))   ` If we have gone too far...
      FIRST SUBRECORD ([People]Children)   ` move to the first subrecord
   End if
```

See Also

NEXT SUBRECORD.

ORDER SUBRECORDS BY

version 3

ORDER SUBRECORDS BY (subtable; subfield{; > or <}{; subfield2; > or <2; ...; subfieldN; > or <N})

Parameter	Type		Description
subtable	Subtable	→	Subtable by which to order the selected subrecords
subfield	Subfield	→	Subfield on which to order by for each level
> or <		→	Ordering direction for each level: > to order in ascending order or < to order in descending order

Description

ORDER SUBRECORDS BY sorts the current subselection of subtable. It sorts only the subselection of the subtable contained in the current parent record.

The direction parameter specifies whether to sort subfield in ascending or descending order. If direction is the “greater than” symbol (>), the subrecords are ordered in ascending order. If direction is the “less than” symbol (<), the subrecords are ordered in descending order.

You can specify more than one level of sort by including more subfields and sort symbols. After the sort is completed, the first subrecord of the sorted subselection is the current subrecord. Sorting subrecords is a dynamic process. Subrecords are never saved in their sorted order. If neither a current record nor a higher-level subrecord exists, ORDER SUBRECORDS BY has no effect.
If a form contains a subform that is to be printed in a fixed frame, this command needs to be called just once before printing in the Before phase of the parent form method.

Example

The following example sorts the [Stats]Sales subtable into ascending order, based on the SalesDollars subfield:

```
⇒	ORDER SUBRECORDS BY ([Stats]Sales; [Stats]SalesDollars; >)
```

See Also

QUERY SUBRECORDS.

QUERY SUBRECORDS

version 3

QUERY SUBRECORDS (subtable; queryFormula)

Parameter	Type		Description
subtable	Subtable	→	Subtable to search
queryFormula	Boolean	→	Query formula

Description

QUERY SUBRECORDS queries subtable and creates a new subselection. This is the only command that queries subrecords and returns a selection of subrecords. The queryFormula is applied to each subrecord in subtable. If the formula evaluates as TRUE, the subrecord is added to the new subselection. When the query is complete, QUERY SUBRECORDS makes the first subrecord the current subrecord of subtable.

Remember that QUERY SUBRECORDS queries only the subrecords of the subtable contained in the currently selected parent record, and not all the subrecords associated with the records of the parent table. QUERY SUBRECORDS does not change the current parent record.

Typically, queryFormula tests a subfield against a variable or a constant, using a relational operator. The queryFormula can contain multiple tests that are joined by AND conjunctions (&) or OR conjunctions (|). Also, the queryFormula can be a function or contain a function. The wildcard character (@) can be used with string arguments.

If neither a current record nor a higher-level subrecord exists, QUERY SUBRECORDS has no effect.

Example

The following example queries for children older than 10 years:

```
⇒	QUERY SUBRECORDS ([People]Children; [People]Children'Age>10)
```

See Also

ALL SUBRECORDS, ORDER SUBRECORDS BY, Records in subselection.

46 System Documents

Introduction

All the documents and applications you use on your computer are stored as **files** on the hard disk(s) **connected** to or **mounted** on your machine, or floppy disk(s) or other similar permanent storage devices. Within 4th Dimension we may indistinctly speak of **file** or **document** when referring to these documents and applications. However most of the commands of this theme use the term document because most of the time you will use them to access documents (in opposition to application or system files) on disk.

A hard disk can be formatted as one or several partitions. Each partition is named a **volume**. No matter if two volumes are partitions physically present on a same hard disk or not, at the 4D level, you will usually treat these volumes as separate and equal entities.

A volume can be located on a hard disk physically connected to you machine or mounted over the network through a file sharing protocol, such as NetBEUI (Windows) or AFP (Macintosh). Whatever the case, at the 4D level, you treat these volumes in the same way while using the System Documents commands (unless you know what you do and use 4D Plug-ins to extend the capability of your application in that domain).

Each volume has a **volume name**. On Windows, volumes are designated by a letter followed by a colon. Usually A: and B: are used to designate the 5 1/4 or 3 1/2 floppy drives. Usually C: designates the volume your use for booting your system (unless you configure your PC otherwise). Then the letters D: through Z: are used for the additional volumes connected or mounted to your PC (CD-ROM drives, additional drives, network drives and so on). On Macintosh, volumes have natural names whose maximal length is 31 characters (these names are the names you see on the desktop at the Finder level.

Normally, you class your documents into **folders** which themselves can contain other folders. It is not a good idea to accumulate hundreds or thousands of files at the same level of a volume, first it is messy, second it slows down your system. On Windows, a folder is or was called a **directory** (although the term folder is more in use since the introduction of Windows 95) and it is usually called folder on Macintosh.

To unique identify a document, you consequently need to know the name of the volume and the name(s) of the folder(s) where the document is located as well as the name of the document itself. If you concatenate all these names you get the **path name** to the document. Within this pathname, folder name are separated by a special character called the **directory (separator) symbol**. On Windows this character is the anti-slash \, on Macintosh it is the colon :.

Let us look at an example. You have a document Important Memo located in the folder Memos, itself located in the folder Documents, itself located in the folder Current Work.

If, on Windows, the whole thing is located on the C: drive (volume), the path name to the document is therefore:

C:\Current Work\Documents\Memos\Important Memo.TXT

If, on Macintosh, the whole thing is located on the disk (volume) Internal Drive, the path name to the document is therefore:

Internal Drive:Current Work:Documents:Memos:Important Memo

Note that, on windows, with this example, the name of the document is suffixed with .TXT, we will see why in the next paragraphs.

Whatever the platform, the path name to a document can be formally expressed the following way: VolName DirSep { DirName DirSep { DirName DirSep { ... } } } DocName.

All the documents (files) located on volumes have several characteristics that are usually called **attributes** or **properties**:

The **name** of the document itself (up to 31 characters on Macintosh, up to 8 characters on Windows 3.1.1, up to 255 characters on Windows 95 or NT 4.0)

Document Type and Creator

On Windows a document has a **type**. On Macintosh a document has a **type** and a **creator**. The type of a document generally indicates what the document is or what it contains. For instance, a text document contains some text (without styles variations). On Windows, the type of a document is determined by the suffix called **File extension** attached to the name of the document. For instance, .TXT is the Windows file extension for text documents. On Macintosh, the type of a document is determined by the **file type** property of the document which is a 4-character signature (not displayed at the Finder level). For instance, the file type of a text document is "TEXT". In addition, on Macintosh, a document has a **creator** which designates the application which created the document. This notion does not exist on Windows. The creator of a document is determined by the **file creator** property of the document which is a 4-character signature (not displayed at the Finder level). For instance, the file creator of a document created by 4D V6 is "4D06".

DocRef: Document reference number

A document is **open** or **closed**. Using the built-in 4D commands, a document can be open by only one process at a time. One process can open several documents, several processes can open multiple documents, but you cannot open the same document twice at a time.

You open a document with the commands Open document, Create document and Append document.
Once a document is open, you can read and write characters from and to the document (see command RECEIVE PACKET and SEND PACKET). When you are done with the document, you usually close it using the command CLOSE DOCUMENT.

All open document are referred to using **DocRef** expression returned by the commands Open document, Create document and Append document. A DocRef uniquely identifies an open document. It is formally an expression of type Time. All commands working with open documents, expects a DocRfe as parameter. If you pass an incorrect DocRef to one of these commands, a file manager error occurs.

Handling I/O errors

When you access (open, close, delete, rename, copy) documents, when you change the properties of a document or when you read and write characters in a document, I/O errors may occur. A document might not be found, it can be locked, it can be already open. You can catch these errors with an error-handling method installed with ON ERR CALL. Most the errors that can occur while using system documents are described in the section OS File Manager Errors.

The Document system variable

The three commands Open document, Create document and Append document enables you to access a document using the standard Open or Save file dialog boxes. When you access a document through a standard dialog, 4D returns you the full pathname to the document in the Document system variable. This system variable has to be distinguished from the document parameter that appears in the parameter list of the commands.

Specifying Document names or Document pathnames

Most of the routines of this section expecting a document name accept both a name or a pathname to the document (*). If you pass a name, the command looks for the document within the folder of the database. If you pass a pathname, it must be valid.

If you pass a wrong name or a wrong pathname, the command generates a file manager error that you can intercept using an ON ERR CALL method.

(*) except when signaled otherwise.

Warning: The maximum length of the parameter document is 255 characters. If you pass a longer name, it will be truncated and a File manager error will be generated.

Useful Project Methods when handling documents on disk

• Detecting on which platform you're running

Although 4th Dimension provides commands, such as MAP FILE TYPES, for eliminating coding variations due to platform specificities, once you start to work at a lower level when handling documents on disk, such as obtaining programmatically path names, you need to know if you are running on a Macintosh or on a Windows platform.

The On Windows project method listed below tells if your database is running on Windows:

```
	` On windows Project Method
	` On windows -> Boolean
	` On windows -> True if on Windows
C_BOOLEAN($0)
C_LONGINT($vlPlatform;$vlSystem;$vlMachine)
PLATFORM PROPERTIES($vlPlatform;$vlSystem;$vlMachine)
$0:=($vlPlatform=Windows)
```

• Using the right directory separator symbol

On Windows, a directory level is symbolized by an anti-slash \. On Macintosh, a folder level is symbolized by a colon :. Depending on which platform you are running, the Directory symbol project method listed below returns you the ASCII code of the right directory symbol (character).

```
	` Directory symbol Project Method
	` Directory symbol -> Integer
	` Directory symbol -> ASCII of "/" (Windows) or ":" (MacOS)
C_INTEGER($0)
If (On Windows )
	$0:=Ascii("/")
Else
	$0:=Ascii(":")
End if
```

• Extracting the file name from a long name

Once you have obtain the long name (path name + file name) to a document, you may need to extract from that long name the file name of the document to, for instance, display it in the title of a window. The Long name to file name project method does exactly this on both Windows and Macintosh.

```
      ` Long name to file name Project Method
      ` Long name to file name ( String ) -> String
      ` Long name to file name ( Long file name ) -> file name

C_STRING(255;$1;$0)
C_INTEGER($viLen;$viPos;$viChar;$viDirSymbol)

$viDirSymbol:=Directory symbol
$viLen:=Length($1)
$viPos:=0
For ($viChar;$viLen;1;-1)
    If (Ascii($1[[$viChar]])=$viDirSymbol)
        $viPos:=$viChar
        $viChar:=0
    End if
End for
If ($viPos>0)
    $0:=Substring($1;$viPos+1)
Else
    $0:=$1
End if
      ` Set the variable <>vbDebugOn to True or False in the On Startup database method
If (<>vbDebugOn)
    If ($0="")
        TRACE
    End if
End if
```

• Extracting the file name from a long name

Once you have obtain the long name (path name + file name) to a document, you may need to extract from that long name the path name to the directory where the document is located to , for instance, save additional documents at the same place. The Long name to path name project method does exactly this on both Windows and Macintosh.

```
    ` Long name to path name Project Name
    ` Long name to path name ( String ) -> String
    ` Long name to path name ( Long file name ) -> Path name

C_STRING(255;$1;$0)
C_STRING(1;$vsDirSymbol)
C_INTEGER($viLen;$viPos;$viChar;$viDirSymbol)

$viDirSymbol:=Directory symbol
$viLen:=Length($1)
$viPos:=0
For ($viChar;$viLen;1;-1)
    If (Ascii($1[[$viChar]])=$viDirSymbol)
        $viPos:=$viChar
        $viChar:=0
    End if
End for
If ($viPos>0)
    $0:=Substring($1;1;$viPos)
Else
    $0:=$1
End if
    ` Set the variable <>vbDebugOn to True or False in the On Startup database method
If (<>vbDebugOn)
    If ($0="")
        TRACE
    End if
End if
```

See Also

Append document, CLOSE DOCUMENT, COPY DOCUMENT, Create document, CREATE FOLDER, DELETE DOCUMENT, Document creator, DOCUMENT LIST, Document type, FOLDER LIST, Get document position, GET DOCUMENT PROPERTIES, Get document size, MAP FILE TYPES, MOVE DOCUMENT, Open document, SET DOCUMENT CREATOR, SET DOCUMENT POSITION, SET DOCUMENT PROPERTIES, SET DOCUMENT SIZE, SET DOCUMENT TYPE, Test path name, VOLUME ATTRIBUTES, VOLUME LIST.

Open document

version 3

Open document (document{; fileType}) → DocRef

Parameter	Type		Description
document	String	→	Document name or Full document pathname or Empty string for standard file dialog box
fileType	String	→	MacOS file type (4-character string) or Windows file extension (1 to 3-character string) or TEXT (.TXT) document if omitted
Function result	DocRef	←	Document reference number

Description

The command Open document opens the document whose name or pathname you pass in document.

If you pass an empty string in document, the Open File dialog box is presented, you then select the document to be open. If you cancel the dialog, no document is open, Open document returns a null DocRef and sets the OK variable to 0.

If the document is correctly opened, Open document returns its document reference number and set the OK variable to 1. If the document does not exist or is already open an error is generated.

On Macintosh, if you use the Open File dialog box, all documents are by default presented. To show another type of documents, specify the a document type in the optional fileType parameter.

On Windows, if you use the Open File dialog box, all types of documents *.* are by default presented. To show another type of documents, pass in fileType, a 1 to 3-character Windows file extension or a Macintosh file type mapped using the command MAP FILE TYPES.

On Windows, even though you do not use the Open File dialog box, you might pass the fileType parameter to specify the file extension of the document you want to open. By default, Open document attempts to open a .TXT file. If you specify the fileType parameter, Open document tries to open the document whose name is “Document.fileType”.

For example:

```
⇒       vhDocRef:=Open document("C:\Letter";"WRI")
```

will try to open the document “C:\Letter.WRI” on your disk. If you pass more than three characters in fileType, Open document takes into account only the first three characters. If a document type is not specified, Open document tries to open the document with no file extension. If it does not find it, it tries to open the document with the .TXT extension. If it does not find it, it will return a “File not found” error.

If a document is open, Open document initially sets the file position at the beginning of the document while Append document sets it at the end of the document.

Once you have open a document you can read and write in the document using the command RECEIVE PACKET and SEND PACKET that you can combine with the commands Get file position and SET FILE POSITION to directly access any part of the document.

Do not forget to eventually call CLOSE DOCUMENT for the document.

Example

The following example opens an existing document called Note, writes the string “Good-bye” into it, and closes the document. If the document already contained the string “Hello”, the string would be overwritten:

```
        C_TIME(vhDocRef)
⇒       vhDocRef:=Open document ("Note")   ` Open a document called Note
        If (OK=1)
            SEND PACKET (vhDocRef;"Good-bye")   ` Write one word into the document
            CLOSE DOCUMENT (vhDocRef)  ` Close the document
        End if
```

See Also

Append document, Create document, Open document.

version 3

Create document (document{; type}) → DocRef

Parameter	Type		Description
document	String	→	Document name or Full document pathname or Empty string for standard file dialog box
type	String	→	MacOS file type (4-character string) or Windows file extension (1 to 3-character string) or TEXT (.TXT) document if omitted
Function result	DocRef	←	Document reference number

Description

The command Create document creates a new document and returns its document reference number.

You pass the name or the full pathname of the new document in document. If document already exists on the disk, it is overwritten. However, if document is locked or already open, an error is generated.

If you pass an empty in document, the Save As dialog box is presented, you can then enter the name of the document you want to create. If you cancel the dialog, no document is created, Create document returns a null DocRef and sets the OK variable to 0.

If the document is correctly created and open, Create document returns its document reference number and set the OK variable to 1.

Whether or not you use the Save As dialog box, Create document creates by default a .TXT (Windows) or TEXT (Macintosh) document. If you want to create another type of document, pass the fileType parameter.

On Macintosh, you pass a file type. On Windows you pass a 1 to 3-character Windows file extension or Macinsoth file type mapped through the MAP FILE TYPES mechanism.

Once you have created and open a document you can write and read the document using the command SEND PACKET and RECEIVE PACKET that you can combine with the commands Get file position and SET FILE POSITION to directly access any part of the document.

Do not forget to eventually call CLOSE DOCUMENT for the document.

Example

The following example creates and opens a new document called Note, writes the string “Hello” into it, and closes the document:

```
        C_TIME(vhDocRef)
⇒       vhDocRef:=Create document ("Note")  ` Create new document called Note
        If (OK=1)
            SEND PACKET(vhDocRef; "Hello")  ` Write one word into the document
            CLOSE DOCUMENT(vhDocRef)  ` Close the document
        End if
```

See Also

Append document, Open document.

Append document

version 3

Append document (document{; type}) → DocRef

Parameter	Type		Description
document	String	→	Document name or Full document pathname or Empty string for standard file dialog box
type	String	→	MacOS file type (4-character string) or Windows file extension (1 to 3-character string) or TEXT (.TXT) document if omitted
Function result	DocRef	←	Document reference number

Description

The command Append document does the same as thing as Open document: it allows you to open a document on disk.

The only difference is that Append document sets the file position at the end of the document while Open document sets its at the beginning of the document.

Refer to Open document for more details about using Append document.

Example

The following example opens an existing document called Note, appends the string "and so long" and a carriage return onto the end of the document, and closes the document. If the document already contained the string "Good-bye", the document would now contain the string "Good-bye and so long", followed by a carriage return:

```
     C_TIME(vhDocRef)
⇒    vhDocRef:=Append document ("Note")  ` Open Note document
     SEND PACKET (vhDocRef;" and so long"+Char(13))  ` Append a string
     CLOSE DOCUMENT (vhDocRef)  ` Close the document
```

See Also

Create document, Open document.

version 3

CLOSE DOCUMENT (docRef)

Parameter	Type		Description
docRef	DocRef	→	Document reference number

Description

CLOSE DOCUMENT closes the document specified by docRef.

Closing a document is the only way to ensure that the data written to a file is saved. You must close all the documents you open with the commands Open document, Create document or Append document.

Example

The following example lets the user create a new document, writes the string “Hello” into it, and closes the document:

```
     C_TIME(vhDocRef)
     vhDocRef:=Create document ("")
     If (OK=1)
        SEND PACKET(vhDocRef; "Hello")  ` Write one word into the document
⇒       CLOSE DOCUMENT(vhDocRef)  ` Close the document
     End if
```

See Also

Append document, Create document, Open document.

COPY DOCUMENT

COPY DOCUMENT (sourceName; destinationName{; *})

Parameter	Type		Description
sourceName	String	→	Name of document to be copied
destinationName	String	→	Name of copied document
*		→	Override existing document if any

Description

The command COPY DOCUMENT copies the document specified by sourceName to the location specified by destinationName.

Both sourceName and destinationName can be a name referring to a document located in the database folder or a pathname referring to a document relatively to the root level of a volume.

An error will occur if there is already a document named destinationName unless you specify the optional * parameter instructing COPY DOCUMENT to delete and override the destination document.

Examples

(1) The following example duplicates a document in its own folder:

```
⇒	COPY DOCUMENT("C:\FOLDER\DocName";"C:\FOLDER\DocName2")
```

(2) The following example copies a document to the database folder (provided C:\FOLDER is not the database folder):

```
⇒	COPY DOCUMENT("C:\FOLDER\DocName";"DocName")
```

(3) The following example copies and from a document from one volume to another one:

```
⇒	COPY DOCUMENT("C:\FOLDER\DocName";"F:\Archives\DocName.OLD")
```

(4) The following example duplicates a document in its own folder overriding an already existing copy:

```
⇒	COPY DOCUMENT("C:\FOLDER\DocName";"C:\FOLDER\DocName2";*)
```

See Also

MOVE DOCUMENT.

MOVE DOCUMENT

version 6.0

MOVE DOCUMENT (srcPathname; dstPathname)

Parameter	Type		Description
srcPathname	String	→	Full pathname to existing document
dstPathname	String	→	Destination pathname

Description

The command MOVE DOCUMENT moves or renames a document.

You specify the full pathname to the document in srcPathName and the new name and/or new location for the document in dstPathName.

Warning: Using MOVE DOCUMENT, you can move a document from and to any directory on the same volume. If you want to move a document between two distinct volumes, use COPY DOCUMENT to “move” the document then delete the original copy of the document using DELETE DOCUMENT.

Examples

(1) The following example renames the document DocName:

```
⇒	MOVE DOCUMENT("C:\FOLDER\DocName";"C:\FOLDER\NewDocName")
```

(2) The following example moves and renames the document DocName:

```
⇒	MOVE DOCUMENT("C:\FOLDER1\DocName";"C:\FOLDER2\NewDocName")
```

(3) The following example moves the document DocName:

```
⇒	MOVE DOCUMENT("C:\FOLDER1\DocName";"C:\FOLDER2\DocName")
```

Note: In the last two example, the destination folder "C:\FOLDER2" must exist. The command MOVE DOCUMENT only moves a document, does not create folders.

See Also

COPY DOCUMENT.

DELETE DOCUMENT

DELETE DOCUMENT (document)

Parameter	Type		Description
document	String	→	Document name or Full document pathname

Description

The command DELETE DOCUMENT deletes the document whose name you pass in document.

If the document does not exist, no error is generated. On the other hand, an error is generated if you try to delete an open document.

DELETE DOCUMENT doesn't accept an empty string argument for document. If an empty string is used, the Open File dialog box is not displayed and an error is generated.

WARNING: DELETE DOCUMENT can delete any file on a disk. This includes documents created with other applications as well as the applications themselves. DELETE DOCUMENT should be used with extreme caution. Deleting a document is a permanent operation and cannot be undone.

Examples

(1) The following example deletes the document named Note:

```
⇒	DELETE DOCUMENT ("Note") ` Delete the document
```

(2) See example for the command APPEND TO CLIPBOARD.

System Variables or Sets

Deleting a document sets the OK system variable to 1. If DELETE DOCUMENT can't delete the document, the OK system variable is set to 0.

Test path name

version 6.0

Test path name (pathname) → Number

Parameter	Type		Description
pathname	String	→	Pathname to directory, folder or document
Function result	Number	←	1, pathname refers to an existing document 0, pathname refers to an existing directory or folder <0, invalid pathname, OS file manager error code

Description

The function Test path name checks if a document or folder whose name or pathname you pass in pathname is present on the disk.

If a document is found, Test path name returns 1. If a folder found, Test path name returns 0.

The following predefined constant are provided by 4D:

Constant	Type	Value
Is a document	Long Integer	1
Is a directory	Long Integer	0

If no document nor folder is found, Test path name returns a negative value (i.e. -43 for File not found).

Example

The following tests if the document "Journal" is present in the folder of the database, then creates it if it was not found:

```
⇒	If (Test path name("Journal") # Is a document)
		$vhDocRef:=Create document("Journal")
		If (OK=1)
			CLOSE DOCUMENT($vhDocRef)
		End if
	End if
```

See Also

Create document, CREATE FOLDER.

version 6.0

CREATE FOLDER (folderPath)

Parameter	Type		Description
folderPath	String	→	Pathname to new folder to create

Description

The command CREATE FOLDER creates a folder according to the pathname you pass in folderPath.

If you pass a name, the folder is created in the folder of the database. If you pass a path name, it must refer to a valid path up to the name of the folder you want to create; starting at the root level of a volume or at the level of the database folder.

Examples

(1) The following example creates the folder “Archives” in the folder of the database:

```
⇒	CREATE FOLDER("Archives")
```

(2) The following example creates the folder Archives in the folder of the database, then it creates the subfolder “January” and “February”:

```
⇒	CREATE FOLDER("Archives")
⇒	CREATE FOLDER("Archives\January")
⇒	CREATE FOLDER("Archives\February")
```

(3) The following example creates the folder “Archives” at the root level of the C volume:

```
⇒	CREATE FOLDER("C:\Archives")
```

(4) The following example will fail if there is no “NewStuff” folder at the root level of the C volume:

```
⇒	CREATE FOLDER("C:\NewStuff\Pictures") ` WRONG, can't two folder levels in one
call
```

See Also

FOLDER LIST, Test path name.

VOLUME LIST

VOLUME LIST (volumes)

Parameter	Type		Description
volumes	Array	←	Names of the volumes currently mounted

Description

The command VOLUME LIST populates the Text or String array volumes with the names of the volumes currently defined (Windows) or mounted (Macintosh) on your machine.

On Macintosh, it returns the list of the volumes visible at the Finder level.

On the other hand, on Windows, it returns the list of the volumes currently defined whether or not each volume is physically present (i.e. the volume A: will be returned whether or not a disk is actually present in the floppy drive).

Example

Using a scrollable area named asVolumes you want to display the list of the volumes defined or mounted on your machine, you write:

```
      Case of
         : (Form event=On Load)
            ARRAY STRING(31;asVolumes;0)
⇒           VOLUME LIST(asVolumes)

            ` ...
      End case
```

See Also

DOCUMENT LIST, FOLDER LIST, VOLUME ATTRIBUTES.

VOLUME ATTRIBUTES

VOLUME ATTRIBUTES (volume; size; used; free)

Parameter	Type		Description
volume	String	→	Volume name
size	Number	←	Volume size expressed in bytes
used	Number	←	Used space expressed in bytes
free	Number	←	Free space expressed in bytes

Description

The command VOLUME ATTRIBUTES returns, expressed in bytes, the size, the used space and the free space for the volume whose name you pass in volume.

Example

Your application includes some batch operations running the night or the week-end that store huge temporary files on disk. To make this process as automatic and flexible as possible, you write a routine that will automatically find the first volume whose free space is sufficient for your temporary files. You might write the following project method:

```
    ` Find volume for space Project Method
    ` Find volume for space ( Long ) -> String
    ` Find volume for space ( Space needed in bytes ) -> Volume name or Empty string

C_STRING(31;$0)
C_STRING(255;$vsDocName)
C_LONGINT($1;$vlNbVolumes;$vlVolume;$vlSize;$vlUsed;$vlFree)
C_TIME($vhDocRef)

    ` Initialize function result
$0:=""
    ` Protect all I/O operations with an error interruption method
ON ERR CALL("ERROR METHOD")
    ` Get the list of the volumes
ARRAY STRING(31;$asVolumes;0)
gError:=0
VOLUME LIST($asVolumes)
If (gError=0)
        ` If running on windows, skip the (usual) two floppy drives
    If (On Windows )
        $vlVolume:=Find in array($asVolumes;"A:")
```

```
            If ($vlVolume>0)
                DELETE ELEMENT($asVolumes;$vlVolume)
            End if
            $vlVolume:=Find in array($asVolumes;"B:")
            If ($vlVolume>0)
                DELETE ELEMENT($asVolumes;$vlVolume)
            End if
        End if
        $vlNbVolumes:=Size of array($asVolumes)
            ` For each volume
        For ($vlVolume;1;$vlNbVolumes)
                ` Get the size, used space and free space
            gError:=0
⇒           VOLUME ATTRIBUTES($asVolumes{$vlVolume};$vlSize;$vlUsed;$vlFree)
            If (gError=0)
                    ` Is the free space large enough (plus an extra 32K) ?
                If ($vlFree>=($1+32768))
                        ` If so, check if the volume is unlocked...
                    $vsDocName:=$asVolumes{$vlVolume}+Char(Directory symbol )
                                                        +"XYZ"+String(Random)+".TXT"
                    $vhDocRef:=Create document($vsDocName)
                    If (OK=1)
                        CLOSE DOCUMENT($vhDocRef)
                        DELETE DOCUMENT($vsDocName)
                            ` If everything's fine, return the name of the volume
                        $0:=$asVolumes{$vlVolume}
                        $vlVolume:=$vlNbVolumes+1
                    End if
                End if
            End if
        End for
    End if
    ON ERR CALL("")
```

Once this project method is added to your application, you can for instance write:

```
    $vsVolume:=Find volume for space (100*1024*1024)
    If($vsVolume#"")
        ` Continue
    Else
        ALERT("A volume with at least 100 MB of free space is required!")
    End if
```

See Also

VOLUME LIST.

FOLDER LIST

version 6.0

FOLDER LIST (pathname; directories)

Parameter	Type		Description
pathname	String	→	Pathname to volume, directory or folder
directories	Array	←	Names of the directories present at this location

Description

The command FOLDER LIST populates the Text or String array directories with the names of the folders located at the pathname you pass in pathname.

If there are no folders at the specified location, the command returns an empty array. If the pathname you pass in pathname is invalid, FOLDER LIST generate a file manager error that you can intercept using an ON ERR CALL method.

Warning: The maximum length of the parameter pathname is 255 characters. If you pass a longer pathname, it will be truncated and a File manager error will be generated.

See Also

DOCUMENT LIST, VOLUME LIST.

DOCUMENT LIST

version 6.0

DOCUMENT LIST (pathname; documents)

Parameter	Type		Description
pathname	String	→	Pathname to volume, directory or folder
documents	Array	←	Names of the documents present at this location

Description

The command DOCUMENT LIST populates the Text or String array directories with the names of the documents located at the pathname you pass in pathname.

If there are no documents at the specified location, the command returns an empty array. If the pathname you pass in pathname is invalid, DOCUMENT LIST generate a file manager error that you can intercept using an ON ERR CALL method.

Warning: The maximum length of the parameter pathname is 255 characters. If you pass a longer pathname, it will be truncated and a File manager error will be generated.

See Also

FOLDER LIST, FOLDER LIST, VOLUME LIST.

Document type

Document type (document) → String

Parameter	Type		Description
document	String	→	Document name
Function result	String	←	Windows file extension (1 to 3-character string) or MacOS file type (4-character string)

Description

The command Document type returns the type of the document whose name or pathname you pass in document.

On Windows, Document type returns the file extension of the document (i.e. 'DOC' for a Microsoft Word document, 'EXE' for an executable file, and so on) or the corresponding MacOS-based 4 characters file type if this latter has been mapped with its equivalent Windows file extension by 4th Dimension (i.e. 'TEXT' for the 'TXT' file extension) or by a prior call to MAP FILE TYPES.

On Macintosh, Document type returns the 4-characters file type of the document (i.e. 'TEXT' for a Text document, 'APPL' for a double-clickable application and so on).

See Also

Document creator, GET DOCUMENT PROPERTIES, MAP FILE TYPES, SET DOCUMENT TYPE.

SET DOCUMENT TYPE

version 6.0

SET DOCUMENT TYPE (document; fileType)

Parameter	Type		Description
document	String	→	Document name or Full document pathname
fileType	String	→	Windows file extension (1 to 3-character string) or MacOS file type (4-character string)

Description

The command SET DOCUMENT TYPE sets the type of the document whose name or pathname you pass in document.

You pass the new type of the document in fileType.

See discussion about file types in System Documents and Document type.

See Also

Document type, MAP FILE TYPES, SET DOCUMENT CREATOR, SET DOCUMENT PROPERTIES.

version 3.5

MAP FILE TYPES (macOS; windows; context)

Parameter	Type		Description
macOS	String	→	MacOS file type (4-character string)
windows	String	→	Windows file extension (1 to 3-character string)
context	String	→	String displayed in List of Types drop-down list of the Windows file dialog boxes

Description

MAP FILE TYPES lets you associate a Windows file extension with a Macintosh file type.

You need to call this routine only once to establish a mapping for an entire worksession with a database. Once the call has been made, the commands Append document, Create document, Create resource file, Open resource file and Open resource file while running on Windows will automatically substitute the Windows file extension for the Macintosh file type you actually pass as a parameter to the routine.

In the macOS parameter you pass a 4-character Macintosh file type. If you do not pass a 4-character string, the command does nothing and generates an error.

In the windows parameter you pass a 1 to 3-character Windows file extension. If you do not pass a 1 to 3-character string, the command does nothing and generates an error.

In the context parameter you pass the string that will be displayed in the List Files of Type drop-down list of the Windows Open File dialog box. The context string is limited to 32 characters; additional characters are ignored.

IMPORTANT: Once you have mapped a Windows file extension to a Macintosh file type, you cannot change or delete this mapping within a single work session. If you need to change a mapping while developing and debugging a 4D application, reopen the database and remap the file extension.

Example

The following line of 4D code (that could be part of the Startup database method) maps the Macintosh MS-Word file type “WDBN” to the Windows file extension “.DOC”:

```
⇒	MAP FILE TYPES ("WDBN";"DOC";"Word documents")
```

Once the call above has been made, the following code will display only Word documents in the Open file dialog on Windows and Macintosh:

```
	$DocRef:=Open document("";"WDBN")
	If (OK=1)
		` ...
	End if
```

See Also

Append document, Create document, Create resource file, Open resource file, Open resource file.

Document creator System documents

version 6.0

Document creator (document) → String

Parameter	Type		Description
document	String	→	Document name or Full document pathname
Function result	String	←	Empty string (Windows) or File Creator (MacOS)

Description

The command Document creator returns the creator of the document whose name or pathname you pass in document.

On Windows, Document creator returns an empty string.

See Also

Document type, SET DOCUMENT CREATOR.

SET DOCUMENT CREATOR System Documents

version 6.0

SET DOCUMENT CREATOR (document; fileCreator)

Parameter	**Type**		**Description**
document	String	→	Document name or Full document pathname
fileCreator	String	→	MacOS file creator (4-character string) or empty string (Windows)

Description

The command SET DOCUMENT CREATOR sets the creator of the document whose name or pathname you pass in document.

You pass the new creator of the document in fileCreator.

This command does nothing on Windows.

See discussion about file creators in System Documents.

See Also

Document creator, SET DOCUMENT PROPERTIES, SET DOCUMENT TYPE.

GET DOCUMENT PROPERTIES

version 6.0

GET DOCUMENT PROPERTIES (document; locked; invisible; created on; created at; modified on; modified at)

Parameter	Type		Description
document	String	→	Document name
locked	Boolean	←	Locked (True) or unlocked (False)
invisible	Boolean	←	Invisible (True) or visible (False)
created on	Date	←	Creation date
created at	Time	←	Creation time
modified on	Date	←	Last modification date
modified at	Time	←	Last modification time

Description

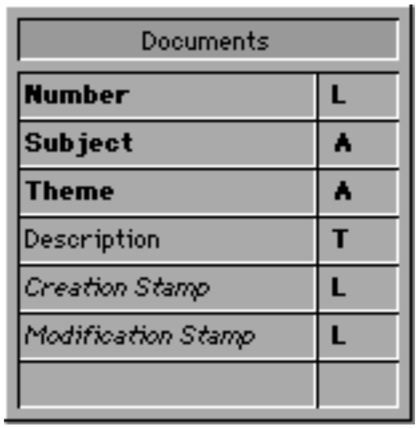
The command GET DOCUMENT PROPERTIES returns information about the document whose name or pathname you pass in document.

After the call:
• Locked returns True if the document is locked. A locked document cannot be open nor deleted.
• Invisible returns True if the document is hidden.
• created on and created at return the date and time when the document was created.
• modified on and modified at return the date and time when the document modified for the last time.

Example

You have created a documentation database and you would like to export all the records you created in the database to documents on disk. Because the database is regularly updated you want to write an export algorithm that create or recreate each document on disk if the document does not exist or if the corresponding record has been modified after the document was saved for the last time. Consequently, you need to compare the date and time of modification of a document (if it exists) with its corresponding record.

For illustrating this example, we use the table whose definition is shown below:



Documents	
Number	L
Subject	A
Theme	A
Description	T
Creation Stamp	L
Modification Stamp	L

Rather than saving both a date and time values into each record, you can save a "time stamp" value which expresses the number of seconds elapsed between an arbitrary anterior date and time (in this example we use Jan, 1st 1995 at 00:00:00) and the date and time when the record was saved.

In our example, the field [Documents]Creation Stamp holds the time stamp when the record was first created and the field [Documents]Modification Stamp holds the time stamp when the record was last modified.

The Time stamp project method listed below calculates the time stamp for a specific date and time or for the current date and time if no parameters are passed:

```
    ` Time stamp Project Method
    ` Time stamp { ( date ; Time ) } -> Long
    ` Time stamp { ( date ; Time ) } -> Number of seconds since Jan, 1st 1995

C_DATE($1;$vdDate)
C_TIME($2;$vhTime)
C_LONGINT($0)

If (Count parameters=0)
    $vdDate:=Current date
    $vhTime:=Current time
Else
    $vdDate:=$1
    $vhTime:=$2
End if
$0:=(($vdDate-!01/01/95!)*86400)+$vhTime
```

Note: Using this method, you can encode dates and times from the 01/01/95 at 00:00:00 to the 01/19/2063 at 03:14:07 which cover the long integer range 0 to 2^31 minus one.

Conversely, the Time stamp to date and Time stamp to time project methods listed below allow extracting the date and the time stored into a time stamp:

```
    ` Time stamp to date Project Method
    ` Time stamp to date ( Long ) -> Date
    ` Time stamp to date ( Time stamp ) -> Extracted date

C_DATE($0)
C_LONGINT($1)

$0:=!01/01/95!+($1\86400)

    ` Time stamp to time Project Method
    ` Time stamp to time ( Long ) -> Date
    ` Time stamp to time ( Time stamp ) -> Extracted time

C_TIME($0)
C_LONGINT($1)

$0:=Time(Time string(†00:00:00†+($1%86400)))
```

For insuring that the records have their time stamps correctly updated no matter the way they are created or modified, we just need to enforce that rule using the trigger of the table [Documents]:

```
    ` Trigger for table [Documents]

Case of
    : (Database event=Save New Record Event)
        [Documents]Creation Stamp:=Time stamp
        [Documents]Modification Stamp:=Time stamp
    : (Database event=Save Existing Record Event)
        [Documents]Modification Stamp:=Time stamp
End case
```

Once this is implemented in the database, we have all we need to write the project method CREATE DOCUMENTATION listed below. We use of GET DOCUMENT PROPERTIES and SET DOCUMENT PROPERTIES for handling the date and time of creation and modification of the documents.

```
    ` CREATE DOCUMENTATION Project Method

C_STRING(255;$vsPath;$vsDocPathName;$vsDocName)
C_LONGINT($vlDoc)
C_BOOLEAN($vbOnWindows;$vbDoIt;$vbLocked;$vbInvisible)
C_TIME($vhDocRef;$vhCreatedAt;$vhModifiedAt)
C_DATE($vdCreatedOn;$vdModifiedOn)

If (Application type=4D Client)
        ` If we are running 4D Client, save the documents
        ` locally on the Client machine where 4D Client is located
    $vsPath:=Long name to path name (Application file)
Else
        ` Otherwise, save the documents where the data file is located
    $vsPath:=Long name to path name (Data file)
End if
    ` Save the documents in a directory we arbitrarily name "Documentation"
$vsPath:=$vsPath+"Documentation"+Char(Directory symbol )
    ` If this directory does not exist, create it
If (Test path name($vsPath) # Is a directory)
    CREATE FOLDER($vsPath)
End if
    ` Establish the list of the already existing documents
    ` because we'll have to delete the obsolete ones, in other words,
    ` the documents whose corresponding records have been deleted.
ARRAY STRING(255;$asDocument;0)
DOCUMENT LIST($vsPath;$asDocument)
    ` Select all the records from the [Documents] table
ALL RECORDS([Documents])
    ` For each record
$vlNbRecords:=Records in selection([Documents])
$vlNbDocs:=0
$vbOnWindows:=On Windows
For ($vlDoc;1;$vlNbRecords)
        ` Assume we will have to (re)create the document on disk
    $vbDoIt:=True
        ` Calculate the name and the path name of the document
    $vsDocName:="DOC"+String([Documents]Number;"00000")
    $vsDocPathName:=$vsPath+$vsDocName
        ` Does this document already exist?
    If (Test path name($vsDocPathName+".HTM")=Is a document)
            ` If so, remove the document from the list of the documents
            ` that may end up deleted
        $vlElem:=Find in array($asDocument;$vsDocName+".HTM")
        If ($vlElem>0)
            DELETE ELEMENT($asDocument;$vlElem)
        End if
            ` Was the document saved after the last time the record was modified?
        GET DOCUMENT PROPERTIES($vsDocPathName+".HTM";$vbLocked;
                  $vbInvisible;$vdCreatedOn;$vhCreatedAt;$vdModifiedOn;$vhModifiedAt)
        If (Time stamp ($vdModifiedOn;$vhModifiedAt)>=
                                                      [Documents]Modification Stamp)
                ` If so, we do not need to recreate the document
            $vbDoIt:=False
        End if
    Else
            ` The document does not exist, reset these two variables so
            ` we know we'll have to compute them before setting the final properties
            ` of the document
        $vdModifiedOn:=!00/00/00!
        $vhModifiedAt:=†00:00:00†
    End if
        ` Do we need to (re)create the document?
    If ($vbDoIt)
            ` If so, increment the number of updated documents
        $vlNbDocs:=$vlNbDocs+1
            ` Delete the document if it already exists
        DELETE DOCUMENT($vsDocPathName+".HTM")
            ` And create it again
        If ($vbOnWindows)
            $vhDocRef:=Create document($vsDocPathName;"HTM")
        Else
            $vhDocRef:=Create document($vsDocPathName+".HTM")
        End if
        If (OK=1)
                ` Here write the contents of the document
            CLOSE DOCUMENT($vhDocRef)
            If ($vdModifiedOn=!00/00/00!)
                    ` The document did not exist, set the modification date and time
                    ` to their right values
                $vdModifiedOn:=Current date
                $vhModifiedAt:=Current time
            End if
                ` Change the properties of the document so its date and time of creation
                ` are made equal to those of the corresponding record
            SET DOCUMENT PROPERTIES($vsDocPathName+".HTM";$vbLocked;
                        $vbInvisible;Time stamp to date ([Documents]Creation Stamp);
                                                Time stamp to time ([Documents]Creation
                                                   Stamp);$vdModifiedOn;$vhModifiedAt)
        End if
    End if
        ` Just to know what's going on
    SET WINDOW TITLE("Processing Document "+String($vlDoc)+
                                                     " of "+String($vlNbRecords))
```

```
      NEXT RECORD([Documents])
   End for
      ` Delete the obsolete documents, in other words
      ` those which are still in the array $asDocument
   For ($vlDoc;1;Size of array($asDocument))
      DELETE DOCUMENT($vsPath+$asDocument{$vlDoc})
      SET WINDOW TITLE("Deleting obsolete document: "+Char(34)+
                                                    $asDocument{$vlDoc}+Char(34))
   End for
      ` We're done
   ALERT("Number of documents processed: "+String($vlNbRecords)+Char(13)+
                            "Number of documents updated: "+String($vlNbDocs)+Char(13)+
                        "Number of documents deleted: "+String(Size of array($asDocument)))
```

See Also

Document creator, Document type, SET DOCUMENT PROPERTIES.

SET DOCUMENT PROPERTIES

version 6.0

SET DOCUMENT PROPERTIES (document; locked; invisible; created on; created at; modified on; modified at)

Parameter	Type		Description
document	String	→	Document name or Full document pathname
locked	Boolean	→	Locked (True) or Unlocked (False)
invisible	Number	→	Invisible (True) or Visible (False)
created on	Date	→	Creation date
created at	Time	→	Creation time
modified on	Date	→	Last modification date
modified at	Time	→	Last modification time

Description

The command SET DOCUMENT PROPERTIES changes the information about the document whose name or pathname you pass in document.

Before the call:
• Pass True in Locked to lock the document. A locked document cannot be open nor deleted. Pass False in Locked to unlock a document.
• Pass True in invisible to hide the document. Pass False in invisible to make the document visible in the desktop windows.
• Pass the document creation date and time in created on and created at.
• Pass the document last modification date and time in modified on and modified at.

The dates and times of creation and last modification are managed by the file manager of your system each time you create or access a document. Using this command, you can change those properties for special purpose. See example for the command GET DOCUMENT PROPERTIES.

See Also

GET DOCUMENT PROPERTIES, SET DOCUMENT CREATOR, SET DOCUMENT TYPE.

Get document size

version 6.0

Get document size (document{; *}) → Number

Parameter	Type		Description
document	DocRef | String	→	Document reference number or Document name
*		→	On MacOS only: - if omitted, size of data fork - if specified, size of resource fork
Function result	Number	←	Size (expressed in bytes) of the document

Description

The command Get document size returns the size, expressed in bytes, of a document.

If the document is open, you pass its document reference number in document.
If the document is not open, you pass its name or pathname in document.

On Macintosh, if you do not pass the optional * parameter, the size of the data fork is returned. If you do pass the * parameter, the size of the resource fork is returned.

See Also

Get document position, SET DOCUMENT POSITION, SET DOCUMENT SIZE.

SET DOCUMENT SIZE

version 6.0

SET DOCUMENT SIZE (document; size)

Parameter	Type		Description
document	DocRef | String	→	Document reference number or Document name
size	Number	→	New size expressed in bytes

Description

The command SET DOCUMENT SIZE sets the size of a document to the number of bytes you pass in size.

If the document is open, you pass its document reference number in document.
If the document is not open, you pass its name or pathname in document.

On Macintosh, the size of the document's data fork is changed.

See Also

Get document position, Get document size, SET DOCUMENT POSITION.

Get document position

version 6.0

Get document position (docRef) → Number

Parameter	Type		Description
docRef	DocRef	→	Document reference number
Function result	Number	←	File position (expressed in bytes) from the beginning of the file

Description

This command operates only on a document currently open whose document reference number you pass in docRef.

Get document position returns the position, starting from the beginning of the document, where the next read (RECEIVE PACKET) or write (SEND PACKET) will occur.

See Also

RECEIVE PACKET, SEND PACKET, SET DOCUMENT POSITION.

SET DOCUMENT POSITION

version 6.0

SET DOCUMENT POSITION (docRef; offset{; anchor})

Parameter	Type		Description
docRef	DocRef	→	Document reference number
offset	Number	→	File position (expressed in bytes)
anchor		→	1 = Relatively to the beginning of the file 2 = Relatively to the end of the file 3 = Relatively to current position

Description

This command operates only on a document currently open whose document reference number you pass in docRef.

SET DOCUMENT POSITION sets the position you pass in offset where the next read (RECEIVE PACKET) or write (SEND PACKET) will occur.

If you omit the optional anchor parameter, the position is relative to the beginning of the document. If you do specify the anchor parameter, you pass one of the values listed above.

Depending on the anchor you can pass positive or negative values in offset.

See Also

Get document position, RECEIVE PACKET, SEND PACKET.

47 System Environment

Screen height

Screen height {(*)} → Number

Parameter	Type		Description
*	Number	→	Windows: height of application window, or height of screen if * is specified Macintosh: height of main screen
Function result	Number	←	Height expressed in pixels

Description

On Windows, Screen height returns the height of 4D application window (MDI window). If you specify the optional * parameter, Screen height returns the height of the screen.

On Macintosh, Screen height returns the height of the main screen, the screen where the menu bar is located.

See Also

SCREEN COORDINATES, Screen width.

Screen width

version 3.5

Screen width {(*)} → Number

Parameter	Type		Description
*	Number	→	Windows: width of application window, or width of screen if * is specified Macintosh: width of main screen
Function result	Number	←	Width expressed in pixels

Description

On Windows, Screen width returns the width of 4D application window (MDI window). If you specify the optional * parameter, Screen width returns the width of the screen.

On Macintosh, Screen width returns the width of the main screen, the screen where the menu bar is located.

See Also

SCREEN COORDINATES, Screen height.

Count screens

version 6.0

Count screens → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of monitors

Description

The command Count screens returns the number of screen monitors connected to your machine.

Windows note: On Windows, Count screens usually returns 1.

See Also

Menu bar screen, SCREEN COORDINATES, SCREEN DEPTH, Screen height, Screen width.

SCREEN COORDINATES

version 6.0

SCREEN COORDINATES (left; top; right; bottom{; screen})

Parameter	Type		Description
left	Number	←	Global left coordinate of screen area
top	Number	←	Global top coordinate of screen area
right	Number	←	Global right coordinate of screen area
bottom	Number	←	Global bottom coordinate of screen area
screen	Number	→	Screen number, or main screen if omitted

Description

The command SCREEN COORDINATES returns in left, top, right, and bottom the global coordinates of the screen specified by screen.

On Windows

Usually, you will not pass the screen parameter.

On Macintosh

If you omit the screen parameter, the command returns the coordinates of the main screen, the screen where the menu bar is displayed.

See Also

Count screens, Menu bar screen, SCREEN DEPTH.

SCREEN DEPTH

System Environment

version 6.0

SCREEN DEPTH (depth; color{; screen})

Parameter	Type		Description
depth	Number	←	Depth of the screen (number of colors = 2 ^ depth)
color	Number	←	1 = Color screen, 0 = Black and white or Gray scale
screen	Number	→	Screen number, or main screen if omitted

Description

The command Screen depth returns in depth and color information about the monitor.

After the call:

• The depth of the screen is returned in depth. The depth of the screen is the exponent of the power of 2 expressing the number of colors displayed on your monitor. For example, if your monitor is set for 256 colors (2^8), the depth of your screen is 8.

The following predefined constants are provided by 4th Dimension:

Constant	Type	Value
Black and white	Long Integer	0
Four colors	Long Integer	2
Sixteen colors	Long Integer	4
Two fifty six colors	Long Integer	8
Thousands of colors	Long Integer	16
Millions of colors 24 bit	Long Integer	24
Millions of colors 32 bit	Long Integer	32

If the monitor is set to display in color, 1 is returned in color. If the monitor is set to display in gray scale, 0 is returned in color. Note that this value is significant on the Macintosh platform.

The following predefined constants are provided by 4th Dimension:

Constant	Type	Value
Is gray scale	Long Integer	0
Is color	Long Integer	1

• The optional parameter screen specifies the monitor for which you want to get information. On Windows, you will not usually pass the screen parameter. On Macintosh, if you omit the screen parameter, the command returns the depth of the main screen, the screen where the menu bar is displayed.

Example

Your application displays many color graphics. Somewhere in your database, you could write:

```
⇒    SCREEN DEPTH ($vlDepth;$vlColor)
     If ($vlDepth<8)
        ALERT("The forms will look better if the monitor"+" was set to display 256 colors or
more.")
     End if
```

See Also

Count screens, SET SCREEN DEPTH.

SET SCREEN DEPTH

version 6.0

SET SCREEN DEPTH (depth; color{; screen})

Parameter	Type		Description
depth	Number	→	Depth of the screen (number of colors = 2 ^ Screen depth)
color	Number	→	1 = Color, 0 = Gray Scale
screen	Number	→	Screen number, or main screen if omitted

Description

This command does nothing on Windows.

On Macintosh, SET SCREEN DEPTH changes the depth and color/gray scale settings of the screen whose number you pass in screen. If you omit this parameter, the command is applied to the main screen.

For details about the values you pass in color and depth, see the description of the command SCREEN DEPTH.

See Also

SCREEN DEPTH.

Menu bar screen

System Environment

version 6.0

Menu bar screen → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Number of screen where menu bar is located

Description

Menu bar screen returns the number of the screen where the menu bar is located.

Windows note: On Windows, Menu bar screen usually returns 1.

See Also

Count screens, Menu bar height.

Menu bar height

version 6.0

Menu bar height → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Height (expressed in pixels) of menu bar (returns zero if menu bar is hidden)

Description

Menu bar height returns the height of the menu bar, expressed in pixels.

See Also

HIDE MENU BAR, Menu bar screen, SHOW MENU BAR.

FONT LIST

version 6.0

FONT LIST (fonts)

Parameter	Type		Description
fonts	Array	←	Array of font names

Description

The command FONT LIST populates the string or text array fonts with the names of the fonts available on your system.

Example

In a form, you want a drop-down list that displays a list of the fonts available on your system. The method of the drop-down list is as follows:

```
     Case of
        : (Form event=On Load)
           ARRAY STRING(63;asFont;0)
⇒          FONT LIST(asFont)
              ` ...

     End case
```

See Also

Font name, Font number.

Font name

version 6.0

Font name (fontNumber) → String

Parameter	Type		Description
fontNumber	Number	→	Font number for which to return the font name
Function result	String	←	Font name

Description

The command Font name returns the name of the font whose number is fontNumber. If there is no available font with that number, the command returns an empty string.

Examples

1. To display a form object with the default system font, you write:

```
⇒	FONT(myObject;Font name(0))  ` 0 is the font number of the default system font
```

2. To display a form object with the default application font, you write:

```
⇒	FONT(myObject;Font name(1))  ` 1 is the font number of the default application font
```

See Also

FONT LIST, Font number.

Font number

version 6.0

Font number (fontName) → Number

Parameter	Type		Description
fontName	String	→	Font name for which to return the font number
Function result	Number	←	Font number

Description

The command Font number returns the number of the font whose name is fontName. If there is no font with this name, the command returns 0.

Example

Some forms in your database use the font whose name is “Kind of Special.” Somewhere in your database, you could write:

```
⇒	If (Font number("Kind of Special")=0)
		ALERT("This form would look better if the font Kind of Special was installed.")
	End if
```

See Also

FONT LIST, Font name.

System folder

System folder → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Pathname to active system directory or folder

Description

The command System folder returns the pathname to the active Windows or Macintosh system folder.

See Also

ACI folder, Temporary folder.

Temporary folder

version 6.0

Temporary folder → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Pathname to temporary folder

Description

The command Temporary folder returns the pathname to the current temporary folder set by your system.

Example

See example for the command APPEND TO CLIPBOARD.

See Also

System folder.

Current machine

version 6.0

Current machine → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Network name of the machine

Description

The command Current machine returns the network name of your machine, as set in the Network Control Panel.

Example

Even if you are not running with the Client/Server version of the 4D environment, your application can include some network services that require your machine to be correctly configured. In the On Startup database method of your application, you write:

```
⇒	If ( (Current machine="") | (Current machine owner=""))
	     ` Display a dialog box asking the user to setup
	     ` the Network identity of his or her machine
	End if
```

See Also

Current machine owner.

Current machine owner

version 6.0

Current machine owner → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	Network name of machine owner

Description

The command Current machine owner returns the owner name of your machine, as set in the Network Control Panel.

Example

See example for the command Current machine.

See Also

Current machine.

Gestalt

version 6.0

Gestalt (selector; value) → Number

Parameter	Type		Description
selector	String	→	4-character gestalt selector
value	Number	←	Gestalt result
Function result	Number	←	Error code result

Description

The command Gestalt returns in value a numeric value that denotes the characteristics of your system hardware and software, depending on the selector you pass in selector.

If the requested information is obtained, Gestalt returns 0 in function result; otherwise, it returns the error -5550. If the selector is unkown, Gestalt returns the error -5551.

Important: The Gestalt Manager is part of MacOS. On Windows, some of the selectors are also implemented, but the usefulness of this command is limited.

For more information about the selectors that you can pass to Gestalt, refer to the Apple Developer documentation related to the Gestalt Manager.

Example

On Macintosh, using version 7.6 of MacOS, the following code displays the alert “You're running system version 0x0760”:

```
⇒	$vlErrCode:=Gestalt("sysv";$vlInfo)
	If ($vlErrCode=0)
		ALERT("You're running system version "+String($vlInfo;"&x"))
	End if
```

48 Table

DEFAULT TABLE

Table

version 3

DEFAULT TABLE (table)

Parameter	Type		Description
table	Table	→	Table to set as the default

Description

DEFAULT TABLE sets table as the default table for the current process.

There is no default table for a process until the DEFAULT TABLE command is executed. After a default table has been set, any command that omits the table parameter will operate on the default table. For example, consider this command:

```
INPUT FORM ([Table]; "form")
```

If the default table is first set to [Table], the same command could be written this way:

```
INPUT FORM ("form")
```

One reason for setting the default table is to create code that is not table specific. Doing this allows the same code to operate on different tables. You can also use pointers to tables to write code that is not table specific. For more information about this technique, see the description of the Table name command.

DEFAULT TABLE does not allow the omission of table names when referring to fields. For example:

```
[My Table]My Field:="A string"  ` Good
```

could not be written as:

```
DEFAULT TABLE ([My Table])
My Field:="A string"   ` WRONG
```

because a default table had been set. However, you can omit the table name when referring to fields in the table method, form, and objects that belong to the table.

In 4th Dimension, all tables are "open" and ready for use. DEFAULT TABLE does not open a table, set a current table, or prepare the table for input or output. DEFAULT TABLE is simply a programming convenience to reduce the amount of typing and make the code easier to read.

Tip: Although using DEFAULT TABLE and omitting the table name may make the code easier to read, many programmers find that using this command actually causes more problems and confusion than it is worth.

Example

The following example first shows code without the DEFAULT TABLE command. It then shows the same code, with DEFAULT TABLE. The code is a loop commonly used to add new records to a database. The INPUT FORM and ADD RECORD commands both require a table as the first parameter:

```
INPUT FORM ([Customers];"Add Recs")
Repeat
   ADD RECORD ([Customers])
Until (OK = 0)
```

Specifying the default table results in this code:

```
⇒    DEFAULT TABLE ([Customers])
     INPUT FORM ("Add Recs")
     Repeat
        ADD RECORD
     Until (OK = 0)
```

See Also

Current default table.

Current default table

version 3

Current default table → Pointer

Parameter	Type		Description
This command does not require any parameters			
Function result	Pointer	←	Pointer to the default table

Description

Current default table returns a pointer to the table that has been passed to the last call to DEFAULT TABLE for the current process.

Example

Provided a default table has been set, the following line of code sets the window title to the name of the current default table:

```
⇒	SET WINDOW TITLE(Table name(Current default table))
```

See Also

DEFAULT TABLE, Table, Table name.

INPUT FORM

Table

version 6.0 (Modified)

INPUT FORM ({table; }form{; *})

Parameter	Type		Description
table	Table	→	Table for which to set the input form, or Default table, if omitted
form	String	→	Name of the form to set as input form
*		→	Automatic window size

Description

The command INPUT FORM sets the current input form for table to form. The form must belong to table.

The scope of this command is the current process. Each table has its own input form in each process.

INPUT FORM does not display the form; it just designates which form is used for data entry, import, or operation by another command. For information about creating forms, see the *4th Dimension Design Reference*..

The default input form is defined in the Design environment Explorer window for each table. This default input form is used if the INPUT FORM command is not used to specify an input form, or if you specify a form that does not exist.

Input forms are displayed by a number of commands, which are generally used to allow the user to enter new data or modify old data. The following commands display an input form for data entry or query purposes:
• ADD RECORD
• DISPLAY RECORD
• MODIFY RECORD
• QUERY BY EXAMPLE

The DISPLAY SELECTION and MODIFY SELECTION commands display a list of records using the output form. The user can double-click on a record in the list, which displays the input form.

The import commands IMPORT TEXT, IMPORT SYLK and IMPORT DIF use the current input form for importing records.

The optional * parameter is used in conjunction with the form properties you set in the Design environment Form Properties window and the command Open window. Specifying the * parameter tells 4D to use the form properties to automatically resize the window for the next use of the form (as an input form or as a dialog box). See Open window for more information.

Note: Whether or not you pass the optional * parameter, INPUT FORM changes the input form for the table.

Example

The following example shows a typical use of INPUT FORM:

```
⇒	INPUT FORM ([Companies]; "New Comp")  ` Form for adding new companies
	ADD RECORD ([Companies])  ` Add a new company
```

See Also

ADD RECORD, DISPLAY RECORD, DISPLAY SELECTION, IMPORT DIF, IMPORT SYLK, IMPORT TEXT, MODIFY RECORD, MODIFY SELECTION, Open window, OUTPUT FORM, QUERY BY EXAMPLE.

OUTPUT FORM

Table

version 3

OUTPUT FORM ({table; }form)

Parameter	Type		Description
table	Table	→	Table for which to set the output form, or Default table, if omitted
form	String	→	Form name

Description

The command OUTPUT FORM sets the current output form for table to form. The form must belong to table.

The scope of this command is the current process. Each table has its own output form in each process.

OUTPUT FORM does not display the form; it just designates which form is printed, displayed, or used by another command. For information about creating forms, see the *4th Dimension Design Reference.*

The default output form is defined in the Design environment Explorer window for each table. This default output form is used if the OUTPUT FORM command is not used to specify an output form, or if you specify a form that does not exist.

Output forms are used by three groups of commands. One group displays a list of records on screen, another group generates reports, and the third group exports data. The DISPLAY SELECTION and MODIFY SELECTION commands display a list of records using an output form. You use the output form when creating reports with the PRINT LABEL and PRINT SELECTION commands. Each of the export commands (EXPORT DIF, EXPORT SYLK and EXPORT TEXT) also uses the output form.

Example

The following example shows a typical use of OUTPUT FORM. Note that although the OUTPUT FORM command appears immediately before the output form is used, this is not required. In fact, the command may be executed in a completely different method, as long as it is executed prior to this method:

```
   INPUT FORM ([Parts]; "Parts In")  ` Select the input form
⇒  OUTPUT FORM ([Parts]; "Parts List")  ` Select the output form
   MODIFY SELECTION ([Parts])  ` This command uses both forms
```

See Also

DISPLAY SELECTION, EXPORT DIF, EXPORT SYLK, EXPORT TEXT, INPUT FORM, MODIFY SELECTION, PRINT LABEL, PRINT SELECTION.

Current form table

version 6.0

Current form table → Pointer

Parameter	Type		Description
This command does not require any parameters			
Function result	Pointer	←	Pointer to the table of the currently displayed form

Description

The command Current form table returns the pointer to the table of the form being displayed or printed in the current process.

If there is no form being displayed or printed in the current process, the command returns Nil.

If there are several windows open for the current process (which means that the window opened last is the current active window), the command returns the pointer to the table of the form displayed in the active window.

If the currently displayed form is the Detail form for a subform area, you are in data entry and you double-clicked on a record or a subrecord of a double-clickable subform area. In this case, the command returns:
• The pointer to the table shown in the subform area, if the subform displays a table.
• A non-significant pointer, if the subform area displays a subtable.

Example

Throughout your application, you use the following convention when displaying a record:
If the variable vsCurrentRecord is present in a form, it displays “New Record” if you are working with a new record. If you are working with the 56th record of a selection composed of 5200 records, it displays “56 of 5200”.

To do so, use the object method to create the variable vsCurrentRecord, then copy and paste it into all of your forms:

```
          ` vsCurrentRecord non enterable variable object method
      Case of
         : (Form event =On Load)
            C_STRING (31;vsCurrentRecord)
            C_POINTER ($vpParentTable)
            C_LONGINT ($vlRecordNum)
⇒           $vpParentTable:=Current form table
            $vlRecordNum:=Record number ($vpParentTable->)
            Case of
               : ($vlRecordNum=-3)
                  vsCurrentRecord:="New Record"
               : ($vlRecordNum=-1)
                  vsCurrentRecord:="No Record"
               : ($vlRecordNum>=0)
                  vsCurrentRecord:=String (Selected record number ($vpParentTable->))+
                                          " of "+String (Records in selection ($vpParentTable->))
            End case
      End case
```

See Also

DIALOG, INPUT FORM, OUTPUT FORM, PRINT SELECTION.

49 Transactions

Using Transactions

Transactions are a series of related data modifications made to a database within a process. A transaction is not saved to a database permanently until the transaction is validated. If a transaction is not completed, either because it is canceled or because of some outside event, the modifications are not saved.

During a transaction, all changes made to the database data within a process are stored locally in a temporary buffer. If the transaction is accepted with VALIDATE TRANSACTION, the changes are saved permanently. If the transaction is canceled with CANCEL TRANSACTION, the changes are not saved.

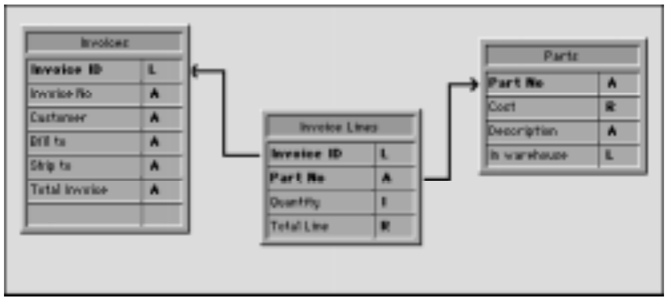
Since transactions deal with temporary record addresses, after a transaction is validated or canceled, the selection for each table of the current process becomes empty. For this reason, you should be cautious when using named selections inside a transaction. After a transaction is validated or canceled, a named selection created before or during the transaction may contain incorrect record addresses. For example, a named selection may contain the address of a deleted record or the temporary address of a record added during the transaction. This warning also applies to sets, because they are based on bit tables with record addresses.

The following commands use record numbers—do not use them in a transaction:
- GOTO RECORD
- RELATE ONE SELECTION
- RELATE MANY SELECTION

Transaction Examples

In this example, the database is a simple invoicing system. The invoice lines are stored in a table called [Invoice Lines], which is related to the table [Invoices] by means of a relation between the fields [Invoices]Invoice ID and [Invoice Lines]Invoice ID. When an invoice is added, a unique ID is calculated, using the Sequence number command. The relation between [Invoices] and [Invoice Lines] is an automatic Relate Many relation. The Auto Assign Related Value check box is checked.

The relation between [Invoice Lines] and [Parts] is manual.



When a user enters an invoice, the following actions are executed:
• Add a record in the table [Invoices].
• Add several records in the table [Invoice Lines].
• Update the [Parts]In Warehouse field of each part listed in the invoice.

This example is a typical situation in which you need to use a transaction. You must be sure that you can save all these records during the operation or that you will be able to cancel the transaction if a record cannot be added or updated. In other words, you must save related data.

If you do not use a transaction, you cannot guarantee the logical data integrity of your database. For example, if one record of the [Parts] records is locked, you will not be able to update the quantity stored in the field [Parts]In Warehouse. Therefore, this field will become logically incorrect. The sum of the parts sold and the parts remaining in the warehouse will not be equal to the original quantity entered in the record. You can avoid such a situation by using transactions.

There are several ways of performing data entry using transactions:

1. You can let 4D handle the transactions for you by selecting **Automatic Transactions during Data Entry** in the Design environment Database Properties dialog box. In this case, 4D starts the transaction and then validates or cancels it depending on whether or not you accepted the data entry. A data entry operation with a form containing a related table in a subform requires a transaction. This option applies to the whole database.

If you want to handle the transactions yourself, you need to use the transaction commands START TRANSACTION, VALIDATE TRANSACTION, and CANCEL TRANSACTION.

2. You can write:

```
READ WRITE([Invoice Lines])
READ WRITE([Parts])
INPUT FORM([Invoices];"Input")
Repeat
   START TRANSACTION
   ADD RECORD([Invoices])
   If (OK=1)
      VALIDATE TRANSACTION
   Else
      CANCEL TRANSACTION
   End if
Until (OK=0)
READ ONLY(*)
```

3. To reduce record locking while performing the data entry, you can also choose to manage transactions from within the form method and access the tables in READ WRITE only when it becomes necessary.

You perform the data entry using the input form for [Invoices], which contains the related table [Invoice Lines] in a subform. The form has two buttons: bCancel and bOK, both of which are no action buttons.

The adding loop becomes:

```
READ WRITE([Invoice Lines])
READ ONLY([Parts])
INPUT FORM([Invoices];"Input")
Repeat
   ADD RECORD([Invoices])
Until (bOK=0)
READ ONLY([Invoice Lines])
```

Note that the [Parts] table is now in read-only access mode during data entry. Read/write access will be available only if the data entry is validated.

The transaction is started in the [Invoices] input form method listed here:

```
Case of
   : (Form Event=On Load)
      START TRANSACTION
      [Invoices]Invoice ID:=Sequence number([Invoices]Invoice ID)
   Else
      [Invoices]Total Invoice:=Sum([Invoice Lines]Total line)
End case
```

If you click the bCancel button, the data entry as well as the transaction must be canceled. Here is the object method of the bCancel button:

```
Case of
   : (Form Event=On Clicked)
      CANCEL TRANSACTION
      CANCEL
End case
```

If you click the bValidate button, the data entry must be accepted and the transaction must be validated. Here is the object method of the bOK button:

```
Case of
   : (Form Event=On Clicked)
      $NbLines:=Records in selection([Invoice Lines])
      READ WRITE([Parts])   ` Switch to Read/Write access for the [Parts] table
      FIRST RECORD([Invoice Lines])   ` Start at the first line
      $ValidTrans:=True  ` Assume everything will be OK
      For ($Line;1;$NbLines)   ` For each line
         RELATE ONE([Invoice Lines]Part No)
         OK:=1  ` Assume you want to continue
            ` Try getting the record in Read/Write access
         While (Locked([Parts]) & (OK=1))
            CONFIRM("The Part "+[Invoice Lines]Part No+" is in use. Wait?")
            If (OK=1)
               DELAY PROCESS(Current process;60)
               LOAD RECORD([Parts])
            End if
         End while
         If (OK=1)
               ` Update quantity in the warehouse
            [Parts]In Warehouse:=[Parts]In Warehouse-[Invoice Lines]Quantity
            SAVE RECORD([Parts])   ` Save the record
         Else
            $Line:=$NbLines+1   ` Leave the loop
            $ValidTrans:=False
         End if
         NEXT RECORD([Invoice Lines])   ` Go next line
      End for
      READ ONLY([Parts])   ` Set the table state to read only
      If ($ValidTrans)
         SAVE RECORD([Invoices])   ` Save the Invoices record
         VALIDATE TRANSACTION   ` Validate all database modifications
      Else
         CANCEL TRANSACTION   ` Cancel everything
      End if
      CANCEL   ` Leave the form
End case
```

In this code, we call the CANCEL command regardless of the button clicked. The new record is not validated by a call to ACCEPT, but by the SAVE RECORD command. In addition, note that SAVE RECORD is called just before the VALIDATE TRANSACTION command. Therefore, saving the [Invoices] record is actually a part of the transaction. Calling the ACCEPT command would also validate the record, but in this case the transaction would be validated before the [Invoices] record was saved. In other words, the record would be saved outside the transaction.

Depending on your needs, you can let 4D handle transactions during data entry or you can customize your database, as shown in these examples. In the last example, the handling of locked records in the [Parts] table could be developed further.

See Also

CANCEL TRANSACTION, In transaction, START TRANSACTION, VALIDATE TRANSACTION.

START TRANSACTION

START TRANSACTION

Parameter	Type	Description

This command does not require any parameters

Description

START TRANSACTION starts a transaction in the current process. All changes to the database are stored temporarily until the transaction is accepted (validated) or canceled.

If you have several global processes, you can have several transactions. You cannot, however, nest transactions. If you start a transaction inside another transaction, 4th Dimension ignores the second transaction.

See Also

CANCEL TRANSACTION, In transaction, Using Transactions, VALIDATE TRANSACTION.

VALIDATE TRANSACTION

version 3

VALIDATE TRANSACTION

Parameter	Type	Description
This command does not require any parameters		

Description

VALIDATE TRANSACTION accepts the transaction in the current process that was started with START TRANSACTION. VALIDATE TRANSACTION saves the changes to the database that occurred during the transaction.

See Also

CANCEL TRANSACTION, In transaction, START TRANSACTION, Using Transactions.

CANCEL TRANSACTION

CANCEL TRANSACTION

Parameter	Type	Description
This command does not require any parameters		

Description

CANCEL TRANSACTION cancels the transaction in the current process that was started with START TRANSACTION. CANCEL TRANSACTION leaves the database unchanged by canceling the operations executed during the transaction.

See Also

In transaction, START TRANSACTION, Using Transactions, VALIDATE TRANSACTION.

In transaction

In transaction → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	Returns TRUE if current process is in transaction

Description

The command In transaction returns TRUE if the current process is in a transaction, otherwise it returns FALSE.

Example

If you perform a multi-record operation (adding, modifying, or deleting records), you may encounter locked records. In this case, if you have to maintain data integrity, you must be in transaction so you can "roll-back" the whole operation and leave the database untouched.

If you perform the operation from within a trigger or from a subroutine (that can be called while in transaction or not), you can use In transaction to check whether or not the current process method or the caller method started a transaction. If a transaction was not started, you do not even start the operation, because you already know that you will not be able to roll it back if it fails.

See Also

CANCEL TRANSACTION, START TRANSACTION, Triggers, VALIDATE TRANSACTION.

50 Triggers

Triggers

Triggers

version 6.0

A Trigger is a method attached to a table. It is a property of a table. You do not call triggers; they are automatically invoked by the 4D database engine each you manipulate table records (add, delete, modify, and load). You can write very simple triggers, and then make them more sophisticated.

Triggers can prevent “illegal” operations on the records of your database. They are a very powerful tool to restrict operations on a table, as well as to prevent accidental data loss or tampering. For example, in an invoicing system, you can prevent anyone from adding an invoice without specifying the customer to whom the invoice is billed.

Compatibility with Previous Versions of 4D

Triggers are a new type of method introduced in version 6. In previous versions of 4th Dimension, table methods (called file procedures) were executed by 4D only when a form for a table was used for data entry, display, or printing—they were rarely used. Note that triggers execute at a much lower level that the old file procedures. No matter what you do to a record via user actions (such as data entry) or programmatically (such as a call to SAVE RECORD), the trigger of a table will be invoked by 4D. Triggers are truly quite different from the old file procedures. If you have converted a version 3 database to version 6, and you want to take advantage of the new Trigger capability, you must deselect the **Use Old File Procedures Scheme** property in the **Database Properties** dialog box shown here.

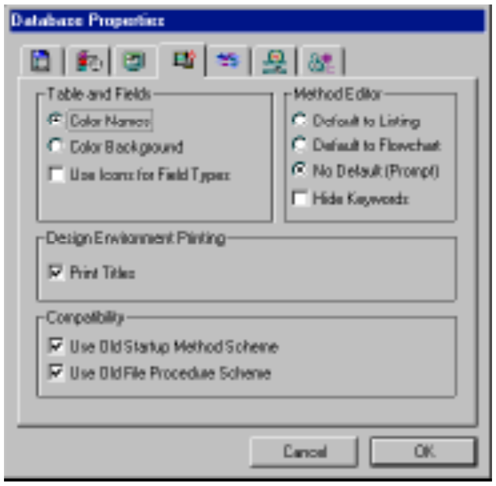

Database Properties

Table and Fields
- [✓ Color Names]
- Color Background
- Use Icons for Field Types

Method Editor
- Default to Listing
- Default to Flowchart
- [✓ No Default (Prompt)]
- Hide Keywords

Design Environment Printing
- [✓ Print Titles]

Compatibility
- [✓ Use Old Startup Method Scheme]
- [✓ Use Old File Procedure Scheme]

Activating and creating a Trigger

By default, when you create a table in the Design Environment, it has no trigger.

To use a trigger for a table, you need to:
• Activate the trigger and tell 4D when it has to be invoked.
• Write the code for the trigger.

Activating a trigger that is not yet written or writing a trigger without activating it will not affect the operations performed on a table.

1. Activating a Trigger

To activate a trigger for a table, you must select one of the **Triggers** options (database events) for the table in the **Table Properties** window:

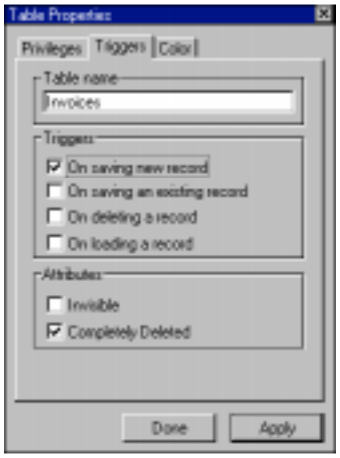


Table name: Invoices

Triggers
- [✓] On saving new record
- [ ] On saving an existing record
- [ ] On deleting a record
- [ ] On loading a record

Attributes
- [ ] Invisible
- [✓] Completely Deleted

On saving new record

If this option is selected, the trigger will be invoked each time a record is added to the table.
This happens when:
• Adding a record in data entry (User environment or ADD RECORD command)
• Creating and saving a record with CREATE RECORD and SAVE RECORD. Note that the trigger is invoked at the moment you call SAVE RECORD, not when it is created.
• Importing records (User environment or using an import command)
• Calling any other commands that create and/or save new records (i.e., ARRAY TO SELECTION, SAVE RELATED ONE, etc.)
• Using a Plug-in that calls the CREATE RECORD and SAVE RECORD commands

On saving an existing record
If this option is selected, the trigger will be invoked each time a record of the table is modified.
This happens when:
• Modifying a record in data entry (User environment or MODIFY RECORD command)
• Saving an already exiting record SAVE RECORD
• Calling any other commands that save existing records (i.e., ARRAY TO SELECTION, APPLY TO SELECTION,MODIFY SELECTION, etc.)
• Using a Plug-in that calls the SAVE RECORD command

On deleting a record
If this option is selected, the trigger will be invoked each time a record of the table is deleted.
This happens when:
• Deleting a record (User environment or calling DELETE RECORD or DELETE SELECTION)
• Performing any operation that provokes deletion of related records through the deletion control options of a relation
• Using a Plug-in that calls the DELETE RECORD command

On loading a record
If this option is selected, the trigger will be invoked each time a record of the table is loaded. This includes all situations in which a current record is loaded from the data file. You will this option less often than the three previous ones. **Note**: This option does not cover the creation of a new record; it only applies to the loading of existing records.

IMPORTANT: If you execute an operation or call a command that acts on multiple records, the trigger is called once for each record. For example, if you call APPLY TO SELECTION for a table whose current selection is composed of 100 records, the trigger will be invoked 100 times.

2. Creating a Trigger

To create a trigger for a table, use the **Explorer** Window or press Alt (on Windows) or Option (Macintosh) and double-click on the table title in the Structure window. For more information, see the *4th Dimension Design Reference* manual.

Database Events

A trigger can be invoked for one of the four **database events** described above. Within the trigger, you detect which event is occurring by calling the function Database event. This function returns a numeric value that denotes the database event.

Typically, you write a trigger with a Case of structure on the result returned by Database event:

```
        ` Trigger for [anyTable]
    C_LONGINT($0)
    $0:=0   ` Assume the database request will be granted
    Case of
⇒       : (Database event=Save New Record Event)
            ` Perform appropriates action for the saving of a newly created record
⇒       : (Database event=Save Existing Record Event)
            ` Perform appropriates actions for the saving of an already existing record
⇒       : (Database event=Delete Record Event)
            ` Perform appropriates actions for the deletion of a record
⇒       : (Database event=Load Record Event)
            ` Perform appropriates actions for the loading into memory of a record
    End case
```

Triggers are Functions

A trigger has two purposes:
• Performing actions on the record before it saved, deleted or after it is just being loaded.
• Granting or rejecting a database operation.

1. Performing Actions

Each time a record is saved (added or modified) to a [Documents] table , you want to "mark" the record with a time stamp for creation and another one for the most recent modification. You can write the following trigger:

```
        ` Trigger for table [Documents]
    Case of
        : (Database event=Save New Record Event)
            [Documents]Creation Stamp:=Time stamp
            [Documents]Modification Stamp:=Time stamp
        : (Database event=Save Existing Record Event)
            [Documents]Modification Stamp:=Time stamp
    End case
```

Note: The function Time stamp used in this example is a small project method that returns the number of seconds elapsed since a fixed date was chosen arbitrarily.

After this trigger has been written and activated, no matter what way you add or modify a record to the [Documents] table (data entry, import, project method, 4D plug-in), the fields [Documents]Creation Stamp and [Documents]Modification Stamp will automatically be assigned by the trigger before the record is eventually written to the disk.

Note: See the example for the command GET DOCUMENT PROPERTIES for a complete study of this example.

2. Granting or rejecting the database operation

To grant or reject a database operation, the trigger must return a **trigger error code** in the $0 function result.

Example

Let's take the case of an [Employees] table. During data entry, you enforce a rule on the field [Employees]Social Security Number. When you click the validation button, you check the field using the object method of the button:

```
   ` bAccept button object method
If (Good SS number ([Employees]SS number))
   ACCEPT
Else
   BEEP
   ALERT ("Enter a Social Number then click OK again.")
End if
```

If the field value is valid, you accept the data entry; if the field value is not valid, you display an alert and you stay in data entry.

If you also create [Employees] records programmatically, the following piece of code would be programmatically valid, but would violate the rule expressed in the previous object method:

```
   ` Extract from a projec method
   ` ...
CREATE RECORD ([Employees])
[Employees]Name :="DOE"
SAVE RECORD ([Employees]) ` ← DB rule violation! The SS number has not been
assigned!
          ` ...
```

Using a trigger for the table [Employees], you can enforce the [Employees]SS number rule at all the levels of the database. The trigger would look like:

```
    ` Trigger for [Employees]
$0:=0
$dbEvent:=Get database event
Case of
    : (($dbEvent=Save New Record Event) | ($dbEvent=Save Existing Record Event))
    If (Not(Good SS number ([Employees]SS number)))
        $0:=-15050
    Else
        ` ...
    End if
        ` ...
End case
```

Once this trigger is written and activated, the line SAVE RECORD ([Employees]) will generate a database engine error -15050, and the record will NOT be saved.

Similarily, a 4D Plug-in would attempt to save an [Employees] record with an invalid social security number. The trigger would generate the same error and the record would not be saved.

The trigger guarantees that nobody (user, database designer, Plug-in, 4D Open client with 4D Server) can violate the social security number rule, either deliberately or accidentally.

Note that even if you do not have a trigger for a table, you can get database engine errors while attempting to save or delete a record. For example, if you attempt to save a record with a duplicated value in a unique indexed field, you the error -9998 is returned.

Therefore, triggers returning errors add new database engine errors to your application:
• 4D manages the “regular” errors: unique index, relational data control, and so on.
• Using triggers, you manage the custom errors unique to your application.

Important: You can return an error code value of your choice. However, do NOT use error codes already taken by the 4D database engine. We strongly recommend that you use error codes between -32000 and -15000. We reserve error codes above -15000 for the database engine.

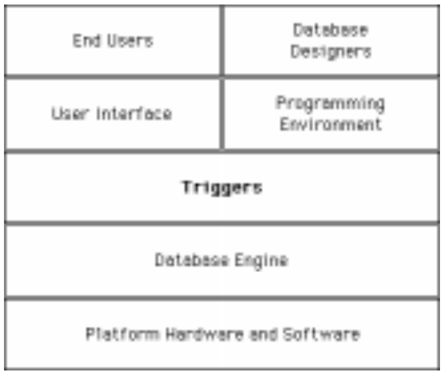
At the process level, you handle trigger errors the same way you handle database engine errors:
• You can let 4D display the standard error dialog box, then the method is halted.
• You can use an error-handling method installed using ON ERR CALL and recover the error the appropriate way.

Note: During data entry, if a trigger error is returned while attempting to validate or delete a record, the error is handled like a unique indexed error. The error dialog is displayed, and you stay in the data entry. Even though you only use a database in the User environment (not in Custom menus), you have the benefit of using triggers.

Although a trigger returns no error ($0:=0), this does not mean that a database operation will be successful; a unique index violation may occur. If the operation is the update of a record, the record may be locked, an I/O error may occur, and so on. The checking is done after the execution of the trigger. However, at the higher level of the executing process, errors returned by the database engine or a trigger are the same—a trigger error is a database engine error.

Triggers and the 4D architecture

Triggers execute at the database engine level. This is summarized in the following diagram:



Triggers are executed on the machine where the database engine is actually located. This is obvious with a 4D single-user version. On 4D Server, triggers are executed within the acting process on the server machine, not on the client machine.

When a trigger is invoked, it executes within the context of the process that attempts the database operation. This process, which invokes the trigger execution, is called the **invoking process**.

In particular, the trigger works with the current selections, current records, table read/write states, and record locking operations of the invoking process.

Warning: A trigger cannot and must not change the current record of the table to which it is attached. Within a trigger, if you need to check a unique value on multiple fields, use the command SET QUERY DESTINATION, which allows you to query a table without changing the current selection or current record of the table.

Be careful about using other database or language objects of the 4D environment, because a trigger may execute on a machine different than that of the invoking process—this is the case with 4D Server!

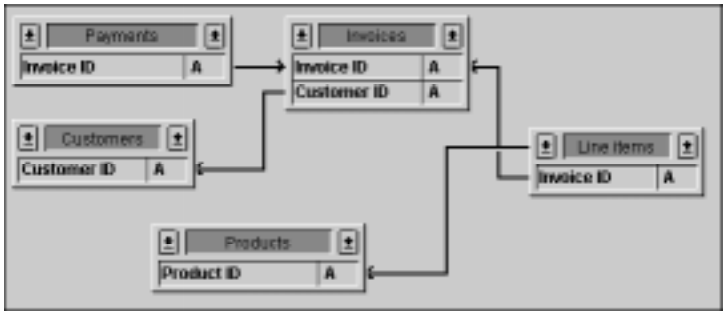
• **Interprocess variables**: A trigger has access to the interprocess variables of the machine where it executes. With 4D Server, it can access a machine different than that of the invoking process.
• **Process variables**: An independent process variables table is shared by all the triggers. A trigger has no access to the process variables of the invoking process.
• **Local variables**: You can use local variables in a trigger. Their scope is the trigger execution; they are created/deleted at each execution.
• **Semaphores**: A trigger can test or set global semaphores as well as local semaphores (on the machine where it executes). However, a trigger must execute quickly, so you must be very careful when testing or setting semaphores from within triggers.
• **Sets and Named selections**: If you use a set or a named selection from within a trigger, you work on the machine where the triggers executes.
• **User Interface**: Do NOT use user interface elements in a trigger (no alerts, no messages, no dialog boxes). Accordingly, you should limit any tracing of triggers in the Debugger window. Remember that in Client/Server, triggers execute on the 4D Server machine. An alert message on the server machine does not help a user on a client. Let the invoking process handle the user interface.

Trigger and Transactions

Transactions must be handled at the invoking process level. Yo ushould not manage transactions at the trigger level. During one trigger execution, if you have to add, modify or delete multiple records (see case study below), you must first use the In transaction command from within the trigger to test if the invoking process is currently in transaction. If this is not the case, the trigger may potentially encounter a locked record. Therefore, if the invoking process is not in transaction, do not even start the operations on the records. Just return an error in $0 in order to signal the invoking process that the database operation it tries to perform must be executed in transaction. Otherwise, if locked records are met, the invoking process will have no means to roll back the actions of the trigger.

Cascading Triggers

Given the following example structure:



Note: The tables have been collapsed; they have more fields than shown here.

Let's say that the database "authorizes" the deletion of an invoice. We can examine how such an operation would be handled at the trigger level (because you could also perform deletions at the process level).

In order to maintain the relational integrity of the data, deleting an invoice requires the following actions to be performed in the trigger for [Invoices]:
• In the [Customer] record, decrement the Gross Sales field by the amount of the invoice.
• Delete all the [Line Items] records related to the invoice.
• This also implies that the [Line Items] trigger decrements the Quantity Sold field of the [Products] record related to the line item to be deleted.
• Delete all the [Payments] records related to the deleted invoice.

First, the trigger for [Invoices] must perform these actions only if the invoking process is in transaction, so a roll-back is possible if a locked record is met.

Second, the trigger for [Line Items] is **cascading** with the trigger for [Invoices]. The [Line Items] trigger executes "within" the execution of the [Invoices] trigger, because the deletion of the list items are consequent to a call to DELETE SELECTION from within the [Invoices] trigger.

Consider that all tables in this example have triggers activated for all database events. The cascade of triggers will be:

- [Invoices] trigger is invoked because the invoking process delete an invoice
  - [Customers] trigger is invoked because the [Invoices] trigger updates the Gross Sales field
  - [Line Items] trigger is invoked because the [Invoices] trigger deletes a line item (repeated)
    - [Products] trigger is invoked because the [Line Items] trigger updates the Quantity Sold field
  - [Payments] trigger is invoked because the [Invoices] trigger deletes a payment (repeated)

In this cascade relationship, the [Invoices] trigger is said to be executing at level 1, the [Customers], [Line Items], and [Payments] triggers at level 2, and the [Products] trigger at level 3.

From within the triggers, you can use the command Trigger level to detect at the level at which a trigger is executed. In addition, you get can command TRIGGER PROPERTIES to get information about the other levels.

For example, if a [Products] record is being deleted at a process level, the [Products] trigger would be executed at level 1, not at level 3.

Using Trigger level and TRIGGER PROPERTIES, you can detect the cause of an action. In our example, an invoice is deleted at a process level. If we delete a [Customers] record at a process level, then the [Customers] trigger should attempt to delete all the invoices related to that customer. This means that the [Invoices] trigger will be invoked as above, but for another reason. From within the [Invoices] trigger, you can detect if it executed at level 1 or 2. If it did execute at level 2, you can then check whether or not it is because the [Customers] record is deleted. If this the case, you do not even need to bother updating the Gross Sales field.

Using Sequence Numbers within a Trigger

While handling an On saving new record database event, you can call the Sequence number command to maintain a unique ID number for the records of a table.

Example

```
   ` Trigger for table [Invoices]
Case of
   : (Database event=On Saving new record)
         ` ...
      [Invoices]Invoice ID Number:=Sequence number ([Invoices])
         ` ...
End case
```

However, while you are in transaction, you must surround this call by a test on the field in order to avoid incrementing the sequence number incorrectly. A typical example is entering records in a subform during a data entry transaction.

Example

```
   ` Trigger for table [Invoice Line Items]
Case of
   : (Database event=On Saving new record)
         ` ...
      If ([Invoice Line Items]Line Items ID Number=0)
         [Invoice Line Items]Line Items ID Number:=Sequence number ([Invoices])
      End if
         ` ...
End case
```

See Also

Database event, Methods, Record number, Trigger level, TRIGGER PROPERTIES.

Database event

Database event → Number

Parameter	Type		Description	
This command does not require any parameters				
Function result	Number	←	0	Outside any trigger execution cycle
			1	Saving a new record
			2	Saving an existing record
			3	Deleting a record
			4	Loading a record

Description

Called from within a trigger, the command Database event returns a numeric value that denotes the type of the database event, in other words, the reason why the trigger has been invoked.

The following predefined constants are provided:

Constant	Type	Value
Save New Record Event	Long Integer	1
Save Existing Record Event	Long Integer	2
Delete Record Event	Long Integer	3
Load Record Event	Long Integer	4

If the database event is Save Existing Record Event, you can use the command Modified to detect if a particular field of the record has been modified. In this case, you can also retrieve the value of the field as it is currently stored on disk, using the command Old.

Note: These two commands are significant only when applied to simple fields such as Alphanumeric or Real. The results of these commands are not significant with Picture, BLOB, or subtable fields.

Within the trigger, if you perform database operations on multiple records, you may encounter conditions (usually locked records) that will make the trigger unable to correctly perform what it is supposed to do. (An example of this situation is updating multiple records in a [Products] table when a record is being added to an [Invoices] table.) At this point, you must stop attempting database operations, and return a database error so the invoking process will know that its database request cannot be performed. Then, the invoking process must be able to cancel, during the transaction, the incomplete database operations performed by the trigger. When this type of situation occurs, you need to know from within the trigger if you are in transaction even before attempting anything. To do so, you use the command In transaction.

While cascading trigger calls, 4th Dimension has no limit other than the available memory. To optimize trigger execution, you may want to write the code of your triggers depending not only on the database event, but also on the level of the call when triggers are cascaded. For example, during a deletion database event for the [Invoices] table, you may want to skip the update of the [Customers] Gross Sales field if the deletion of the [Invoices] record is part of the deletion of **all** the invoices related to a [Customers] record being deleted. To do so, use the commands Trigger level and TRIGGER PROPERTIES.

Example

You use the command Database event to structure your triggers as follows:

```
          ` Trigger for [anyTable]
       C_LONGINT($0)
       $0:=0   ` Assume the database request will be granted
       Case of
⇒         : (Database event=Save New Record Event)
             ` Perform appropriates action for the saving of a newly created record
⇒         : (Database event=Save Existing Record Event)
             ` Perform appropriates actions for the saving of an already existing record
⇒         : (Database event=Delete Record Event)
             ` Perform appropriates actions for the deletion of a record
⇒         : (Database event=Load Record Event)
             ` Perform appropriates actions for the loading into memory of a record
       End case
```

See Also

In transaction, Modified, Old, Trigger level, TRIGGER PROPERTIES, Triggers.

Trigger level

Trigger level → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Level of trigger execution (0 if outside any trigger execution cycle)

Description

The command Trigger level returns the execution level of the trigger.

For more information on execution levels, see the topic Cascading Triggers in the section Triggers.

See Also

Database event, TRIGGER PROPERTIES, Triggers.

TRIGGER PROPERTIES

version 6.0

TRIGGER PROPERTIES (triggerLevel; dbEvent; tableNum; recordNum)

Parameter	Type		Description
triggerLevel	Number	→	Trigger execution cycle level
dbEvent	Number	←	Database event
tableNum	Number	←	Involved table number
recordNum	Number	←	Involved record number

Description

The command TRIGGER PROPERTIES returns information about the trigger execution level you pass in triggerLevel. You use TRIGGER PROPERTIES in conjunction with Trigger level to perform different actions depending on the cascading of trigger execution levels. For more information, see the topic Cascading Triggers in the section Triggers.

If you pass a non-existing trigger execution level, the command returns 0 (zero) in all parameters.

The nature of the database event for the trigger execution level is returned in dbEvent. The following predefined constants are provided:

Constant	Type	Value
Save New Record Event	Long Integer	1
Save Existing Record Event	Long Integer	2
Delete Record Event	Long Integer	3
Load Record Event	Long Integer	4

The table number and record number for the record involved by the database event for the trigger execution level are returned in tableNum and recordNum.

Note: Remember that while in transaction, newly created records have temporary record numbers.

See Also

About Record Numbers, Database event, Trigger level, Triggers.

51 User Interface

BEEP

Parameter	**Type**	**Description**

This command does not require any parameters

Description

The command BEEP causes the PC or Macintosh to generate a beep. Your computer (on Windows or Macintosh) can emit a sound other than a beep, depending on the sound chosen in the Sound control panel.

Warning: Do not call BEEP from within a Web connection process, because the beep will be produced on the 4th Dimension Web server machine and not on the client Web browser machine.

Example

In the following example, if no records are found by the query, a beep is emitted and an alert is displayed:

```
     QUERY([Customers];[Customers]Name=$vsNameToLookFor)
     If (Records in selection([Customers])=0)
⇒        BEEP
         ALERT("There is no Customer with such a name.")
     End if
```

See Also

PLAY.

PLAY

version 3

PLAY (objectName{; channel})

Parameter	Type		Description
objectName	String	→	Sound name Windows: .WAV, .MID or .AVI file any platform: → MacOS-based ‘snd’ resource or empty string for stopping asynchronous play
channel	Number	→	if specified, synthesizer channel and asynchronous if omitted, synchronous

Description on Windows

On Windows, the command PLAY plays sound (.WAV files), MIDI (.MID files), or Video (.AVI files) Windows files. You pass the full pathname of the file you want to play in objectName.

Note: You cannot play multimedia files or objects in asynchronous mode. To do so, use OLE Services.

On Macintosh or on Windows (with some restrictions, see Important Note below), the command PLAY plays the sound resource named by objectName on Macintosh.

The channel parameter specifies the Macintosh synthesizer channel. If channel is not specified, the channel is for simple digitized sounds and is synchronous. Synchronous means that all processing stops until the sound has finished. If channel is 0, the channel is for simple digitized sounds and is asynchronous. Asynchronous means that processing does not stop and the sound plays in the background.

To stop playing a synchronous sound, use the following statement:

```
⇒	PLAY ("";0)
```

If you work with a database on Macintosh and Windows concurrently, you can also play Macintosh sounds on the Windows platform. To do so:

• On the Macintosh, using a resource editor such as ResEdit or Resorcerer, copy the required 'snd ' resources into the resource fork of the structure file.
• Transport the database from Macintosh to Windows, using 4D Transporter.

Important Note: The Windows version of 4th Dimension does not play Macintosh sounds that have been compressed by MACE. If your Macintosh 'snd' resource does not play on Windows, determine whether it complies with the following requirements:

snd resource field	**Value (in hexadecimal)**
Version	0x0001
NbSynth	0x0001
SynthResID	0x0005
SynthInitOptions	0x000000A0
NbSoundCommand	0x0001
FirstCommand	0x8051

You can check the internal data of a 'snd' resource using Resorcerer.

Examples

1. The following example shows how to play a video file on Windows:

```
        $DocRef := Open document ( ""; "AVI")
        If (OK=1)
            CLOSE DOCUMENT($DocRef)
⇒           PLAY (Document)
        End if
```

2. The following example code appears in a startup method. It welcomes the user with a sound called Welcome Sound on Macintosh:

```
⇒       PLAY ("Welcome Sound")  ` Play the Welcome Sound
```

See Also

BEEP.

Get platform interface

User Interface

version 6.0

Get platform interface → Number

Parameter	Type		Description
This command does not require any parameters			
Function result	Number	←	Current platform interface in use

Description

The command Get platform interface returns a numeric value that denotes the current platform interface used for displaying forms.

The function can return one of the following values:

Constant	Type	Value
Automatic interface	Long Integer	-1
Macintosh interface	Long Integer	0
Windows 3.1 interface	Long Integer	1
Windows 95 interface	Long Integer	2
Copland interface	Long Integer	3

You can change the platform interface using the command SET PLATFORM INTERFACE or within the Design environment **Database Properties** dialog box.

See Also

GET PLATFORM INTERFACE.

SET PLATFORM INTERFACE

version 3.5

SET PLATFORM INTERFACE (interface)

Parameter	Type		Description
interface	Number	→	New platform interface setting: -1 Automatic 0 MacOS (System7) 1 Windows 3.1 2 Windows 95 3 Copland

Description

The command SET PLATFORM INTERFACE sets the platform interface used for displaying the forms.

You pass in interface one of the following predefined constants:

Constant	Type	Value
Automatic interface	Long Integer	-1
Macintosh interface	Long Integer	0
Windows 3.1 interface	Long Integer	1
Windows 95 interface	Long Integer	2
Copland interface	Long Integer	3

The command does nothing if the value you pass does not change the current platform interface.

Note: The platform interface can also be changed in the Design environment **Database Properties** dialog box.

Example

In a 4D Client/Server architecture, the Macintosh and Windows stations can use different platform interfaces concurrently. To do so, you can call the SET PLATFORM INTERFACE command in the On Startup database method:

```
      ` This example assumes that user preferences are stored in a [Preferences] table
      ` Look for the record corresponding to the current user
   QUERY([Preferences];[Preferences]User name=Current User)
   If (Records in selection([Preferences])=0)
         ` If not found, look for the default preferences
      QUERY([Preferences];[Preferences]User name="Default")
   End if
      ` Set the Platform Interface according to the user preferences
   SET PLATFORM INTERFACE ([Preferences]Platform Interface)
```

See Also

Get platform interface.

SET TABLE TITLES

version 6.0

SET TABLE TITLES (tableTitles; tableNumbers)

Parameter	Type		Description
tableTitles	String Array	→	Table names as they must appear in dialog boxes
tableNumbers	Numeric Array	→	Actual table numbers

Description

SET TABLE TITLES enables you to mask, rename, and reorder the tables of your database when they appear in standard 4th Dimension dialog boxes such as the Query editor, within the User or Custom Menus environments.

The arrays tableTitles and tableNumbers must be synchronized. In the array tableTitles, you pass the names of the tables as you would like them to appear. If you do not want to show a particular table, do not include its name or new title in the array. The tables will appear in the order you specify in this array. In each element of the array tableNumbers, you pass the actual table number corresponding to the table name or new title passed in the same element number in the array tableTitles.

For example, you have a database composed of the tables A, B, and C, created in that order. You want these tables to appear as X, Y, and Z. In addition you do not want to show table B. Finally, you want to show Z and X, in that order. To do so, you pass Z and X in a two-element tableTitles array, and you pass 3 and 1 in a two-element tableNumbers array.

SET TABLE TITLES does NOT change the actual structure of your database. It only affects posterior uses of the standard 4th Dimension dialog boxes, such as the Query Editor, within the User or Custom menus environments. The scope of the command SET TABLE TITLES is the worksession. One benefit in Client/Server, is that several 4D Client stations can simultaneously “see” your database in different ways. You can call SET TABLE TITLES as many times as you want. Note, however, that it affects only the next appearances of the standard 4th Dimension dialog boxes.

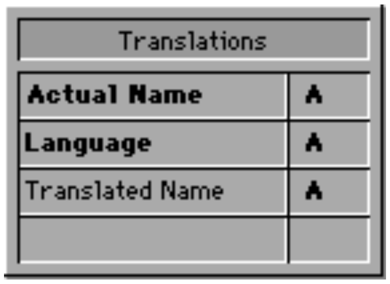
Use the command SET TABLE TITLES for:
• Dynamically localizing a database.
• Showing tables the way you want, independent from the actual definition of your database.
• Showing tables in a way that depends on the identity or custom privileges of a user.

WARNING: SET TABLE TITLES does NOT override the Invisible property of a table. If a table is set to be invisible at the Design level of your database, though it is included in a call to SET TABLE TITLES, it will not appear.

Example

• You are building a 4D application that you plan to sell internationally. Therefore, you must carefully consider localization issues. Regarding the standard 4th Dimension dialog boxes that can appear in the User and Custom Menus environments, you can address localization needs by using a [Translations] table and a few project methods to create and use fields localized for any number of countries.

• In your database, add the following table:



Translations	
Actual Name	A
Language	A
Translated Name	A

• Then, create the TRANSLATE TABLES AND FIELDS project method listed below. This method browses the actual structure of your database and creates all the necessary [Translations] records for the localization corresponding to the language passed as parameter.

```
     ` TRANSLATE TABLES AND FIELDS project method
     ` TRANSLATE TABLES AND FIELDS ( String )
     ` TRANSLATE TABLES AND FIELDS ( Language )

   C_STRING(31;$1)
   C_LONGINT($vlTable;$vlField)

   For ($vlTable;1;Count tables) ` Loop through the tables
         ` Check if there is a translation of the table name for the specified language
      QUERY([Translations];[Translations]Actual Name=Table name($vlTable);*)
      QUERY([Translations]; & ;[Translations]Language=$1)
      If (Records in selection([Translations])=0)
            ` If not, create the record
         CREATE RECORD([Translations])
         [Translations]Actual Name:=Table name($vlTable)
         [Translations]Language:=$1
            ` The translated table name will have to be entered
         SAVE RECORD([Translations])
      End if
```

```
    For ($vlField;1;Count fields($vlTable))
            ` Check if there is a translation of the field name for the specified language
        QUERY([Translations];[Translations]Actual Name=Field name($vlTable;$vlField);*)
        QUERY([Translations]; & ;[Translations]Language=$1)
        If (Records in selection([Translations])=0)
                ` If not, create the record
            CREATE RECORD([Translations])
            [Translations]Actual Name:=Field name($vlTable;$vlField)
            [Translations]Language:=$1
                ` The translated field name will have to be entered
            SAVE RECORD([Translations])
        End if
    End for
End for
```

• At this point, if you execute the following line, you create as many records as needed for a Spanish localization of the tables and fields titles.

TRANSLATE TABLES AND FIELDS ("Spanish")

• After this call has been executed, you can then enter the [Translations]Translated Name for each of the newly created records.

• Finally, each time you want to show your database's standard 4D dialog boxes using the Spanish localization, you execute the following line:

LOCALIZED TABLES AND FIELDS ("Spanish")

with the project method LOCALIZED TABLES AND FIELDS:

```
    ` LOCALIZED TABLES AND FIELDS global method
    ` LOCALIZED TABLES AND FIELDS ( String )
    ` LOCALIZED TABLES AND FIELDS ( Language )

C_STRING(63;$1)
C_LONGINT($vlTable;$vlNbTable;$vlField;$vlNbField)

$vlNbTable:=Count tables  ` Get the number of tables present in the database
    ` Initialize the arrays to be passed to SET TABLE TITLES
ARRAY STRING(31;$asTableName;$vlNbTable)
ARRAY INTEGER($aiTableNumber;$vlNbTable)
For ($vlTable;1;$vlNbTable)  ` Loop through the tables
    $asTableName{$vlTable}:=Table name($vlTable)  ` Get the name of the table
    $aiTableNumber{$vlTable}:=$vlTable  ` Store the actual table number
        ` Look for the translation
    QUERY([Translations];[Translations]Actual Name=$asTableName{$vlTable};*)
    QUERY([Translations]; & ;[Translations]Language=$1)
```

```
        If (Records in table([Translations])>0)
                ` If available, use the localized table name
            $asTableName{$vlTable}:=[Translations]Translated Name
        End if
        $vlNbField:=Count fields($vlTable)  ` Get the number of fields for that table
            ` Initialize the arrays to be passed to SET FIELD TITLES
        ARRAY STRING(31;$asFieldName;$vlNbTable)
        ARRAY INTEGER($aiFieldNumber;$vlNbTable)
        For ($vlField;1)  ` Loop through the fields
            $asFieldName{$vlField}:=Field name($vlTable;$vlField)  ` Get the name of the field
            $aiFieldNumber{$vlField}:=$vlField  ` Store the actual field number
            QUERY([Translations];[Translations]Actual Name=$asFieldName{$vlField};*)
                ` Look for the translation
            QUERY([Translations]; & ;[Translations]Language=$1)
            If (Records in table([Translations])>0)
                    ` If available, use the localized field name
                $asFieldName{$vlField}:=[Translations]Translated Name
            End if
        End for
        SORT ARRAY($asFieldName;$aiFieldNumber;>)
        SET FIELD TITLES(Table($vlTable)->;$asFieldName;$aiFieldNumber)
    End for
    SORT ARRAY($asTableName;$aiTableNumber;>)
⇒   SET TABLE TITLES($asTableName;$aiTableNumber)
```

• Note that new localizations can be added to the database without modifying or recompiling the code.

See Also

Count tables, SET FIELD TITLES, Table name.

SET FIELD TITLES

User Interface

version 6.0

SET FIELD TITLES (table | subtable; fieldTitles; fieldNumbers)

Parameter	Type		Description
table | subtable	Table or Subtable	→	Table or Subtable for which to set the field titles
fieldTitles	String Array	→	Field names as they must appear in dialog boxes
fieldNumbers	Numeric Array	→	Actual field numbers

Description

SET FIELD TITLES enables you to mask, rename, and reorder the fields of the table or subtable passed in table or subtable when they appear in standard 4th Dimension dialog boxes, such as the Query editor, within the User or Custom Menus environments.

The arrays fieldTitles and fieldNumbers must be synchronized. In the array fieldTitles, you pass the name of the fields as you would like them to appear. If you do not want to show a particular field, do not include its name or new title into the array. The fields will appear in the order you specify in this array. In each element of the array fieldNumbers, you pass the actual field number corresponding to the field name or new title passed in the same element number in the array fieldTitles.

For example, you have a table or subtable composed of the fields F, G, and H, created in that order. You want these fields to appear as M, N, and O. In addition you do not want to show field N. Finally, you want to show O and M in that order. To do so, pass O and M in a two-element fieldTitles array and pass 3 and 1 in a two-element fieldNumbers array.

SET FIELD TITLES does NOT change the actual structure of your table. It only affects posterior uses of the standard 4th Dimension dialog boxes, such as the Query editor, within the User or Custom menus environments. The scope of the command SET FIELD TITLES is the worksession. One benefit in Client/Server, is that several 4D Client stations can simultaneously “see” your table in different manners. You can call SET FIELD TITLES as many times as you want. Note, however, that it affects only the next appearances of the standard 4th Dimension dialog boxes.

Use the command SET FIELD TITLES for:
• Dynamically localizing a table.
• Showing fields the way you want, independent of the actual definition of your table.
• Showing fields in a way that depends on the identity or custom privileges of a user.

WARNING: SET FIELD TITLES does NOT override the Invisible property of a field. If a field is set to be invisible at the Design level of your database, even though it is included in a call to SET FIELD TITLES, it will not appear.

Example

See example for the command SET TABLE TITLES.

See Also

Count fields, Field name, SET TABLE TITLES.

Shift down

version 6.0

Shift down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Shift key

Description

Shift down returns TRUE if the Shift key is pressed.

Example

The following object method for the button bAnyButton performs different actions, depending on which modifier keys are pressed when the button is clicked:

```
      ` bAnyButton Object Method
   Case of
            ` Other multiple key combinations could be tested here
            ` ...
      : (Shift down & Windows Ctrl down)
            ` Shift and Windows Ctrl (or Macintosh Command) keys are pressed
         DO ACTION1
            ` ...
      : (Shift down)
            ` Only Shift key is pressed
         DO ACTION2
            ` ...
      : (Windows Ctrl down)
            ` Only Windows Ctrl (or Macintosh Command) key is pressed
         DO ACTION3
            ` ...
            ` Other individual keys could be tested here
            ` ...
   End case
```

See Also

Caps lock down, Macintosh command down, Macintosh control down, Macintosh option down, Windows Alt down, Windows Ctrl down.

Caps lock down

Caps lock down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Caps Lock key

Description

Caps lock down returns TRUE if the Caps Lock key is pressed.

Example

See example for the command Shift down.

See Also

Macintosh command down, Macintosh control down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

Windows Ctrl down

version 6.0

Windows Ctrl down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Windows Ctrl key (Command key on Macintosh)

Description

Windows Ctrl down returns TRUE if the Windows Ctrl key is pressed.

Note: When called on a Macintosh platform, Windows Ctrl down returns TRUE if the Macintosh Command key is pressed.

Example

See example for the command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

Windows Alt down

User Interface

version 6.0

Windows Alt down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Windows Alt key (Option key on Macintosh)

Description

Windows Alt down returns TRUE if the Windows Alt key is pressed.

Note: When called on a Macintosh platform, Windows Alt down returns TRUE if the Macintosh Option key is pressed.

Example

See example for the command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh control down, Macintosh option down, Shift down, Windows Ctrl down.

Macintosh command down

Macintosh command down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Macintosh Command key (Ctrl key on Windows)

Description

Macintosh command down returns TRUE if the Macintosh command key is pressed.

Note: When called on a Windows platform, Macintosh command down returns TRUE if the Windows Ctrl key is pressed.

Example

See example for the command Shift down.

See Also

Caps lock down, Macintosh control down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

Macintosh option down

User Interface

version 6.0

Macintosh option down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Macintosh Option key (Alt key on Windows)

Description

Macintosh option down returns TRUE if the Macintosh Option key is pressed.

Note: When called on a Windows platform, Macintosh option down returns TRUE if the Windows Alt key is pressed.

Example

See example for the command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh control down, Shift down, Windows Alt down, Windows Ctrl down.

Macintosh control down

version 6.0

Macintosh control down → Boolean

Parameter	Type		Description
This command does not require any parameters			
Function result	Boolean	←	State of the Macintosh Control key

Description

Macintosh control down returns TRUE if the Macintosh Control key is pressed.

Note: When called on a Windows platform, Macintosh control down always return FALSE. This Macintosh key has no equivalent on Windows.

Example

See example for the command Shift down.

See Also

Caps lock down, Macintosh command down, Macintosh option down, Shift down, Windows Alt down, Windows Ctrl down.

GET MOUSE

User Interface

version 6.0

GET MOUSE (mouseX; mouseY; mouseButton{; *})

Parameter	Type		Description
mouseX	Number	←	Horizontal coordinate of mouse
mouseY	Number	←	Vertical coordinate of mouse
mouseButton	Number	←	Mouse button state: 0 = Button up 1 = Button down 2 = Right button down (Windows only) 3 = Both buttons down (Windows only)
*		→	if specified, local coordinate system is used if omitted, global coordinate system is used

Description

The command GET MOUSE returns the current state of the mouse.

The horizonal and vertical coordinates are returned in mouseX and mouseY. If you you omit the * parameter, they are expressed relative to the screen. If you pass the * parameter, they are expressed relative to the frontmost window of the current process.

The parameter mouseButton returns the state of the buttons, as listed previously.

Example

See the example for the command Pop up menu.

See Also

Caps lock down, Macintosh command down, Macintosh control down, Macintosh option down, ON EVENT CALL, Shift down, Windows Alt down, Windows Ctrl down.

Pop up menu

User Interface

version 6.0

Pop up menu (contents{; default}) → Number

Parameter	Type		Description
contents	Text	→	Text menu definition
default	Number	→	Default selected menu item number
Function result	Number	←	Selected menu item number

Description

The command Pop up menu displays a pop-up menu at the current location of the mouse.

In other to follow user interface rules, you usually call this command in response to a mouse click and if the mouse button is still down.

You define the items of the pop-up menu with the parameter contents as follows:
• Separate each item from the next one with a semi-colon (;). For example, "ItemText1;ItemText2;ItemText3".
• To disable an item, place an opening parenthesis (() in the item text.
• To specify a separation line, pass "(-" as item text.
• To specify a font style for a line, place in the item text a less than sign (<) followed by one of these characters:

<B	Bold
<I	Italic
<U	Underline
<O	Outline (Macintosh only)
<S	Shadow (Macintosh only)

• To add a check mark to an item, place in the item text a question mark (?) followed by the character you want as a check mark. On Macintosh, the character is displayed; on Windows, a check mark is displayed, no matter what character you passed.
• To add an icon to an item, place in the item text a circumflex accent (^) followed by a character whose ASCII code minus 48 is the resource ID of a MacOS-based icon resource.
• To add a shortcut to an item, place in the item text a slash (/) followed by the shortcut character for the item. Note that this last option is purely informative; no shortcut will activate the pop-up menu. However, you may want to include a shortcut if the pop-up menu item has an equivalent in the main menu bar of your application.

The optional default parameter allows you to specify the default menu item selected when the pop-up menu is displayed. Pass a value between 1 and the number of menu items. If you omit this parameter, the command selects the first menu item as the default.

If you select a menu item, the command returns its number; otherwise, it returns zero (0).

Note: Use pop-up menus that have a reasonable number of items. If you want to display more than 50 items, you might think about a using scrollable area in a form instead of a pop-up menu.

Example

The project method MY SPEED MENU pulls down a navigation speed menu:

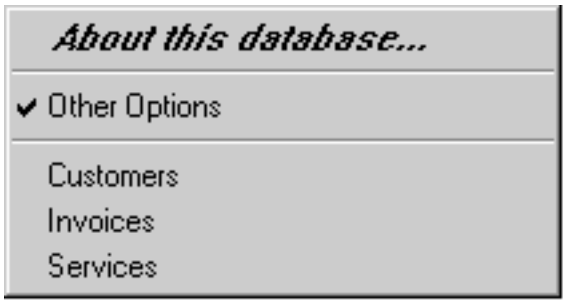
```
         ` MY SPEED MENU project method
     GET MOUSE($vlMouseX;$vlMouseY;$vlButton)
     If (Macintosh control down | ($vlButton=2))
         $vtItems:="About this database...<I;(-;!-Other Options;(-"
         For ($vlTable;1;Count tables)
             $vtItems:=$vtItems+";"+Table name($vlTable)
         End for
⇒        $vlUserChoice:=Pop up menu($vtItems)
         Case of
             : ($vlUserChoice=1)
                 ` Display Information
             : ($vlUserChoice=2)
                 ` Display options
             Else
                 If ($vlUserChoice>0)
                     ` Go to table whose number is $vlUserChoice-4
                 End if
         End case
     End if
```

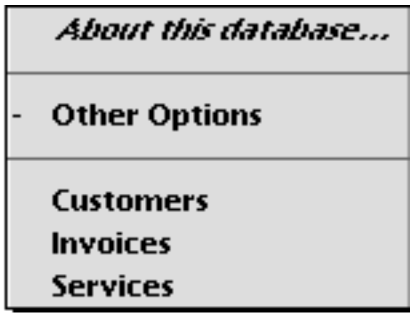
This project method can be called from:
• The method of a form object that reacts to a mouse click without waiting for the mouse button to be released (i.e., an invisible button)
• A process that “spies” events and communicate with the other processes
• An event-handling method installed using ON EVENT CALL

In the last two cases, the click does not need to occur in any form object. This is one of the advantages of the Pop up menu command. Generally, you use form objects to display pop-up menus. Using Pop up menu, you can display the menu anywhere.

The pop-up menu is displayed on Windows by pressing the right mouse button; it is displayed on Macintosh by pressing Control-Click. Note, however, that the method does not actually check whether or not there was a mouse click; the caller method tests that.

The following is the pop-up menu as it appears on Windows (left) and Macintosh (right). Note the standard check mark for the Windows version.





See Also

GET MOUSE.

version 6.0

POST KEY

POST KEY (code{; modifiers{; process}})

Parameter	Type		Description
code	Number	→	ASCII code of character or function key code
modifiers	Number	→	State of modifier keys
process	Number	→	Destination process reference number, or Application event queue, if omitted, or 0

Description

The command POST KEY simulates a keystroke. Its effects is like if the user actually entered a character on the keyboard.

You pass the ASCII code of the character in code.

If you pass the parameter modifiers, you pass one or a combination of the Events (modifiers) constants. For example, to simulate the Shift key, pass Shif key bit. If you do not pass the modifiers parameter, no modifiers are simulated.

If you specify the process parameter, the keystroke is sent to the process whose process number you pass in process. If you pass 0 (zero) or if you omit the parameter, the keystroke is sent at the application level, and the 4D scheduler will dispatch it to the appropriate process.

Example

See example for the command Process number.

See Also

POST CLICK, POST EVENT.

POST CLICK

version 6.0

POST CLICK (mouseX; mouseY{; process}{; *})

Parameter	Type		Description
mouseX	Number	→	Horizontal coordinate
mouseY	Number	→	Vertical coordinate
process	Number	→	Destination process reference number, or Application event queue, if omitted, or 0
*		→	if specified, local coordinate system is used if omitted, global coordinate system is used

Description

The command POST CLICK simulates a mouse click. Its effect as if the user actually clicked the mouse button.

You pass the horizontal and vertical coordinates of the click in mouseX and mouseY. If you pass the * parameter, you express these coordinates relative to the frontmost window of the process whose process number you pass in process. If you omit the * parameter, you express these coordinates relative to the screen.

If you specify the process parameter, the click is sent to the process whose process number you pass in process. If you pass 0 (zero) or if you omit the parameter, the click is sent at the application level, and the 4D scheduler will dispatch it to the appropriate process.

See Also

POST EVENT, POST KEY.

POST EVENT

version 6.0

POST EVENT (what; message; when; mouseX; mouseY; modifiers{; process})

Parameter	Type		Description
what	Number	→	Type of event
message	Number	→	Event message
when	Number	→	Event time expressed in ticks
mouseX	Number	→	Horizontal coordinate of mouse
mouseY	Number	→	Vertical coordinate of mouse
modifiers	Number	→	Modifier keys state
process	Number	→	Destination process reference number, or Application event queue, if omitted, or 0

Description

The command POST EVENT simulates a keyboard or mouse event. Its effect is as if the user actually acted on the keyboard or the mouse.

You pass one of the following values in what:

Constant	Type	Value
Mouse down event	Long Integer	1
Mouse up event	Long Integer	2
Key down event	Long Integer	3
Key up event	Long Integer	4
Auto key event	Long Integer	5

If the event is a mouse-related event, you pass 0 (zero) in message. If the event is a keyboard-related event, you pass the ASCII code of the simulated character in message.

Usually, you pass the value returned by Tickcount in when.

If the event is a mouse-related event, you pass the horizontal and vertical coordinates of the click in mouseX and mouseY. If you omit the * parameter, you express these coordinates relative to the screen. If pass the * parameter, you express these coordinates relative to the frontmost window of the process whose process number you pass in process.

In the parameter modifiers, you pass one or a combination of the Events (modifiers) constants. For example, to simulate the Shift key, pass Shift key bit.

If you specify the process parameter, the event is sent to the process whose process number you pass in process. If you pass 0 (zero) or if you omit the parameter, the event is sent at the application level, and the 4D scheduler will dispatch it to the appropriate process.

See Also

POST CLICK, POST KEY.

GET HIGHLIGHT

version 3

GET HIGHLIGHT (area; startSel; endSel)

Parameter	Type		Description
area	Field | Variable	→	Enterable field or variable
startSel	Number	←	Current text selection starting position
endSel	Number	←	Current text selection ending position

Description

The command GET HIGHLIGHT is used to determine what text is currently highlighted.

Warning: Although you pass a field or variable enterable area name to GET HIGHLIGHT, this command returns the significant selection position only when it is applied to the area currently being edited.

Note: This command cannot be used with fields in the List form of a subform.

Text can be highlighted by the user or by the HIGHLIGHT TEXT command.

The parameter startSel returns the position of the first highlighted character.
The parameter endSel returns the position of the last highlighted character plus one.

If startSel and endSel are returned equal, the insertion point is positioned before the character specified by startSel. The user has not selected any text, and no characters are highlighted.

Examples

1. The following example gets the highlighted selection from the field called [Products]Comments:

```
⇒   GET HIGHLIGHT ([Products]Comments;vFirst;vLast)
    If (vFirst<vLast)
       ALERT("The selected text is: "+Substring([Products]Comments;vFirst;vLast-vFirst))
    End if
```

2. See example for the command FILTER KEYSTROKE.

See Also

FILTER KEYSTROKE, HIGHLIGHT TEXT, Keystroke.

HIGHLIGHT TEXT

User Interface

version 3

HIGHLIGHT TEXT (area; startSel; endSel)

Parameter	Type		Description
area	Field | Variable	→	Enterable field or variable
startSel	Number	→	New text selection starting position
endSel	Number	→	New text selection ending position

Description

The command HIGHLIGHT TEXT highlights a section of the text in area.

If area is not the object currently being edited, the focus is then set to this area.

Note: This command cannot be used with fields in the List form of a subform.

startSel is the first character position to be highlighted, and lastSel is the last character plus one to be highlighted. If startSel and lastSel are equal, the insertion point is positioned before the character specified by startSel, and no characters are highlighted.

If lastSel is greater than the number of characters in area, then all characters between startSel and the end of the text are highlighted.

Example

1. The following example selects all the characters of the enterable field [Products]Comments:

```
⇒	HIGHLIGHT TEXT([Products]Comments;1;Length([Products]Comments)+1)
```

2. The following example moves the insertion point to the beginning of the enterable field [Products]Comments:

```
⇒	HIGHLIGHT TEXT([Products]Comments;1;1)
```

3. The following example moves the insertion point to the end of the enterable field [Products]Comments:

```
	$vLen:=Length([Products]Comments)+1
⇒	HIGHLIGHT TEXT([Products]Comments;$vLen;$vLen)
```

4. See example for the command FILTER KEYSTROKE.

See Also

GET HIGHLIGHT.

SET CURSOR

SET CURSOR {(cursor)}

Parameter	Type		Description
cursor	Number	→	MacOS-based cursor resource number

Description

The command SET CURSOR changes the mouse cursor to the cursor stored in the MacOS-based ‘CURS’ resource whose ID number you pass in cursor.

If you omit the parameter, the mouse cursor is set to the standard arrow.

Use the command RESOURCE LIST to get the list of available cursors.

See Also

RESOURCE LIST.

Last object

Last object → Pointer

Parameter	Type		Description
This command does not require any parameters			
Function result	Pointer	←	Pointer to the last or current enterable area

Description

Last object returns a pointer to the last or current enterable area, in other words, the object that the cursor is in or has just left. You can use Last object to perform an action on a form area without having to know which object is currently selected. Be sure to test that the object is the correct data type, using Type, before performing an action on it. This command cannot be used with fields in subforms.

Example

The following example is an object method for a button. The object method changes the data in the current object to uppercase. The object must be a text or string data type (type 0 or 24):

```
⇒     $vp := Last object  ` Save the pointer to the last area
          ` If it is a string or text area
      If ((Type ($vp->) = Is Alpha field) | (Type($vp->) = Is String var))
              ` Change the area to uppercase
         $vp-> := Uppercase ($vp->)
      End if
```

REDRAW

version 3

REDRAW (object)

Parameter	Type		Description
object	Object	→	Subtable for which to redraw the subform, or Table for which to redraw the subform, or Field for which to redraw the area, or Variable for which to redraw the area

Description

When you use a method to change the value of a field or subfield displayed in a subform, you must execute REDRAW to ensure that the form is updated.

INVERT BACKGROUND

version 3

INVERT BACKGROUND (textVar | textField)

Parameter	Type		Description
textVar | textField	Variable | Field	→	Text variable or field to invert

Description

INVERT BACKGROUND is used to invert textVar or textField in the form.

The scope of the command is the form being used.

You can use INVERT BACKGROUND when displaying on screen or printing to a dot matrix printer. A postscript printer will not print an inverted background.

You cannot invert a variable in an output form. Avoid using INVERT BACKGROUND on an enterable variable. Entering characters will only partially erase the inverted display.

Example

The following example is an object method for a variable in an input form. It tests the value of a field. If the field is positive, the object method does nothing. If the field is negative, the object method inverts the display of the variable in the form:

```
    vAmount:=[Accounts]Amount  ` Put the value of field in the variable
    If (vAmount < 0)  ` If it is a negative amount...
⇒       INVERT BACKGROUND (vAmount)  ` Invert the background
    End if
```

Note: This command, originally created for black and white user interfaces, is now rarely used. You now generally use colors to highlight a field or a variable.

See Also

SET COLOR, SET RGB COLOR.

52 Users and Groups

EDIT ACCESS

Parameter	Type	Description

This command does not require any parameters

Description

EDIT ACCESS allows the user to edit the password system. The Passwords window in the Design environment is used to edit the access.

Groups can be edited by the Designer, the Administrator and group owners. The Designer and the Administrator can edit any group. Group owners can edit only the groups they own. Users can be added to and removed from groups. The command has no effect if no groups are defined.

The Designer and the Administrator can add new users, as well as assign them to groups.

Example

The following example displays the Passwords window to the user:

```
⇒	EDIT ACCESS
```

See Also

CHANGE ACCESS, CHANGE PASSWORD.

CHANGE ACCESS

Parameter	**Type**	**Description**

This command does not require any parameters

Description

CHANGE ACCESS allows the user to change to a different access level without leaving the database. The same password dialog box that is displayed when the user launches the database is presented, and the user can gain access as a different user.

The dialog box displayed will depend on the Preferences set in the Database Properties dialog box in the Design Environment.

Example

The following example displays the password dialog box to the user:

```
⇒    CHANGE ACCESS
```

See Also

CHANGE PASSWORD.

CHANGE PASSWORD

CHANGE PASSWORD (password)

Parameter	Type		Description
password	String	→	New password

Description

CHANGE PASSWORD changes the password of the current user. This command replaces the current password with the new password you pass in password.

Warning: Password are case-sensitive.

Example

The following example allows the user to change his or her password.

```
CHANGE ACCESS   ` Present user with password dialog
If (OK=1)
   $pw1:=Request("Enter new password for "+Current user)
      ` The password should be at leat five characters long
   If (((OK=1) & ($pw1#"")) & (Length($pw1)>5))
         ` Make sure the password has been entered correctly
      $pw2:=Request("Enter password again")
      If ((OK=1) & ($pw1=$pw2))
         CHANGE PASSWORD($pw2)  ` Change the password
      End if
   End if
End if
```

See Also

CHANGE ACCESS.

Current user

Current user → String

Parameter	Type		Description
This command does not require any parameters			
Function result	String	←	User name of the current user

Description

Current user returns the user name of the current user.

Example

See example for the command User in group.

See Also

CHANGE ACCESS, CHANGE PASSWORD, User in group.

User in group

User in group (user; group) → Boolean

Parameter	Type		Description
user	String	→	User name
group	String	→	Group name
Function result	Boolean	←	TRUE = user is in group FALSE = user is not in group

Description

User in group returns TRUE if user is in group.

Example

The following example searches for specific invoices. If the current user is in the Executive group, he or she is allowed access to forms that display confidential information. If the user is not in the Executive group, a different form is displayed:

```
     QUERY([Invoices];[Invoices]Retail>100)
⇒    If (User in group(Current user;"Executive"))
        OUTPUT FORM([Invoices];"Executive Output")
        INPUT FORM([Invoices];"Executive Input")
     Else
        OUTPUT FORM([Invoices];"Standard Output")
        INPUT FORM([Invoices];"Standard Input")
     End if
     MODIFY SELECTION([Invoices];*)
```

See Also

Current user.

DELETE USER

version 6.0

DELETE USER (UserID)

Parameter	Type		Description
UserID	Number	→	ID number of user to delete

Description

The command DELETE USER deletes the user whose unique user ID number you pass in userID. You must pass a valid user ID number returned by the command GET USER LIST.

If the user account does not exist or has already been deleted, the error -9979 is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

Deleted user names no longer appear in the Password window displayed when the database is open or when you call CHANGE ACCESS. However, in order to maintain unique user ID numbers, the user account is kept in the password system. Deleted user names are displayed in green in the Design environment Passwords window.

See Also

GET USER LIST, GET USER PROPERTIES, Is user deleted, SET USER PROPERTIES.

Error Handling

If you do not have the proper access privileges for calling DELETE USER or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

Is user deleted

Is user deleted (userNumber) → Boolean

Parameter	Type		Description
userNumber	Number	→	User ID number
Function result	Boolean	←	TRUE = User account is deleted or does not exist FALSE = User account is active

Description

The command Is user deleted tests the user account whose unique user ID number you pass in userID.

If the user account does not exist or has been deleted, Is user deleted returns TRUE. Otherwise, it returns FALSE.

See Also

DELETE USER, GET USER PROPERTIES, SET USER PROPERTIES.

Error Handling

If you do not have the proper access privileges for calling Is user deleted or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

GET USER LIST

GET USER LIST (userNames; userNumbers)

Parameter	Type		Description
userNames	String Array	←	User names as they appear in the Password editor window
userNumbers	Numeric Array	←	Corresponding unique user ID numbers

Description

GET USER LIST populates the arrays userNames and userNumbers with the names and unique ID numbers of the users as they appear in the Passwords window.

The array userNames is filled with the user names displayed in the Passwords window, including users whose accounts are disabled (user names displayed in green in the Passwords window).

Note: Use the command Is user deleted to detect deleted users.

The array userNumbers, synchronized with userNames, is filled with the corresponding unique user ID numbers. These numbers can have the following values or ranges:

User ID number	User description
1	Designer user
2	Administrator user
3 to 15000	User created by the Designer of the database (user #3 is the first user created by the Designer, user #4 the second, and so on).
-11 to -15000	User created by the Administrator of the database (user #-11 is the first user created by the Designer, user #-12 is the second, and so on).

See Also

GET GROUP LIST, GET USER PROPERTIES, SET USER PROPERTIES.

Error Handling

If you do not have the proper access privileges for calling GET USER LIST or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

GET USER PROPERTIES

version 6.0

GET USER PROPERTIES (userID; name; startup; password; nbLogin; lastLogin{; memberships})

Parameter	Type		Description
userID	Number	→	Unique user ID number
name	String	←	Name of the user
startup	String	←	Startup method name
password	String	←	Crypted password
nbLogin	Number	←	Number of logins to the database
lastLogin	Date	←	Date of last login to the database
memberships	Numeric Array	←	ID numbers of groups to which the user belongs

Description

GET USER PROPERTIES returns the information about the user whose unique user ID number you pass in userID. You must pass a valid user ID number returned by the command GET USER LIST.

If the user account does not exist or has been deleted, the error -9979 is generated. You can catch this error with an error-handling method installed using ON ERR CALL. Otherwise, you can call Is user deleted to test the user account before calling GET USER PROPERTIES.

User ID numbers can have the following values or ranges:

User ID number	User description
1	Designer user
2	Administrator user
3 to 15000	User created by the Designer of the database (user #3 is the first user created by the Designer, user #4 the second, and so on).
-11 to -15000	User created by the Administrator of the database (user #-11 is the first user created by the Designer, user #-12 is the second, and so on).

After the call, you retrieve the name, startup method, encrypted password, number of logins and date of last login for the user, in the parameters name, startup, password, nbLogin and lastLogin.

If you pass the optional memberships parameter, the unique ID numbers of the groups to which the user belongs are returned. Group ID numbers can have the following ranges:

Group ID number	Group description
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

See Also

GET GROUP LIST, GET USER LIST, SET USER PROPERTIES.

Error Handling

If you do not have the proper access privileges for calling GET USER PROPERTIES or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

SET USER PROPERTIES

version 6.0

SET USER PROPERTIES (userID; name; startup; password; nbLogin; lastLogin{; memberships})

Parameter	Type		Description
userID	Number	→	Unique ID number of user account, or -1 for adding a user affiliated with the Designer, or -2 for adding a user affiliated with the Administrator
		←	Unique ID number of new user
name	String	→	New user name
startup	String	→	Name of new user startup method
password	String	→	New (unencrypted) password
nbLogin	Number	→	New number of logins to the database
lastLogin	Number	→	New date of last login to the database
memberships	Numeric Array	→	ID numbers of groups to which the user belongs

Description

SET USER PROPERTIES enables you to change and update the properties of an existing user account whose unique user ID number you pass in userID, or to add a new user affiliated with the Designer or the Administrator.

If you are changing the properties of an existing user account, you must pass a valid user ID number returned by the command GET USER LIST.

If the user account does not exist or has been deleted, the error -9979 is generated. You can catch this error with an error-handling method installed using ON ERR CALL. Otherwise, you can call Is user deleted to test the user account before calling SET USER PROPERTIES.

User ID numbers can have the following values or ranges:

User ID number	User description
1	Designer user
2	Administrator user
3 to 15000	User created by the Designer of the database (user #3 is the first user created by the Designer, user #4 the second, and so on).
-11 to -15000	User created by the Administrator of the database (user #-11 is the first user created by the Designer, user #-12 is the second, and so on).

To add a new user affiliated with the Designer pass -1 in userID. To add a new user affiliated with the Administrator pass -2 in userID. In both cases, when adding a new user, 4th Dimension tries to reuse the first available disabled user account; it creates a new account only if no disabled account is available. After the call, if the user is successfully added, its unique ID number is returned in userID.

If you do not pass -1, -2 or a valid user ID number, SET USER PROPERTIES does nothing.

Before the call, you pass the new name, startup method, password, number of logins and date of last login for the user, in the parameters name, startup, password, nbLogin and lastLogin. You pass an unencrypted password in the password parameter. 4D will encrypt it for you before saving it in the user account. If you do not want to change all the properties of the user (aside from the memberships, see below), first call GET USER PROPERTIES and pass the returned values for the properties you want to leave unchanged.

If you do not pass the optional memberships parameter, the current memberships of the user are left unchanged. If you do not pass memberships when adding a user, the user will not belong to any group.

If you pass the optional memberships parameter, you change all the memberships for the user. Before the call, you must populate the array memberships with the unique ID numbers of the groups to which the user will belong. Group ID numbers can have the following ranges:

Group ID number	**Group description**
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

To revoke all the memberships of a user, pass an empty memberships array.

See Also

DELETE USER, GET GROUP LIST, GET USER LIST, GET USER PROPERTIES, Is user deleted.

Error Handling

If you do not have the proper access privileges for calling SET USER PROPERTIES or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

GET GROUP LIST

GET GROUP LIST (groupNames; groupNumbers)

Parameter	Type		Description
groupNames	String Array	←	Names of the groups as they appear in the Password editor window
groupNumbers	Numeric Array	←	Corresponding unique group ID numbers

Description

GET GROUP LIST populates the arrays groupNames and groupNumbers with the names and unique ID numbers of the groups as they appear in the Password editor window.

The array groupNames is filled with the names of all groups that appear in the Password editor window, including the disabled groups (group names appearing in red in the Password editor window).

Note: Use the command GET GROUP PROPERTIES to detect groups for which entry is disabled.

The array groupNumbers, synchronized with groupNames, is filled with the corresponding unique group ID numbers. These numbers can have the following ranges:

Group ID number	Group description
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

See Also

GET GROUP PROPERTIES, GET USER LIST, SET GROUP PROPERTIES.

Error Handling

If you do not have the proper access privileges for calling GET GROUP LIST or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

GET GROUP PROPERTIES

version 6.0

GET GROUP PROPERTIES (groupID; name; owner{; members})

Parameter	Type		Description
groupID	Number	→	Unique group ID number
name	String	←	Name of the group
owner	Number	←	User ID number of group owner
members	Numeric Array	←	Group members

Description

GET GROUP PROPERTIES returns the properties of the group whose unique group ID number you pass in groupID. You must pass a valid group ID number returned by the command GET GROUP LIST. Group ID numbers can have the following values or ranges:

Group ID number	Group description
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

If you do not pass a valid group ID number, GET GROUP PROPERTIES returns empty parameters.

After the call, you retrieve the name and owner of the group, in the parameters name and owner.

If you pass the optional members parameter, the unique ID numbers of the users and groups belonging to the group are returned. Member ID numbers can have the following ranges:

Member ID number	Member Description
1	Designer user
2	Administrator user
3 to 15000	User created by the Designer of the database (user #3 is the first user created by the Designer, user #4 the second, and so on).
-11 to -15000	User created by the Administrator of the database (user #-11 is the first user created by the Designer, user #-12 is the second, and so on).
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

See Also

GET GROUP LIST, GET USER LIST, SET GROUP PROPERTIES.

Error Handling

If you do not have the proper access privileges for calling GET GROUP PROPERTIES or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

SET GROUP PROPERTIES

version 6.0

SET GROUP PROPERTIES (groupID; name; owner{; menbers})

Parameter	Type		Description
groupID	Number	→	Unique ID number of group, or -1 for adding a Designer group, or -2 for adding an Administrator group
		←	Unique ID number of new group
name	String	→	New group name
owner	Number	→	User ID number of new group owner
menbers	Numeric Array	→	New group members

Description

SET GROUP PROPERTIES enables you to change and update the properties of an existing group whose unique group ID number you pass in groupID, or to add a new group affiliated with the Designer or the Administrator.

If you are changing the properties of an existing group, you must pass a valid group ID number returned by the command GET GROUP LIST. Group ID numbers can have the following values or ranges:

Group ID number	Group description
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

To add a new group affiliated with the Designer, pass -1 in groupID. To add a new group affiliated with the Administrator, pass -2 in groupID. In both cases, when adding a new group, 4th Dimension tries to reuse the first available disabled group account; it creates a new account only if no disabled account is available. After the call, if the group is successfully added, its unique ID number is returned in groupID.

If you do not pass -1, -2 or a valid group ID number, SET GROUP PROPERTIES does nothing.

Before the call, you pass the new name and owner of the group in the parameters name and owner. If you do not want to change all the properties of the group (besides the members, see below), first call GET GROUP PROPERTIES and pass the returned values for the properties you want to leave unchanged.

If you do not pass the optional members parameter, the current member list of the group is left unchanged. If you do not pass members while adding a group, the group will have no members.

Note: The group owner is not automatically set as a member of the group that he or she owns. It is up to you to include the group owner in the group, using the members parameter.

If you pass the optional members parameter, you change the whole member list for the group. Before the call, you must populate the array members with the unique ID numbers of the users and groups the group will get as members. Member ID numbers can have the following ranges:

Member ID number	Member Description
1	Designer user
2	Administrator user
3 to 15000	User created by the Designer of the database (user #3 is the first user created by the Designer, user #4 the second, and so on).
-11 to -15000	User created by the Administrator of the database (user #-11 is the first user created by the Designer, user #-12 is the second, and so on).
15001 to 32767	Group created by the Designer or affiliated Group Owner (group #15001 is the first group created by the Designer, group #15002 the second, and so on).
-15001 to -32768	Group created by the Administrator or affiliated Group Owner (group #-15001 is the first group created by the Administrator, group #-15002 the second, and so on).

To remove all the members from a group, pass an empty members array.

See Also

GET GROUP LIST, GET GROUP PROPERTIES, GET USER LIST.

Error Handling

If you do not have the proper access privileges for calling SET GROUP PROPERTIES or if the Password system is already accessed by another process, an access privilege error is generated. You can catch this error with an error-handling method installed using ON ERR CALL.

53 Variables

SAVE VARIABLES

SAVE VARIABLES (document; variable{; variable2; ...; variableN})

Parameter	Type		Description
document	String	→	Document in which to save the variables
variable	Variable	→	Variables to save

Description

The command SAVE VARIABLES saves one or several variables in the document whose name you pass in document.

The variables do not need to be of the same type, but have to be of type String, Text, Real, Integer, Long Integer, Date, Time, Boolean, or Picture.

If you supply an empty string for document, the standard Save File dialog box appears; the user can then choose the document to create. In this case, the 4D system variable Document is set to the name of the document if one is created.

If the variables are properly saved, the OK variable is set to 1. If not, OK is set to 0.

Note: When you write variables to documents with SAVE VARIABLES, 4th Dimension uses an internal data format. You can retrieve the variables only with the LOAD VARIABLES command. Do not use RECEIVE VARIABLE or RECEIVE PACKET to read a document created by SAVE VARIABLES.

WARNING: This command does not support array variables. Use the new BLOB commands instead.

Example

The following example saves three variables to a document named UserPrefs:

```
⇒	SAVE VARIABLES ("User Prefs";vsName;vlCode;vgIconPicture)
```

System Variables or Sets

If the variables are saved properly, the OK system variable is set to 1; otherwise it is set to 0.

See Also

BLOB TO DOCUMENT, BLOB TO VARIABLE, DOCUMENT TO BLOB, LOAD VARIABLES, VARIABLE TO BLOB.

LOAD VARIABLES

LOAD VARIABLES (document; variable{; variable2; ...; variableN})

Parameter	Type		Description
document	String	→	Document containing 4D variables
variable	Variable	→	Variables to receive the values

Description

The command LOAD VARIABLES loads one or several variables from the document specified by document. The document must have been created using the command SAVE VARIABLES.

The variables variable, variable2...variableN are created; if they already exist, they are overwritten.

If you supply an empty string for document, the standard Open File dialog box appears, so the user can choose the document to open. If a document is chosen, the 4D system variable Document is set to the name of the document.

In compiled databases, each variable must be of the same type as those loaded from disk.

WARNING: This command does not support array variables. Use the new BLOB commands instead.

Example

The following example loads three variables from a document named UserPrefs:

```
⇒    LOAD VARIABLES ("User Prefs";vsName;vlCode;vgIconPicture)
```

System Variables or Sets

If the variables are loaded properly, the OK system variable is set to 1; otherwise it is set to 0.

See Also

BLOB TO DOCUMENT, BLOB TO VARIABLE, DOCUMENT TO BLOB, RECEIVE VARIABLE, VARIABLE TO BLOB.

CLEAR VARIABLE

CLEAR VARIABLE (variable)

Parameter	Type		Description
variable	Variable	→	Variable to clear

Description

This command acts differently in interpreted mode and in compiled mode.

In interpreted mode
CLEAR VARIABLE erases variable from memory. Consequently, the variable becomes undefined; trying to read its value will generate a syntax error. Note that if you again assign a value to the variable, 4D recreates the variable on the fly. After a variable is cleared, Undefined returns True when applied to that variable.

In compiled mode
CLEAR VARIABLE only resets variable to its default type value (i.e., empty string for String and Text variables, 0 for numeric variables, no elements for arrays, etc.). The variable still exists—variables can never be undefined in compiled code.

The variable you pass in variable must be a process or an interprocess variable.

Note: You do not need to clear process variables when a process ends; 4D clears them automatically.

Local variables, which are variables preceded by a dollar sign ($), cannot be cleared with CLEAR VARIABLE. They are cleared automatically when the method in which are located completes execution.

Example

In a form, you are using the drop-down list asMyDropDown whose sole purpose is user interface. In other words, you use that array during data entry, but once you are done with the form, you will no longer use that array. Consequently, during the On Unload event, you just get rid of the array:

```
          ` asMyDropDown drop-drop list object method
      Case of
          : (Form event=On Load)
                  ` Initialize the array one way or another
              ARRAY STRING(63;asMyDropDown;...)
                  ` ...
          :(Form event=On Unload)
                  ` No longer need the array
⇒             CLEAR VARIABLE (asMyDropDown)
                  ` ...
      End case
```

See Also

Undefined.

Undefined

version 3

Undefined (variable) → Boolean

Parameter	Type		Description
variable	Variable	→	Variable to test
Function result	Boolean	←	True = Variable is currently undefined False = Variable is currently defined

Description

Undefined returns True if variable has not been defined, and False if variable has been defined. A variable is defined if a value is assigned to it. A variable is undefined if it does not have a value assigned to it, or if it has been cleared with CLEAR VARIABLE.

If the database has been compiled using 4D Compiler, the Undefined function returns False for all variables.

Examples

1. Up to version 6, a good way to test if you were running in interpreted mode or in compiled mode was to write:

```
       anyVar:="Hello"
       CLEAR VARIABLE(anyVar)
⇒      If (Undefined(anyVar))
          ` You are in interpreted mode
       Else
          ` You are in compiled mode
       End if
```

Starting with version 6, it is more convenient to use the built-in command Compiled application.

2. The following code manages the creation of processes when a menu item for a particular module of your application is chosen. If the process already exists, you bring it to the front; if it does not exist, you start it. To do so, for each module of the application, you maintain an interprocess variable ◊PID_... that you initialize in the On Startup database method.

When developing the database, you add new modules. Instead modifying the On Startup database method (to add the initialization of the corresponding ◊PID_...) and then reopening the database to reinitialize everything each time you add a module, you use the Undefined command to manage the addition of the new module, on the fly:

```
          ` M_ADD_CUSTOMERS global procedure

          ` This line takes care of intermediate development stages
⇒      If (Undefined(◊PID_ADD_CUSTOMERS))
          C_LONGINT(◊PID_ADD_CUSTOMERS)
          ◊PID_ADD_CUSTOMERS:=0
       End if

       If  (◊PID_ADD_CUSTOMERS=0)
          ◊PID_ADD_CUSTOMERS:=New process("P_ADD_CUSTOMERS";64*1024;
                                                              "P_ADD_CUSTOMERS")
       Else
          SHOW PROCESS(◊PID_ADD_CUSTOMERS)
          BRING TO FRONT(◊PID_ADD_CUSTOMERS)
       End if
          ` Note: P_ADD_CUSTOMERS, the process master method, sets
          ` ◊PID_ADD_CUSTOMERS to zero when it ends.
```

See Also

CLEAR VARIABLE.

54 Web Server

Web Services, Overview

Web Server

version 6.0

Both 4th Dimension and 4D Server include a Web Server engine that enables you to publish 4D databases on the Web. The unique and unmatched features of the 4D Web Server engine are:

• **Direct Web services**
Your database is directly published on the Web. You do not need to develop a database system, a Web site, nor a CGI interface between them. Your database is your Web site.

• **On-line, transparent HTML translation**
On-line, 4D transparently and dynamically translates your forms and design components into HTML pages. These new HTML pages become instantaneously available to Web browsers, even though they are already connected to the database. Today, most Web database systems are either CGI-based systems or static HTML-based systems.

A CGI system requires you to develop a database, a Web site, and CGI. A static HTML-based system requires you to run a utility program that re-translates your modifications into HTML, each time you modify a design component in your database. In both cases, the Web components are created off-line and require manual intervention from the database and/or Web developer. However, with 4th Dimension, you can modify your design components as much as you want, whenever you want. As soon as you save the changes in the Design environment, your modifications are transparently made available to the Web browsers. So, while developing and testing your application, you can immediately test the result in an already connected Web browser.

• **Dynamic access to records and data**
4D handles Web browsers as standard clients of the 4D database engine. For example, if you modify some records on the local 4th Dimension database or from a client 4D Server workstation, those records become instantaneously available to the Web browsers. You do not need to re-process the records for HTML publishing, as in other systems.

• **Session maintenance and database context**
Web browsers, as the name indicates, enable you to browse Web pages in a random way—you can jump from one page to another one, from one Web site to another one, and so on. In the context of using Web browsers for Client/Server database purposes, you need to make the browser comply with the logic of the database transactions. For example, if you are adding a record, you need to enforce the rule that the record data entry must be validated or canceled. The user must not be able to exit the record via any browser navigation control, and leave it in an uncertain state. The 4D Web Server engine includes built-in session and database context maintenance. Throughout the URLs of the Web pages, it maintains unique context and subcontext ID numbers, which guarantee complete synchronization between the current Web page displayed in the browser and the context of the database connection on the 4D side.

• Transparent multi-user maintenance
When they become 4D clients, Web browsers are treated as complete database clients. For example, if you start modifying a record in a Web browser, 4D automatically locks the record, preventing any other concurrent client from modifying it. After you validate or cancel the data entry, 4D automatically unlocks the record. In addition, 4D allows you to perform data entry under transaction, as you are entitled to do using 4th Dimension or 4D Client.

• Web Process maintenance
4D includes several processes for handling its Web Client/Server architecture. The built-in main Web Server process handles Web connection attempts. After a Web browser has been granted access to the database, the Web connection starts running into a separate process automatically created for that purpose. As a result of the fully integrated multi-tasking architecture of 4D, regular or Web 4D clients can perform simultaneous database requests (such as queries) that will be processed in parallel by the database engine. Clients are also guaranteed that a request posted by a particular Web client will not intervene with the contexts of the other processes.

• Optimized Web Server archictecture
The 4D Web Server engine has the same capabilities as the 4D database engine. For example, if you load a series of record values to arrays, the operation is performed locally on the Web server machine. The result of the request is then sent, as a whole, to the requesting Web client.

In addition, because a Web connection is a plain and full featured 4D process on the Web server side, you can run any of your favorite 4D algorithms. They are executed locally on the Web server side, and only the result, if any, goes to the Web browser. For example, you can perform a query that involves relations, sets, and the computation of statistical results, with only the result returning the Web browser. You can build a complex Web database system without having to become a Web expert, because you can compile your 4D applications.

• HTML and JavaScript encapsulation
Although 4D performs almost everything you need to publish databases on the Web, you can encapsulate HTML and JavaScript code to customize your 4D development. For example, you may want to enhance the Home page of your Database/Web site with an eye-catching HTML page.
You can build an entire custom HTML page and put it on the Web using the SEND HTML FILE command. You can also encapsulate HTML into a 4D form whose appearance on the Web browser is a mixed concatenation of the 4D and HTML objects present in the form. Within the encapsulated HTML, you then can implement JavaScript code that performs actions and data control on the client Web browser side, without requiring a request to be sent back to the server.

• Binding between HTML and 4D objects
If you use encapsulated HTML code within your 4D development, you need, on the 4D side, to be able to retrieve the values and data that have been entered into the HTML objects. Rather than making you write complicated HTML parsing routines, 4D provides a simple, built-in system for binding HTML objects to your 4D variables—your objects just need to have the same name. As a result, analyzing and responding to HTML requests becomes quite simple to implement. You just need to write 4D code that deals with the 4D variables automatically filled out by the data that came back from the Web browser.

Where to go from here?

• To setup your machine and databases for Web publishing, read Web Services, Configuration.
• To learn how to publish a database on the Web, read Web Services, Your First Time (Part I) and Web Services, Your First Time (Part II).
• To learn more about HTML encapsulation, read Web Services, HTML and JavaScript Encapsulation.
• To learn more about Web and Process interaction, read Web Services, Web Connection Processes.

See Also

SEND HTML FILE, SET HTML ROOT, SET HTML ROOT, SET WEB DISPLAY LIMITS, SET WEB TIMEOUT, STOP WEB SERVER.

4th Dimension and 4D Server 6.0 include Web Services that enable you to serve your database on the Web, transparently and dynamically.

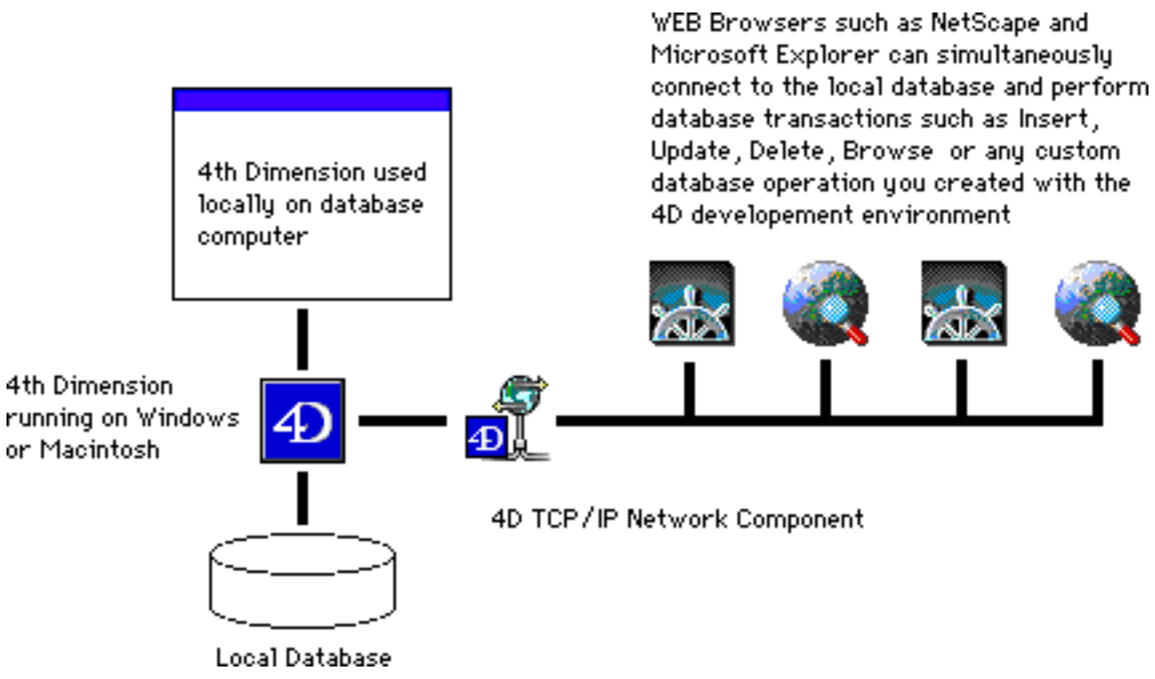
4th Dimension and the Web

If you publish a 4D database on the Web using 4th Dimension, you can simultaneously:

• Use the database locally with 4D

• Connect to the database using Web browsers

This is summarized in the following diagram:

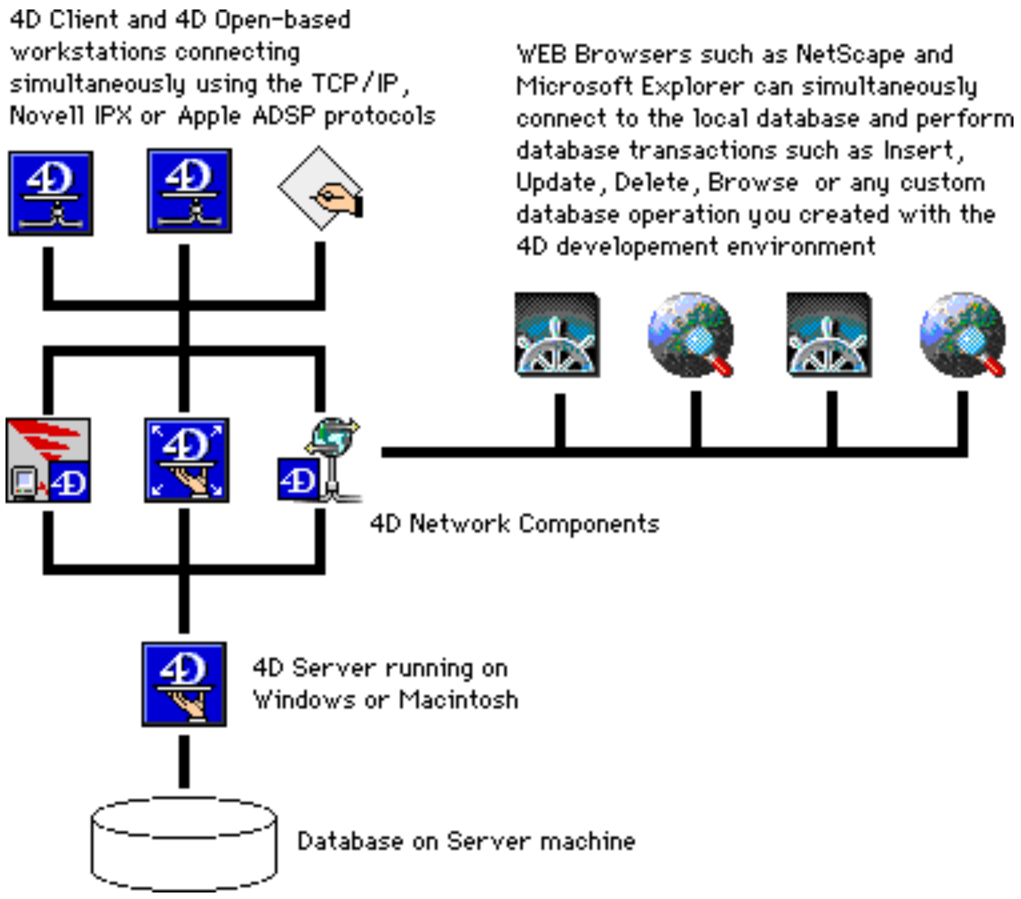

WEB Browsers such as NetScape and Microsoft Explorer can simultaneously connect to the local database and perform database transactions such as Insert, Update, Delete, Browse or any custom database operation you created with the 4D developement environment
4th Dimension used locally on database computer
4th Dimension running on Windows or Macintosh
4D
4D TCP/IP Network Component
Local Database

4D Server and the Web

If you publish a 4D database on the Web using 4D Server, you can simultaneously connect to and operate the 4D database, using:

- 4D Client workstations
- 4D Open-based applications
- Web browsers

This is summarized in the following diagram:


4D Client and 4D Open-based workstations connecting simultaneously using the TCP/IP, Novell IPX or Apple ADSP protocols
WEB Browsers such as NetScape and Microsoft Explorer can simultaneously connect to the local database and perform database transactions such as Insert, Update, Delete, Browse or any custom database operation you created with the 4D developement environment
4D Network Components
4D Server running on Windows or Macintosh
Database on Server machine

Serving a 4D database on the Web

To serve a 4D database on the Web using 4th Dimension or 4D Server, you must have the approriate connection licenses and network components:

• The required Web connection licenses must be installed in your application. For more information, please refer to your *4D Installation Guide*.

• Web connections are made over the network using the TCP/IP protocol. Consequently:

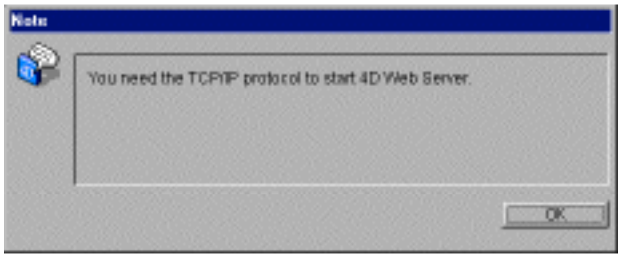
- You must have TCP/IP installed on your machine and correctly configured. Refer to your computer or Operating System manuals for more information.

- You must have the 4D TCP/IP Network component installed: On Macintosh, the network component must be installed directly into 4th Dimension, 4D Server or 4D custom merged application. On Windows, the network component must be installed into the ACI\NETWORK directory of your active WINDOWS directory.

Note: In both cases, refer to the Network Components manual for more information.

• After the Web connection license and TCP/IP have been installed or checked, you need to start the Web Services from within 4D. This last point is discussed in the next section.

If you try to start the 4D Web Services while TCP/IP is not running or while the 4D TCP/IP network component is not present, 4th Dimension displays the following alert:



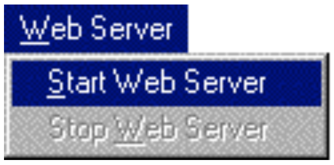
You need the TCP/IP protocol to start 4D Web Server.

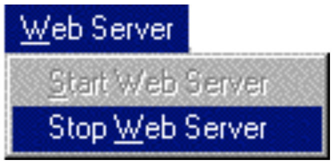
If this message appears, perform the installations as described, or troubleshoot your TCP/IP configuration.

Starting the 4D Web Services

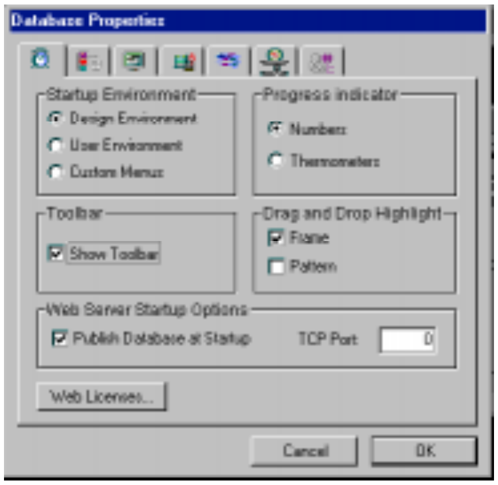
The 4D Web Services can be started in three different ways:

• Using the Web Server menu from the main menu bar of 4D Server or the 4th Dimension User environment. The Web Server menu allows you to start and stop the Web Services at your convenience:





• Automatically publishing the database each time it is opened. To automatically publish a database on the Web, choose the **File** menu's **Database Properties...** option from the main menu bar of 4D Server or the 4th Dimension Design environment. The Database Properties window appears:


Database Properties

Startup Environment: [✓ Design Environment] User Environment, Custom Menus

Progress indicator: [✓ Numbers] Thermometers

Toolbar: [✓ Show Toolbar]

Drag and Drop Highlight: [✓ Frame] Pattern

Web Server Startup Options: [✓ Publish Database at Startup] TCP Port 0

In the **Web Server Startup Options** section, select the **Publish Database at Startup** check box, then click OK. Once this is done, the database will be automatically published on the Web each time you open it with 4th Dimension or 4D Server.

• Programmatically, by calling the command START WEB SERVER.

Tip: You do not need to quit 4D and reopen your database to start or stop publishing a database on the Web. You can interrupt and restart the Web services as many times as you want, using the Web Server menu or calling the commands START WEB SERVER and STOP WEB SERVER.

Connecting to a 4D database published on the Web

After you have started publishing a 4D database on the Web, you can connect to it using a Web browser. To do so:

• If your Web site has a registered name (i.e., "Flowers International"), indicate that name in the Open, Address, or Location area of your browser. Then type Enter to connect.

• If your Web Site does not have a registered name, indicate the IP address of your machine
(i.e., 123.4.567.89) in the Open, Address, or Location area of your browser. Then type Enter.

At this point, you can connect if everything is fine. If you cannot connect, you may encounter one of the following situations:

1. You get a message such as "...the server may not be accepting connections or may be busy...".
In this case, check the following:
• Verify that the name or the IP address you entered is correct.
• Verify that 4th Dimension or 4D Server is up running and has started its Web Services.
• Check if the database is configured for being served on a TCP Port other than the default Web TCP Port (see situation 3).
• Check whether TCP/IP is correctly configured on both the server and browser machines. Both machines must be on the same net and subnet, or your routers must be correctly configured.
• Check your hardware connections.
• If you are not locally testing your own site, but rather attempting to connect to a Web database served on Internet or Intranet by someone else, ultimately, the message might be true: the server may be off or busy. So, retry later until you can log on, or contact the Web provider.

2. You connect, but you get a Web page with the message "This database has not been setup for the Web yet". This means that the 4D Web Services are up running and you correctly connected to the database. However, the database needs to include some minimal components in order to be operable over the Web. For more information, see the section Web Services, Your First Time (Part I). Note: This could also mean that the database is currently out of available Web Licenses.

(3) You connect, but you do NOT obtain the Web page you were expecting! This can occur when you have several Web servers running simultaneously on the same machine. Examples:

• You are running only one 4D Web database on a Windows NT 4.0 system that is already running its own Web services.

• You are running several 4D Web databases on the same machine.

In this kind of situation, you need to change the TCP port number on which your 4D Web database is published. To do so, read the next section.

Setting the TCP port number to a specific value

By default, 4D publishes a Web database on the regular Web TCP Port, which is port 80. If that port is already used by another Web service, you need to change the TCP Port used by 4D for this database. To do so, choose the **File** menu's **Database Properties...** menu item from the main menu bar of 4D Server or the 4th Dimension Design environment (see the screen shot above). Go to the **TCP Port** enterable area and indicate an appropriate value (a TCP port not already used by another TCP/IP service running on the same machine).

Note: If you specify 0, 4D will use the default TCP port number 80.

From a Web browser, you need to include that non-default TCP port number into the address you enter for connecting to the Web database. The address must have a suffix consisting of a colon followed by the port number. For example, if you are using the TCP port number 700, you will specify “123.4.567.89:700”.

WARNING: If you use TCP port numbers other than the default 80, be careful not to use port numbers that are defaults for other services that you might want to use simultaneously. For example, if you also plan to use the FTP protocol on your Web server machine, do not use the TCP port 20 and 21, which are the default ports for that protocol (unless you know what you are doing). For more information about default TCP port numbers and protocols, buy any book about the TCP/IP protocol and look for a table of RFC 1700 standard assigned numbers. Ports numbers below 256 are reserved for well known services and ports numbers from 256 to 1024 are reserved for specific services originated on the UNIX platforms. If you use a port number in the few thousands, you will be OK.

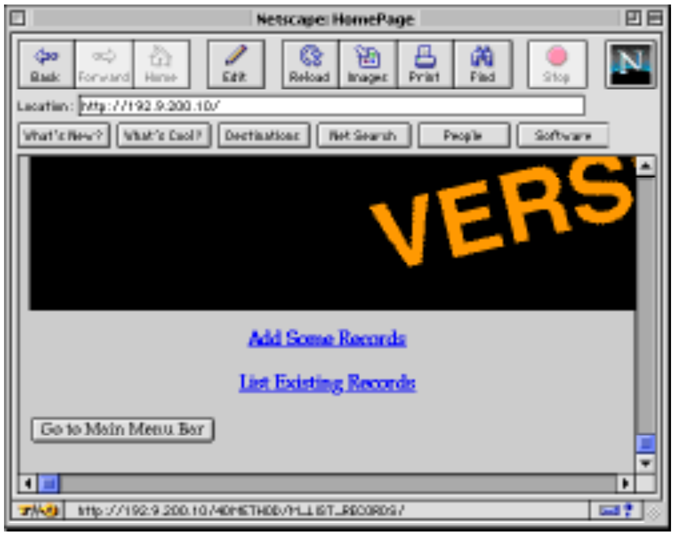
See Also

SEND HTML FILE, SET HTML ROOT, SET WEB DISPLAY LIMITS, SET WEB TIMEOUT, STOP WEB SERVER.

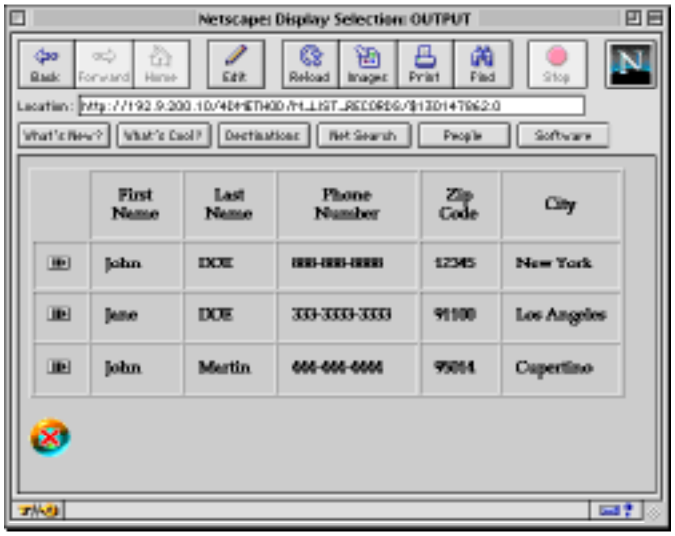
Example

Among the examples provided with 4th Dimension, you will find a database named WebDemo1. When you open this database on Windows or Macintosh, it should automatically publish itself on your network as a Web Server. If error messages appear while opening the database, please refer to the section Web Services, Configuration for troubleshooting.

1. Connect to the Web server database
Connect to the Web Server you just started by opening the database on a Web browser running on a second machine. You should get a Web page similar to the one shown here, which was obtained with Netscape running on Macintosh:



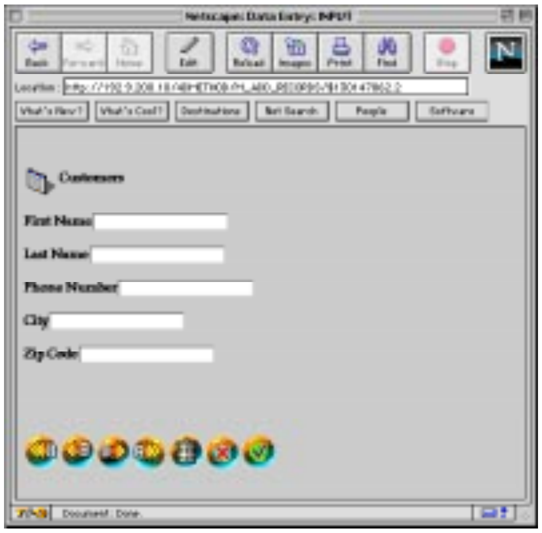
2. Display and browse the records
Click on the linked text List Existing Records. This presents the Web equivalent of a 4D Display Selection screen:



	First Name	Last Name	Phone Number	Zip Code	City
	John	DOE	888-888-8888	12345	New York
	Jane	DOE	333-3333-3333	91100	Los Angeles
	John	Martin	666-666-6666	95014	Cupertino

At this point, you can browse the records at your convenience. After you click the Done button (the one with the red X), you go back to the Web site Home page.

3. Add records
In the Web site Home page, click on the Linked text Add Some Records to display the Web equivalent of a 4D Add Record screen:

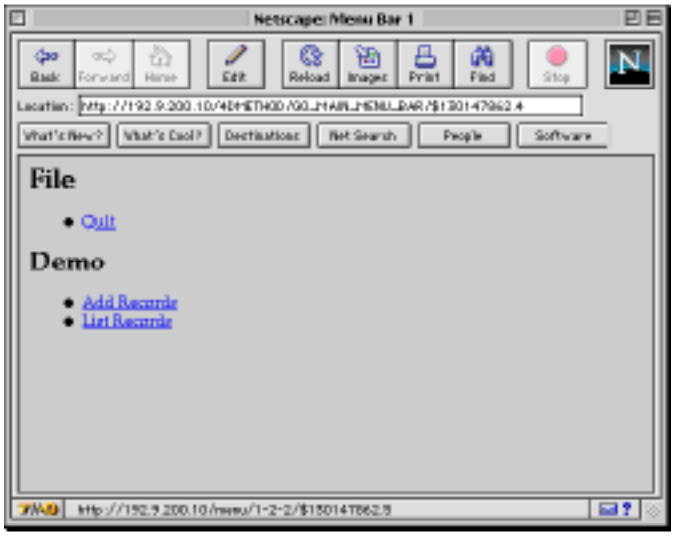


Customers

First Name
Last Name
Phone Number
City
Zip Code

You can add as many as records as you wish. When you are done, click the Cancel button (the one with the red cross) to return to the Web site Home page.

4. List or add records in the Main menu
In the Home page, click the Go to Main Menu Bar button. This exits the Home page and presents the Web equivalent of the 4D Custom menu bar:



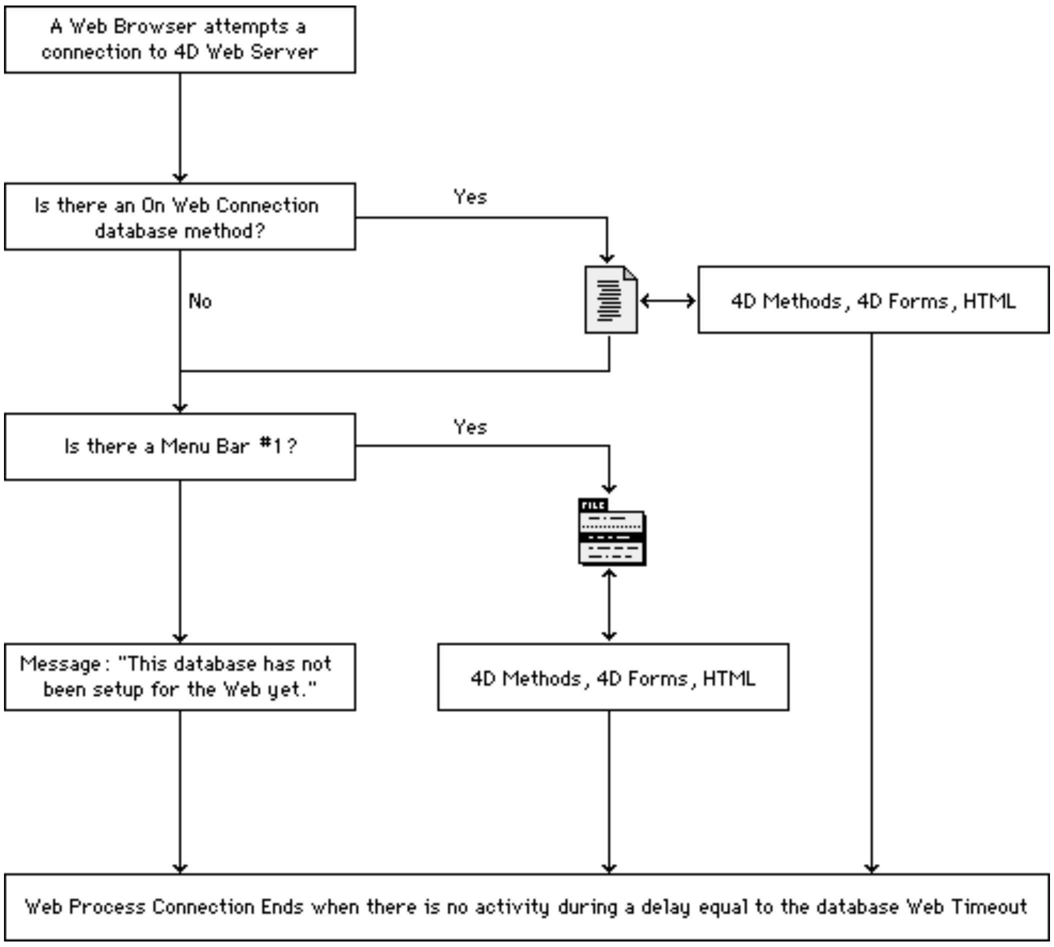
At this point, clicking on each menu item allows you to List or Add records: the same 4D methods that were used from the Home page are associated to the menu items.

5. Terminate the connection
When you are done, just quit your browser. 4D will terminate the Web connection process once the timeout delay has elapsed.

Initiating a Web connection

Each time a Web browser connects to a 4D database published as a Web Server, 4D performs the following actions:
• It executes the On Web Connection database method, if it exists.
• If there is no such database method or if the method has been achieved, 4D then displays the menu bar #1, if it exists.
• If there is no menu bar, 4th Dimension displays a default Web page that states: *"This database has not been setup for the Web yet"*.

The following diagram summarizes these actions:


A Web Browser attempts a connection to 4D Web Server
Is there an On Web Connection database method?
Yes
No
4D Methods, 4D Forms, HTML
Is there a Menu Bar #1 ?
Yes
Message : "This database has not been setup for the Web yet."
4D Methods, 4D Forms, HTML
Web Process Connection Ends when there is no activity during a delay equal to the database Web Timeout

The On Web connection database method can call any of the project methods or forms defined in the database as well as HTML pages. The database method can actually handle the whole session.

A Web connection to 4D or 4D Server is not the same as a Client/Server connection. The HTTP protocol, which supports HTML and the Web, is not a “session-based” protocol; it is rather a “request-based” protocol. In Client/Server, you connect, work in a session, and then disconnect from the server. With HTTP, each time you perform an action that requires the attention of or an action from the Web Server, a request is sent to the server. In short, an HTTP request can be understood as the sequence “Connect+Request+Wait for reply+Disconnect.”

In order to run a Client/Server session above HTTP, 4D maintains, through a transparent encoding of the URLs, a context that uniquely identifies your Web connection and at the same time associates the connection to the 4D process handling the connection.

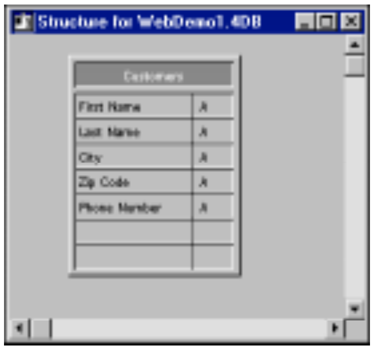
However, 4D has no way to provide an equivalent of the Client/Server disconnect action that terminates a session. That is the reason why the termination of a Client/Server session is handled through a timeout scheme. The 4D process handling the Web connection terminates after no activity has been detected for a delay time equal to the database Web timeout settings.

Database and Web Server in one

You can completely manage a 4D Web Server session using 4D Menus Bars, Forms and Methods. In the preceding example, listing and adding records was performed by simple 4D methods and forms. If we had not included an HTML home page, a Web browser would have obtained, upon connection, the menu bar #1 shown.

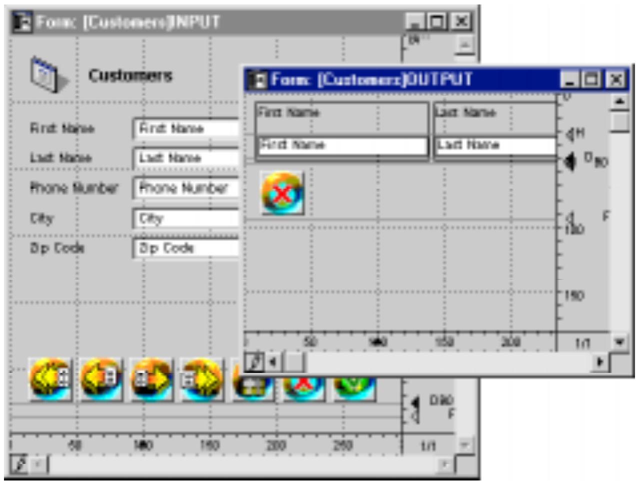
If we eliminate the HTML home page, building a Web Server supporting database Client/Server transactions consists of building a 4D database on Windows or Macintosh, for one or multiple users. The following steps explain the process of creating the example database in this way.

1. Here is the Structure of the example database:

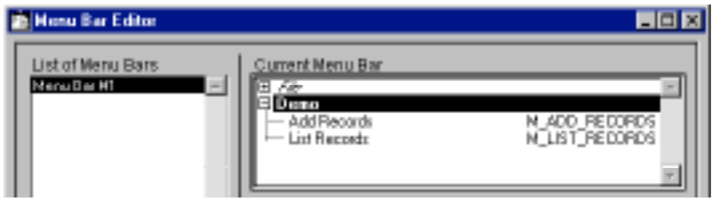


First Name	A
Last Name	A
City	A
Zip Code	A
Phone Number	A

2. Input and Output forms are added to enable you to work with records.

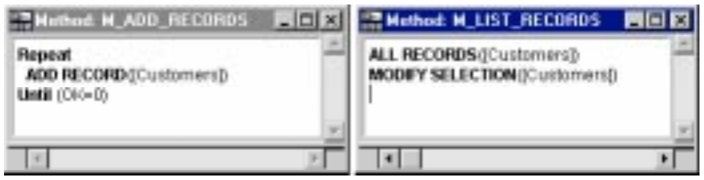


3. Menu bar #1 is added to enable you to work with Custom menus and to support Web connections.



Add Records	M_ADD_RECORDS
List Records	M_LIST_RECORDS

4. Two project methods are written.



```
Repeat
  ADD RECORD([Customers])
Until (OK=0)
```

```
ALL RECORDS([Customers])
MODIFY SELECTION([Customers])
```

That is it! In less than five minutes, you have created a 4D database that is both a locally operable database and a Web Server that you can publish on your Intranet network or on the Internet.

Go to Web Services, Your First Time (Part II).

See Also

SEND HTML FILE, SET WEB DISPLAY LIMITS, SET WEB DISPLAY LIMITS, SET WEB TIMEOUT, START WEB SERVER, STOP WEB SERVER.

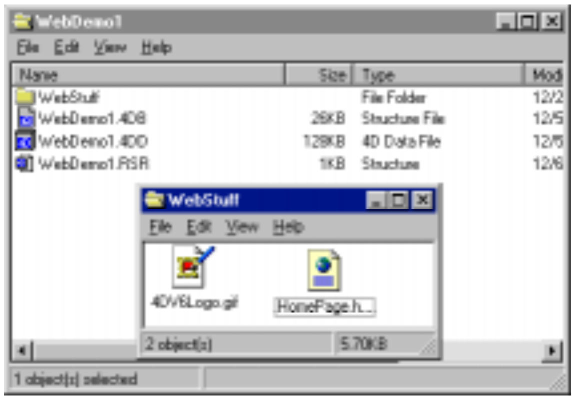
Adding an HTML touch to a database

If you want Web users to interact with more than just the menu bar #1 of your database, you can add an On Web connection database method that will display either a 4D form or an HTML page. You can reuse HTML pages from existing sites created with any HTML tool.

Your 4D-based Web Site can be a completely 4D-based system or a combination of 4D forms and HTML pages. The interesting point in using HTML pages from within your 4D database is that you benefit from both the 4D and HTML development environments. Remember, you do not have to use HTML pages if you do not want to!

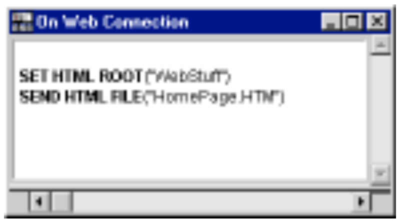
Example

In this example, we add an existing HTML page to the database. The following graphic shows directory of the database:



Name	Size	Type	Mod
WebStuff		File Folder	12/2
WebDemo1.4DB	26KB	Structure File	12/5
WebDemo1.4DD	128KB	4D Data File	12/5
WebDemo1.RSR	1KB	Structure	12/6

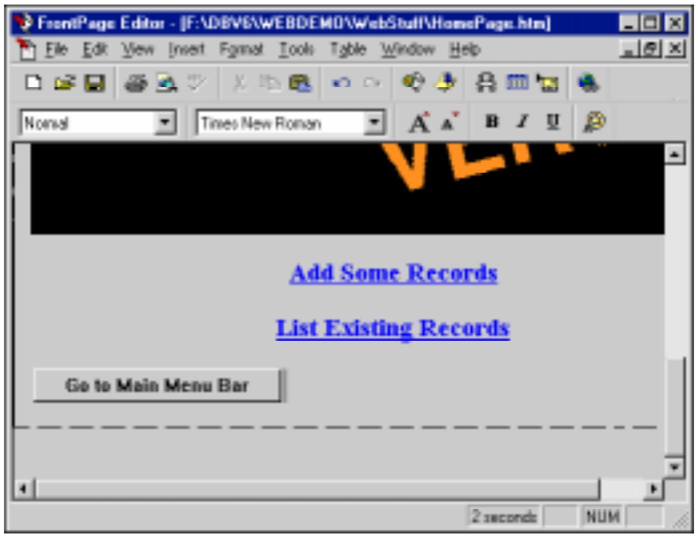
We will use the HTML document HomePage.HTM as the home page for the database. The document 4DV6Logo.GIF is a picture used within the HTML document. The HTML document is added to the 4D database by the On Web Connection database method shown here:



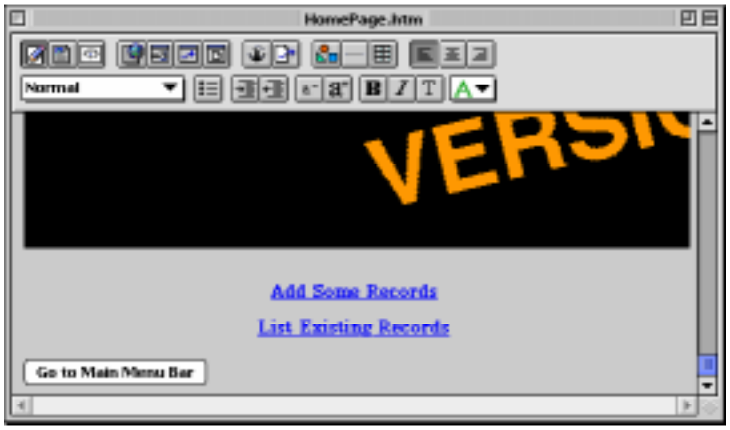
```
SET HTML ROOT("WebStuff")
SEND HTML FILE("HomePage.HTM")
```

The command SET HTML ROOT tells 4th Dimension where to look (by default) for the HTML documents. The command SEND HTML FILE sends the HTML document as current Web page to the connected Web Browser.

The following is the HomePage.HTM document viewed with Microsoft Front Page:


FrontPage Editor - [F:\DBV6\WEBDEMO\WebStuff\HomePage.htm]
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
Add Some Records
List Existing Records
Go to Main Menu Bar
2 seconds
NUM

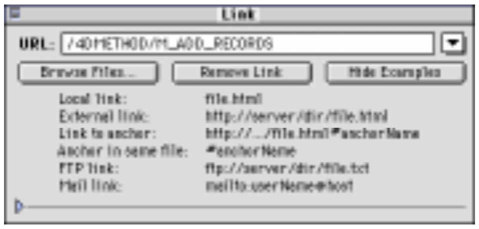
The following is the same HTML document viewed with Claris Home Page:



Linking URLs

The two linked text items, “Add Some Records” and “List Existing Records,” trigger the execution of the 4D project methods M_ADD_RECORDS and M_LIST_RECORDS through their URLs. The convention is quite simple: any HTML object can link to a project method of your database with the URL "/4DMETHOD/Name_of_your_Method".

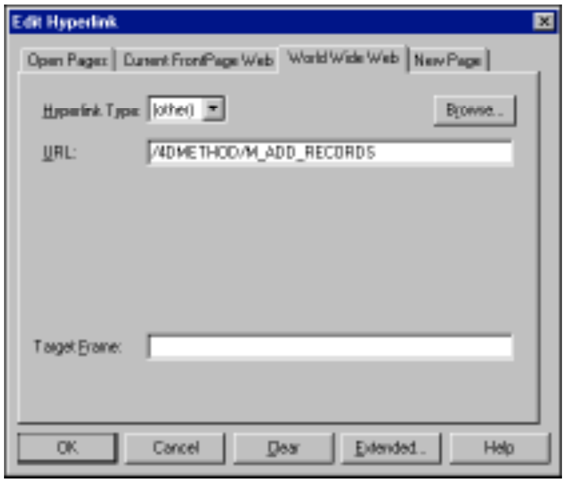
Here is the URL for the text “Add Some Records,” in Claris Home Page:



URL: /4DMETHOD/M_ADD_RECORDS

Local link:	file.html
External link:	http://server/dir/file.html
Link to anchor:	http://.../file.html#anchorName
Anchor in same file:	#anchorName
FTP link:	ftp://server/dir/file.txt
Mail link:	mailto:userName@host

Here is the same URL, in Microsoft Front Page:



Hyperlink Type: (other)

URL: /4DMETHOD/M_ADD_RECORDS

Target Frame:

After these links have been defined, when the Web browser sends back the URL, 4D executes the project method specified after the /4DMETHOD/ keyword. Then, after the project method has been completed, you back to the HTML page that triggered its execution. Note that the project method can itself display 4D forms, other HTML pages, and so on.

Buttons

The HTML document in this example includes a button used to submit a record. There are three types of HTML buttons: normal, submit, and reset.

• Normal - Normal buttons can be attributed an URL that refers to a 4D method using the /4DMETHOD/ keyword. Normal buttons are used for navigation purposes.

• Submit - Submit buttons submit the form with the values entered by the user (if any) to the Web server. They are useful for handling data entry that you prefer to perform via an HTML page rather than a plain 4D form

• Reset - Reset buttons are not very useful within a 4D development: they clear the form of the values entered by the user (if any) and does not send any request to the server.

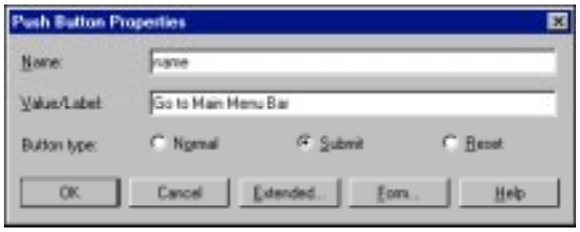
While integrating HTML pages into 4D, you will typically use normal or submit type buttons.

Specifying the 4D method to be executed

To submit the HTML form on the 4D side, you need to specify the POST action 4D method that will be executed by 4D after the form is submitted.

Specifying the POST action 4D method, using Microsoft Front Page:

1. Examine the properties of the submit button. The following dialog box appears:

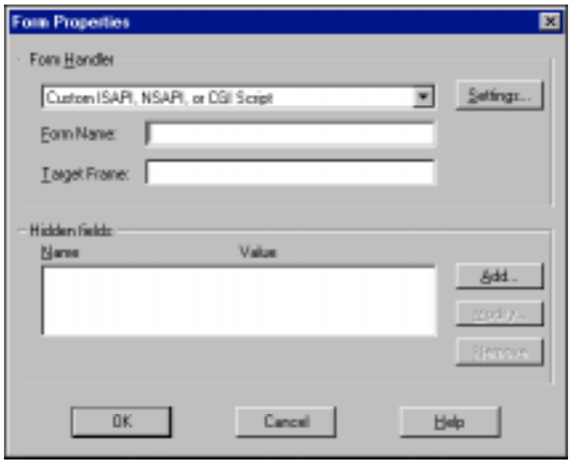

Push Button Properties

Name: name

Value/Label: Go to Main Menu Bar

Button type: Normal [✓ Submit] Reset

2. Click the Form... button. The Form Properties dialog box appears:


Form Properties

Form Handler: Custom ISAPI, NSAPI, or CGI Script

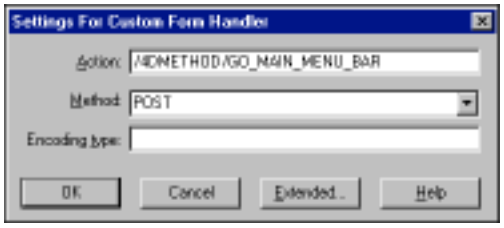
Form Name:

Target Frame:

Hidden fields:

Name	Value

3. Click on the Settings button. The following dialog box appears:


Settings For Custom Form Handler

Action: /4DMETHOD/GO_MAIN_MENU_BAR

Method: POST

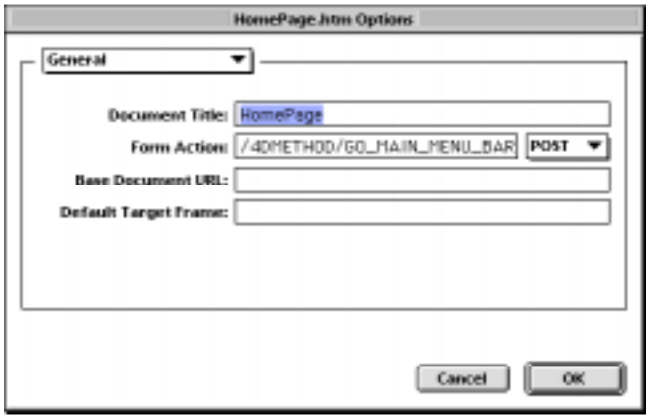
Encoding type:

4. As for a linked URL, specify "/4DMETHOD/Name_of_your_Method" as the Action. Here, we enter GO_MAIN_MENU_BAR as 4D method to be executed when the Go Main Menu Bar submit button is clicked.

5. Select **POST** from the Method drop-down list.

Specifying the POST action 4D method, using Claris Home Page:

1. Select **Document Options...** from the Edit menu. The following dialog box appears:


HomePage.htm Options

General

Document Title: HomePage

Form Action: /4DMETHOD/GO_MAIN_MENU_BAR POST

Base Document URL:

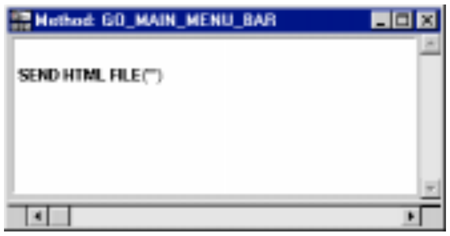
Default Target Frame:

2. Select **POST** from the pop-up menu.

3. Enter "/4DMETHOD/Name_of_your_Method" as **Form Action**.

The project method

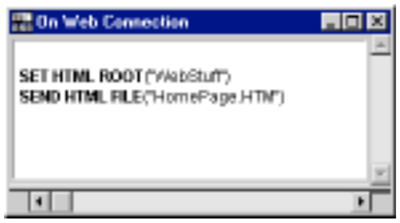
The GO_MAIN_MENU_BAR 4D project method is shown here:


Method: GO_MAIN_MENU_BAR

```
SEND HTML FILE("")
```

In this example, this method has only one purpose: getting out of the current HTML page displayed on the Web browser. To do so, it simply calls SEND HTML FILE, passing an empty string. This tells 4D to exit the current HTML page and go back to the line of code of the 4D project method that issued the SEND HTML FILE, which started the “HTML mode.”

In this example, that means that we go back to executing the On Web connection database method:



```
SET HTML ROOT("WebStuff")
SEND HTML FILE("HomePage.HTM")
```

The SEND HTML FILE call was the last line of that method, therefore the On Web connection database method consequently ends, and 4D switches to the menu bar #1 of the database.

That is it! In five minutes, by designing the Web page and adding the GO_MAIN_MENU_BAR 4D project method, you have a database and a Web server that combines Client/Server capabilities with HTML development.

Where to go from here?

• For a complete description about integrating HTML forms and code into 4D see the section, Web Services, HTML and Javascript Encapsulation.

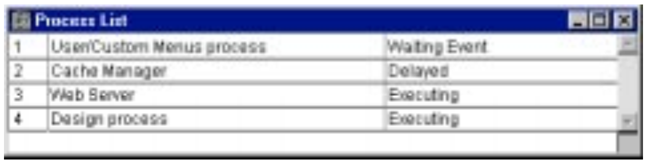
• If you have trouble while setting up your first 4D Web Server, see the section Web Services, Configuration.

See Also

SEND HTML FILE, SET HTML ROOT, SET WEB TIMEOUT, START WEB SERVER, STOP WEB SERVER.

Web Server Process

The **Web Server** process runs and executes when the database is being published as a Web site.
In the Design **Process List** window shown here, the Web Server process is the third process that is running and executing:



1	User/Custom Menus process	Waiting Event
2	Cache Manager	Delayed
3	Web Server	Executing
4	Design process	Executing

This is a 4D kernel process; you cannot abort this process using the **Abort** menu item of the **Design Process** menu. Also, you cannot attempt interprocess communication using commands such as CALL PROCESS. Note that the Web Server process does not have any user interface components (windows, menus, and so on).

You can start the Web Server process in the following ways:
• Choose **Start Web Server** from the User environment **Web Server** menu.
• Call the 4D command START WEB SERVER.
• Open a database whose **Publish Database at Startup** Database Properties is checked.

You can stop running the Web Server process in the following ways:
• Choose **Stop Web Server** from the User environment **Web Server** menu.
• Call the 4D command STOP WEB SERVER.
• Quit the database being currently published.

The purpose of the Web Server process is only to handle Web connection attempts. Starting the Web Server process does not mean that you open an actual Web connection, it just means that you allow Web users to initiate Web connections. Stopping the Web Server process does not mean that you close currently running Web connection processes (if any), it just means that you no longer allow Web users to initiate new Web connections.

If there are open Web connection processes when you stop the Web Server process, each of these processes continues executing until the Web user stops querying the database for a delay time greater or equal to the **Web Server Connections Timeout** (set in the Database Properties window or set programmatically using the SET WEB TIMEOUT command).

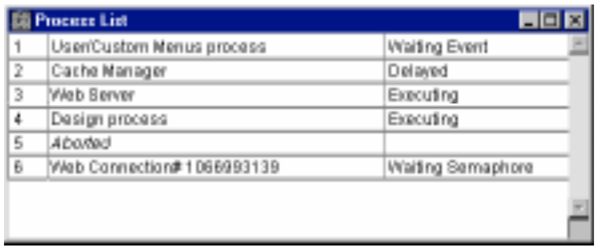
Web Connection Processes

Each time a Web browser attempts to connect to the database, the request is handled by the Web Server process, which performs the following steps:

• First, it checks to verify that there is an available Web License for the new connection. If all licenses are already being used, it sends the following message to the Web browser: *“This database has not been setup for the Web yet.”*

• If there is a license available, the Web Server process creates a temporary local 4D process to initiate the connection with the Web browser. This temporary process executes very quickly and then aborts.

• If the Web connection is initiated successfully, then a **Web Connection process** is started. This is the process that will handle the entire Web session for that connection. The Design Process List window shown here displays the Web connection process “Web Connection# 1066993139,” started after a Web browser connection has been initiated:

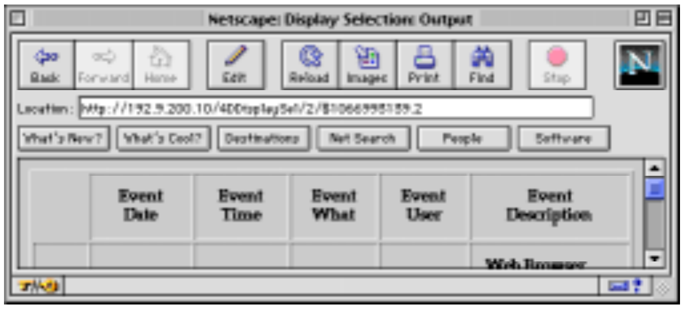


1	User/Custom Menus process	Waiting Event
2	Cache Manager	Delayed
3	Web Server	Executing
4	Design process	Executing
5	*Aborted*	
6	Web Connection# 1066993139	Waiting Semaphore

Also note the aborted fifth process, which was started and stopped by the Web Server process; this process handled the initialization of the Web connection.

Web Connection Context ID

The number present in the name of the Web connection process is called the **context ID**, which is randomly generated and uniquely identifies each Web connection. The context ID is maintained on both the 4D and the browser sides during the entire Web connection. In this example, the context ID is 1066993139. In the Web browser window shown here, you can see this number in the URL displayed in the Location area of the browser:



Location: http://192.9.200.10/4DDisplaySel/2/$1066993139.2

	Event Date	Event Time	Event What	Event User	Event Description
					Web Browser

The URLs are automatically maintained by 4D during the whole Web session. Each time an HTTP request is received by the 4D TCP/IP Network component, 4D extracts the context ID from the URL, and thereby can redirect the request to the right Web Connection process.

Context IDs:

• Enable 4D to maintain both a Web and Database session over each Web connection.
• Transparently handle multiple concurrent Web connections.
• Prevent future undesirable connections when using bookmarks, because a different context ID is generated at each connection.

Synchronizing Web and Database sessions: Web Connection Subcontext ID

In the window shown above, note that the context ID is followed by a dot and a second number, called the **subcontext ID**. 4D automatically manages and increments this number each time a new 4D-based HTML page is sent to the browser. The subcontext ID is essential to the maintenance of the database session.

Usually, a Web browser includes navigation controls, such as the Back and Forward buttons, History windows, and so on. These controls are useful when you are browsing documents, news, bulletin boards, etc. They are less appealing when you perform a database transaction.

For example, if a Web user is adding a record to a table, you need to know whether or not the data entry is validated, that is, whether or not the Web user clicked the Accept or Cancel buttons of your 4D form. If, at this point, the Web user navigates to other pages, the data entry is left in an uncertain state. To prevent this, 4D uses the subcontext ID to synchronize the Web session on the browser side with the database session on the 4D side.

Each time a form is submitted or an HTTP request is sent to 4D by the browser, if a desynchronization of the Web and database sessions is detected, 4D sends the message *“Using browser navigation controls, you left a form requiring data validation. 4th Dimension will now return to that form so you can accept or cancel it.”* 4D then goes back to the Web data entry page using the subcontext ID.

This synchronization is also essential for the Web Connection process. You need to correctly get out of, for example, an ADD RECORD ([...]) to pursue the execution of your 4D code.

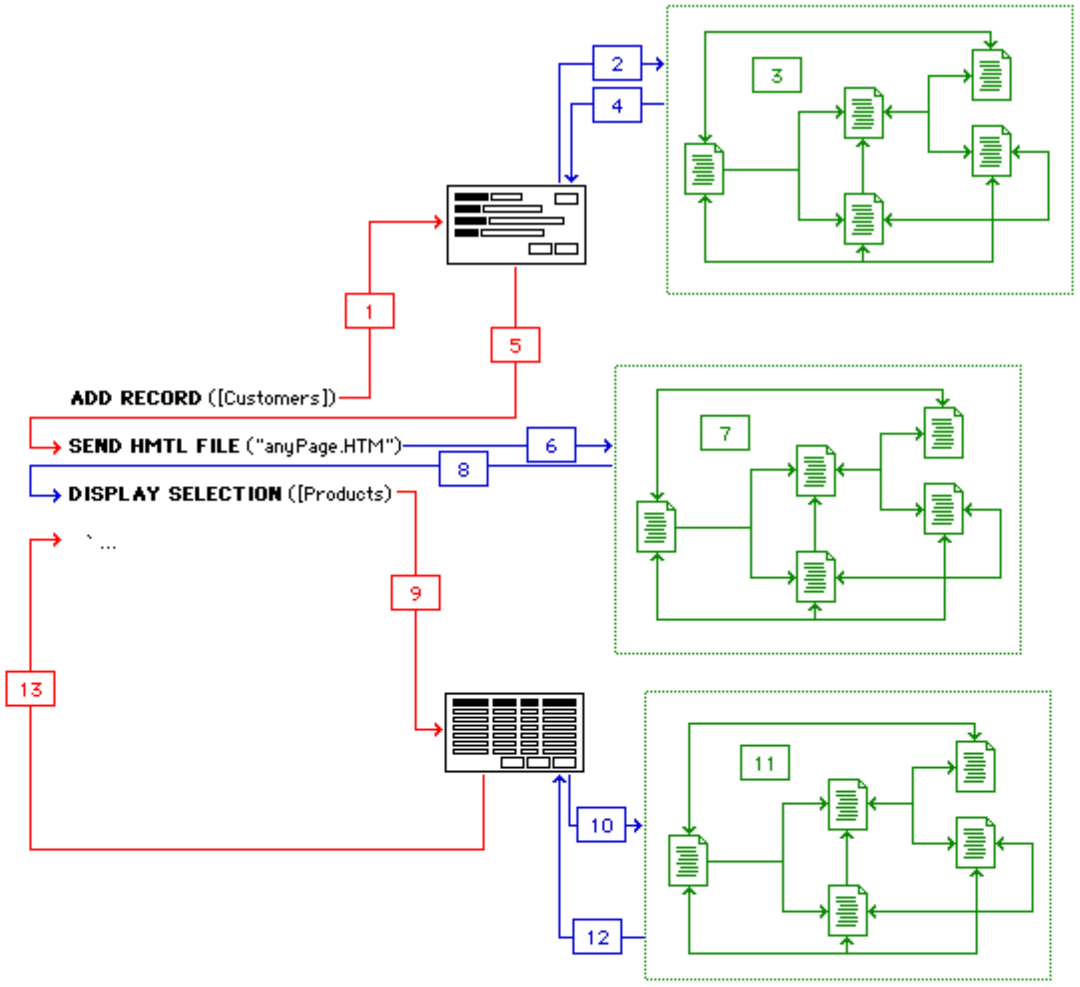
The synchronization is selective. If the current Web page displayed on the browser side is a 4D form (ADD RECORD, DISPLAY SELECTION, DIALOG, etc.), the synchronization will eventuallly occur.
If the current Web page is an HTML page accessed by link from another Web page (sent using the command SEND HTML FILE), then you can navigate freely through the pages.

Given the following piece of 4D code:

```
ADD RECORD ([Customers])
SEND HTML FILE ("anyPage.HTM")
DISPLAY SELECTION ([Products])
```

The following figure details what happens both on 4D and the Web browser during execution.

• Lines in red denote 4D form translations and submissions.
• Lines in blue denote switching back and forth between 4D-based and non 4D-based HTML pages.
• Areas in green denote non 4D-based HTML pages.



ADD RECORD ([Customers])
SEND HMTL FILE ("anyPage.HTM")
DISPLAY SELECTION ([Products])
` ...
1
2
3
4
5
6
7
8
9
10
11
12
13

Description of the steps

(1) An ADD RECORD is issued. 4D translates the current input form of the table into an HTML page and sends it to the Web browser. If the form is a multi-page form, the standard 4D page navigation buttons allow you to navigate through the pages of the form. This 4D-based navigation is implemented and performed transparently by 4D (via Web form submission).

(2) During data entry (therefore within the ADD RECORD call), a button is clicked and its object method issues a SEND HTML FILE call.

(3) Within the SEND HTML FILE call, if the HTML page includes links, it is possible to navigate through several pages. Eventually, when a SEND HTML FILE("") is issued, the HTML mode is exited.

(4) The object method of the button that was clicked and the data entry initiated by ADD RECORD are executed. Note that steps (2) and (3) can be repeated several times within the data entry.

(5) Finally, the data entry is accepted or canceled, and the Web Connection process is executed.

(6) The next call is a SEND HTML FILE.

(7) This step is analogous to step 3. If the HTML page includes links, it is possible to navigate through several pages. Eventually, when a SEND HTML FILE("") is issued, the HTML mode is exited.

(8) The Web Connection process is executed.

(9) A DISPLAY SELECTION is issued. 4D translates the current output form of the table into an HTML page and sends to the Web browser. During the DISPLAY SELECTION, 4D transparently navigates between the selection page and the display of individual records. 4D also uses MODIFY SELECTION to manage data entry and record locking, via Web form submissions.

(10) During navigation through the selection, a button in the footer area of the form is clicked and its object method issues a SEND HTML FILE call.

(11) This step is analogous to steps 7 and 3. If the HTML page includes links, it is possible to navigate through several pages. Eventually, when a SEND HTML FILE("") is issued, the HTML mode is exited.

(12) The object method of the button that was clicked and the selection display initiated by DISPLAY SELECTION are executed. Note that steps (10) and (11) can be repeated several times during navigation of the selection.

(13) Finally, the selection display is exited and the Web Connection process is executed.

And so on...

Free Web navigation (clicking on the Back or Forward buttons for instance) is possible within any SEND HTML FILE (green areas in the figure above). On the other hand, any 4D-based HTML page (data entry, selection display... including standard dialog boxes such those displayed by CONFIRM or Request) is exited through the use of one of the browser navigation controls, 4D will eventually synchronize the Web sessions and the Database sessions by going back to the Web page whose subcontext ID corresponds to that of the issued 4D command currently being executed on the Web Connection process side.

Web Connection process and Web session

From the user viewpoint, the user's actions on the Web browser side pilot a Web session. From the programmatic viewpoint, the Web Connection process pilots the Web session, not the reverse. The Web browser displays the pages sent by the Web Connection process, which either:

• Executes 4D code, or

• Waits for the submission from the browser of the current Web page.

From a Design viewpoint, the Web Connection process should be seen as a 4D process whose domain of execution is 4th Dimension or 4D Server, but whose user interface is remotely echoed on the connected Web browser.

With this in mind, always take into account this duality of the Web Connection process when designing Web database applications. For example:

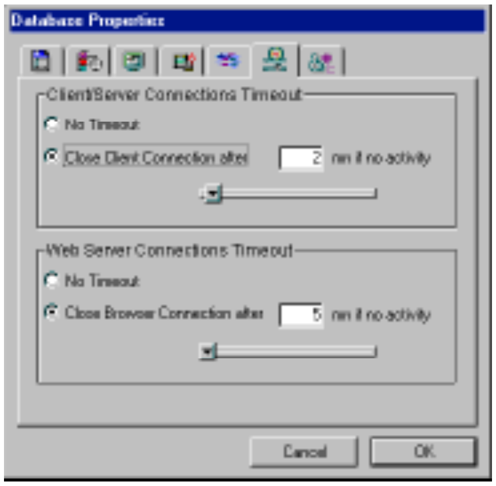
• During data entry of any kind, the main menu bar is that of the browser, not that of 4D. Within a form, do not rely on the 4D menu bar; it is on the Web server machine, not on the Web browser machine,

• When you design forms to be used on the Web browser, remember that the 4D form set of features is limited to that of HTML (but sometimes with some 4D additions). Do not rely on the whole 4D forms feature set (i.e., object types and form events). For detailed information, see the section Web Services, HTML and Javascript Encapsulation.

• In terms of interprocess communication, CALL PROCESS, when applied to a Web Connection process, has no effects because its current active form is displayed on the Web browser. On the other hand, a Web Connection process can issue a CALL PROCESS toward another 4D process.
In addition, interprocess communication can be indifferently performed in both directions using the GET PROCESS VARIABLE and SET PROCESS VARIABLE commands, which do not require a process to have a user interface.

Web Connection Timeout

As explained previously, a Web Connection process is either executing 4D code or waiting for the submission of the Web page currently displayed on the browser side. In the latter case, a Web Connection process will wait for a delay greater than or equal to the **Web Server Connections Timeout,** set in the Database Properties window (shown below) or set programmatically using the SET WEB TIMEOUT command.



Client/Server Connections Timeout
No Timeout
[✓ Close Client Connection after] 2 mn if no activity

Web Server Connections Timeout
No Timeout
[✓ Close Browser Connection after] 5 mn if no activity

The scope of the Web Server Connections Timeout setting is the database session. All Web Connection processes are subjected to that value; they are immediately affected if that setting is changed. The default value is 5 minutes. You can increase or decrease this timeout at your convenience. For example, you can increase the timeout if your application allows Web users to surf to other Web sites via HTML links in the pages served by your database. By increasing the timeout, you enable users to navigate longer within the other Web sites without closing their connections to your databases.

WARNING: In Version 6.0, there is no way to programmatically stop a Web Connection process. If you specify a long timeout, the process will wait for that delay, even though the Web user may have stopped working with the Web Connection for quite some time. If you specify No Timeout, the Web Connection processes will stop only when the database is exited.

Tip: When developing and testing a Web database, you may reach the maximum number of Web Connections allowed by the Web Licenses present in your 4th Dimension or 4D Server. For example, this can happen while debugging the On Web Connection Database Method, in which you reconnect several times. Turning off the Web Server will not abort the Web Connection processes still waiting for the completion of the timeout delay. However, unlike Web Server process, Web Connection processes can be aborted using the Abort command from the Process menu (available in the Design environment when the Process List window is the frontmost window).

On Web Connection Database Method

The On Web Connection database method is automatically called by 4th Dimension or 4D Server each time a Web browser initiates a connection to the database. This happens only if the database is published as a Web server.

The On Web Connection database method receives two text parameters that are passed by 4D. You can declare these two parameters as follows:

```
   ` On Web Connection Database Method
C_TEXT($1;$2)
   ` Code for the method
```

URL extra data

The first parameter ($1) is the URL entered by the user in the location area of his or her Web browser, from which the host address has been removed.

Let's take the example of an Intranet connection. Suppose that the IP address of your 4D Web Server machine is 123.4.567.89. The following table shows the values of $1 depending on the URL entered in the Web browser:

URL entered in Web browser Location area	Value of parameter $1
123.4.567.89	/
http://123.4.567.89	/
123.4.567.89/Customers	/Customers
http://123.4.567.89/Customers	/Customers
http://123.4.567.89/Customers/Add	/Customers/Add
123.4.567.89/Do_This/If_OK/Do_That	/Do_This/If_OK/Do_That

Note that you are free to use this parameter at your convenience. 4D simply ignores the value passed beyond the host part of the URL.

For example, you can establish a convention where the value "/Customers/Add" means "go directly to add a new record in the [Customers] table." By supplying the Web users of your database with a list of possible values and/or default bookmarks, you can provide shortcuts to the different parts of your application. This way, Web users can quickly access resources of your Web site without going through the whole navigation path each time they make a new connection to your database.

WARNING: In order to prevent a user from reentering a database with a bookmark created during a previous session, 4D intercepts any URL that corresponds to one of the standard 4D URLs.

Header of the HTTP request

The second parameter ($2) is the header of the HTTP request sent by the Web browser. Note that this header is passed to your On Web Connection database method as it is. Its contents will vary depending on the nature of the Web browser which is attempting the connection.

With Netscape 3.0 running on Windows NT, you may receive a header similar to this:

```
GET HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.01 (WinNT; I)
Host: 192.9.200.11
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

With Microsoft Internet Explorer running on Windows NT, you may receive a header similar to this:

```
GET / HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
Accept-Language: en
User-Agent: Mozilla/1.22 (compatible; MSIE 2.0d; Windows NT)
Connection: Keep-Alive
If-Modified-Since: Sunday, 10-Dec-96 01:51:37 GMT
```

If your application deals with this information, it is up to you to parse the header.

Example: Implementing Client Local Home Pages

In the following example, the parameter $1, sent to the On Web Connection database method, is used to implement Client Home Pages within an organization.

The database has two tables: [Customers] and [Tables]. The On Startup database method shown here initializes interprocess arrays used later by the On Web Connection database method.

```
    ` On Startup Database Method

    ` Table List
ARRAY STRING(31;<>asTables;Count tables)
For ($vlTable;1;Size of array(<>asTables))
    <>asTables{$vlTable}:=Table name($vlTable)
End for
```

```
          ` Standard Web Actions at Login
     ARRAY STRING(31;<>asActions;2)
     <>asActions{1}:="Add"
     <>asActions{2}:="List"
```

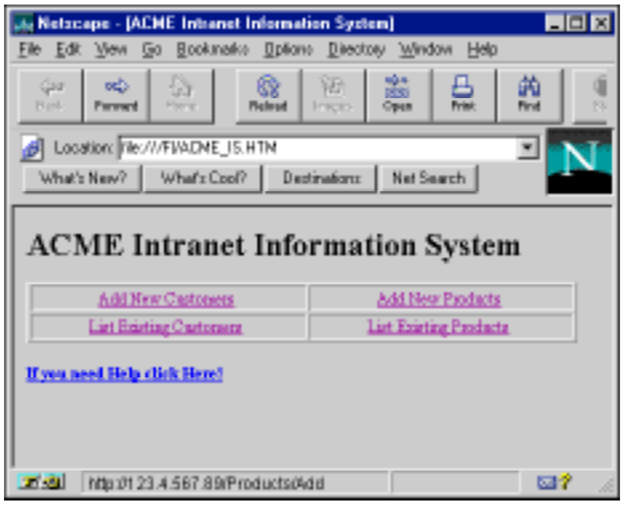
The main job of the On Web Connection database method is to decipher the extra data passed in the URL after the host part of the address and to act accordingly. The method is as follows:

```
          ` On Web Connection Database Method

     C_TEXT($1;$2)
     C_TEXT($vtURL)

          ` Just in case, check that $1 is equal to "/" or "/..."
     If ($1="/@")
               ` Copy the URL into a local variable minus the first "/"
          $vtURL:=Substring($1;2)
               ` Parse the URL and populate a local array with the tokens of the URL
               ` For example, if the URL extra data is "aaa/bbb/ccc", the resulting array
               ` will be of the three elements "aaa", "bbb" and "ccc" in that order
          $vlElem:=0
          ARRAY TEXT($atTokens;$vlElem)
          While ($vtURL # "")
               $vlElem:=$vlElem+1
               INSERT ELEMENT($atTokens;$vlElem)
               $vlPos:=Position("/";$vtURL)
               If ($vlPos>0)
                    $atTokens{$vlElem}:=Substring($vtURL;1;$vlPos-1)
                    $vtURL:=Substring($vtURL;$vlPos+1)
               Else
                    $atTokens{$vlElem}:=$vtURL
                    $vtURL:=""
               End if
          End while
               ` If extra data was passed after the HOST part of the URL
          If ($vlElem>0)
                    ` Using the interprocess array initialized in the On Startup DB method
                    ` Check whether the 1st token is a name of a table
               $vlTableNumber:=Find in array(<>asTables;$atTokens{1})
               If ($vlTableNumber>0)
                         ` If so, get pointer to this table
                    $vpTable:=Table($vlTableNumber)
                         ` Set the Input and Output forms
                    INPUT FORM($vpTable->;"Input Web")
                    OUTPUT FORM($vpTable->;"Output Web")
```

```
                ` Using an interprocess array initialized in the On Startup DB Method
                ` Check whether the 2nd token is a known standard action
            $vlAction:=Find in array(<>asActions;$atTokens{2})
            Case of
                    ` Adding records
                : ($vlAction=1)
                    Repeat
                        ADD RECORD($vpTable->;*)
                    Until (OK=0)
                    ` Listing records
                : ($vlAction=2)
                    READ ONLY($vpTable->)
                    ALL RECORDS($vpTable->)
                    DISPLAY SELECTION($vpTable->;*)
                    READ WRITE($vpTable->)
                Else
                    ` Here could additional standard table actions be implemented
            End case
        Else
            ` Here could other standard actions be implemented
        End if
    End if
End if
    ` Whatever happened above, pursue with the normal Log On process
WWW NORMAL LOG ON
```

At this point, people within the organization can connect to the database and enter a URL according to the convention set by the methods listed. Users can also create bookmarks if they do not want to re-enter the URL each time. In fact, the ultimate solution is to provide each member of the organization with an HTML page that they will use locally to access the database.

This HTML page is shown here:


Netscape - [ACME Intranet Information System]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find
Location: file:///F|/ACME_IS.HTM
What's New? What's Cool? Destinations Net Search
ACME Intranet Information System
Add New Customers
Add New Products
List Existing Customers
List Existing Products
If you need Help click Here!
http://123.4.567.89/Products/Add

In other words, the HTML page ACME_IS.HTM is the Client Local Home Page for the 4D-based information system of the organization. If a user clicks on the Add New Products link, the Web browser will connect to the host having the URL http://123.4.567.89/Products/Add. Provided that the IP address of the database computer is 123.4.567.89, the On Web Connection database method receives the extra URL data "/Products/Add" in $1, and therefore proceeds to add records in the [Products] table.

Finally, users can drag and drop links from that page onto the desktop to create Internet Shortcut icons, such as the Add New Customers icon shown here. Simply double-clicking these icons will bring them directly into any part of your 4D Web database.

Add New
Customers

The source code of this HTML page is listed here:

```
<HTML>
<HEAD>
 <TITLE>ACME Intranet Information System</TITLE>
</HEAD>
<BODY>
<H1><B>ACME Intranet Information System</B></H1>
<P ALIGN=CENTER><TABLE BORDER=1 CELLPADDING=1 WIDTH="100%">
 <TR>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Customers/Add">Add New Customers</
 </TD>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Products/Add">Add New Products</A>
 </TD>
 </TR>
 <TR>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Customers/List">List Existing Customers
 </TD>
 <TD>
  <P ALIGN=CENTER><A HREF="http://123.4.567.89/Products/List">List Existing Products</
 </TD>
 </TR>
</TABLE></P>
<P><B><A HREF="Help.HTM">If you need Help click Here!</A></B></P>
</BODY>
</HTML>
```

See Also

Database Methods.

Web Services, HTML Support

Web Server

version 6.0

Before implementing HTML and JavaScript code into your 4D application, it is a good idea to review what 4th Dimension already does for you.

Menu Bars

• Each menu bar is translated into one HTML page. Each menu title appears as text only and menu items appear as links to 4D methods.
• Clicking a menu item on the Web Browser side starts the execution of the associated 4D method on the Web Connection process side.

Forms

• Objects are translated from top to bottom and from left to right. Note, however, that HTML is a word processing oriented application; horizontal objects positions will be different and wrap-around may occur.
• Multi-page forms are supported transparently. Note, however, that a page zero is not supported; consequently, page navigation buttons must be present on each page.
• Automatic actions, when appropriate, are supported transparently.
• Form events (On Load, On Unload, On Clicked) are supported. Other events are not supported.
• The Header, Detail, Break and Footer tags are taken into account during calls to DISPLAY SELECTION and MODIFY SELECTION. The Header of the form appears once at the beginning of the HTML page, the detail area is repeated as many times as necessary, and variables (such as buttons) placed in the Footer area appear at the end of the HTML page, just under the automatic selection page navigation links.

Fields Objects

When a 4D form is translated to an HTML page, field objects are translated as follows:

4D Field Type	HTML Object	HTML Markup
Alphanumeric	Text field (*)	<INPUT Type="text" ...>
Text	Text field (*)	<TEXTAREA ...> (**)
		<INPUT Type="text" ...> (***)
Real	Text field (*)	<INPUT Type="text" ...>
Integer	Text field (*)	<INPUT Type="text" ...>
Long Integer	Text field (*)	<INPUT Type="text" ...>
Date	Text field (*)	<INPUT Type="text" ...>
Time	Text field (*)	<INPUT Type="text" ...>
Boolean	Radio or Check box (*)	<INPUT Type="radio" ...>
		<INPUT Type="checkbox" ...>
Picture	Image (always non-enterable)	<IMG SRC="..." ...>
Subtable	No HTML support	None
BLOB	No HTML support	None

(*) or text only if non enterable
(**) If the text value is composed of several lines
(***) If the text value is composed of only one line or is empty

Note: Enterable variables behave like fields of the same type.

Form Objects

When a 4D form is translated to an HTML page, form objects are translated as follows:

4D Form Object	Equivalent HTML Object	HTML Markup
Line	Horizontal Line (1)	<HR>
Rectangle	No HTML support	None
Oval	No HTML support	None
Rounded Rectangle	No HTML support	None
Static Picture	Image or Image Map (2)	<IMG SRC="..."> <INPUT Type="image" ...>
Group Box	Text	Text with font markups if any
Static Text	Text	Text with font markups if any
Button	Submit button	<INPUT Type="submit" ...>
Default Button	Submit button	<INPUT Type="submit" ...>
Radio Button	Radio button (3)	<INPUT Type="radio" ...>
Check Box	Check Box	<INPUT Type="checkbox" ...>
Popup menu	Drop-down List	<SELECT ...>...</SELECT>
Drop-down List	Drop-down List	<SELECT ...>...</SELECT>
Menu/Drop-down List		Drop-down List <SELECT ...>...</SELECT>
Combo Box	Drop-down List	<SELECT ...>...</SELECT>
Scrollable Area	Scrollable List (4)	<SELECT ...>...</SELECT>
Invisible Button	See note 2	
Highlight Button	See note 2	
3D Button	See note 2	
Button Grid	See note 2	
Graph	Image (non-enterable)	<IMG SRC="..." ...>
Plug-in	Image (non-enterable)	<IMG SRC="..." ...>

The following objects are not supported by HTML and therefore are ignored during the translation:

Hierarchical Popup menu, Hierarchical List, Subform, Tab Control, Radio Picture, Thermometer, Ruler, Dial, Picture Menu, Picture Button, 3D Check Box, 3D Radio Button.

Notes

1. Non-horizontal lines are not supported in HTML; they are therefore ignored.

2. Invisible-like buttons are objects of type Invisible Button, Highlight Button, 3D Button, and Button Grid. If a static picture is not overlapped by an invisible-like button, the picture is translated as a static image. If it is overlapped by at least one invisible-like button, it is translated as a Server-Side Image Map. On the Web browser side, the image is treated as a Server-Side Image Map. On the 4D side, when the submission is received, 4D recalculates the position of the click in order to generate an On Clicked event for the appropriate button, as if the button was actually clicked. Managing invisible-like buttons is therefore quite simple, provided that they overlap with static pictures. You manage these buttons through the Form method or their object methods, as you would do in the regular 4D interface. This also provides you with a very simple way to handle Web Image Mapping. If an invisible-like button does not overlap with any static picture objects, it is ignored during the translation.

3. Radio button grouping is maintained though the translation.

4. Grouped scrollable areas are not supported in HTML. 4D translates them as independent scrollable lists located on the same line.

Display Selection / Modify Selection

• The UserSet mechanism is not supported
• An automatic selection paging mechanism is provided by 4D. For more information, see the description of the SET WEB DISPLAY LIMITS command.

4D Commands

While developing a 4D Web database, you may ask what happens when this or that command is called. Will the command take effect on the Web Server machine or on the Web Browser machine? The Web Connection Process is executing on the Web Server machine, but its user interface is remotely echoed on the connected Web Browser. Consequently, for Web database development, the 4D commands can be classified as follows:

Commands that are not affected by execution from within a Web Connection process

A command such as CREATE RECORD works within the executing process; in this case, it creates a record within the Web Connection process. The same applies to commands such as Screen width, which returns the width of the screen on the Web Server machine (the machine on which the process is executing).

Commands that include extra built-in capabilities for transparent Web support

Command Name	**Comments**
ADD RECORD	Automatic translation of the form, multi-page forms supported
ALERT	Automatic translation of the dialog box
CONFIRM	Automatic translation of the dialog box
DIALOG	Automatic translation of the form, multi-page forms supported
DISPLAY SELECTION	Automatic translation of the form Built-in Web paging mechanism UserSet mechanism is not supported
MODIFY RECORD	Automatic translation of the form, multi-page forms supported
MODIFY SELECTION	Automatic translation of the form Built-in Web paging mechanism UserSet mechanism is not supported
QUERY	Standard Query dialog box supported
QUERY BY EXAMPLE	Automatic translation of the form, multi-page forms supported
Request	Automatic translation of the dialog box

Command to use when you know what you want to do

The following commands execute locally on the Web Server machine.

For example, you can invoke the printing of a selection from a Web Browser. However, the printing will be performed on the Web Server machine.

In addition, when a user interface component is involved, it appears on the Web Server machine, i.e., Open document("") vs Open Document("This document"). You should avoid such calls, because the Web Browser will wait for a reply until the dialog box is closed on the Web Server machine. On the other hand, it is perfectly OK to call these routines when no dialog boxes are involved.

Command Name	**Comments**
Append document	OK, if no file dialog box is invoked
BEEP	Beeps on Web Server machine
Create document	OK, if no file dialog box is invoked
DISPLAY RECORD	Does nothing
EXPORT DIF	OK, if no file dialog box is invoked
EXPORT SYLK	OK, if no file dialog box is invoked
EXPORT TEXT	OK, if no file dialog box is invoked
IMPORT DIF	OK, if no file dialog box is invoked
IMPORT SYLK	OK, if no file dialog box is invoked
IMPORT TEXT	OK, if no file dialog box is invoked

LOAD SET	OK, if no file dialog box is invoked
LOAD VARIABLES	OK, if no file dialog box is invoked
MESSAGE	Messages will appear on Web Server machine
Open document	OK, if no file dialog box is invoked
Open external window	Window opens on Web Server machine
Open resource file	OK, if no file dialog box is invoked
Open window	Window opens on Web Server machine
PLAY	Sound is played on 4D machine
PRINT FORM	OK, if no Printing dialog box is invoked
PRINT LABELS	OK, if no Printing dialog box is invoked
PRINT RECORD	OK, if no Printing dialog box is invoked
PRINT SELECTION	OK, if no Printing dialog box is invoked
QUIT 4D	Supported, you can shutdown the Web server remotely
SAVE SET	OK, if no file dialog box is invoked
SAVE VARIABLES	OK, if no file dialog box is invoked
SELECT LOG FILE	OK, if no file dialog box is invoked
SET CHANNEL	OK, if no file dialog box is invoked (documents)
TRACE	Debugger window appears on Web Server machine

Command Not Supported by Web Connection Processes

Command Name	**Comments**
ADD DATA SEGMENT	Do NOT call this command from within a Web Connection process This command has not yet been designed to be used on the Web
ADD SUBRECORD	Do NOT call this command from within a Web Connection process This command has not yet been designed to be used on the Web
CHANGE ACCESS	Do NOT call this command from within a Web Connection process This command has not yet been designed to be used on the Web
EDIT ACCESS	Do NOT call this command from within a Web Connection process The Passwords window appears on the 4D machine The Browser will wait until the window is closed
GRAPH TABLE	Do NOT call this command from within a Web Connection process This command has not yet been designed to be used on the Web
MODIFY SUBRECORD	Do NOT call this command from within a Web Connection process This command has not yet been designed to be used on the Web
ORDER BY	Programmatical support only Standard Order By dialog box not supported on the Web yet

PRINT SETTINGS	Do NOT call this command from within a Web Connection process The Printing dialog boxes will appear on the 4D machine The Browser will wait until the dialog boxes are closed
REPORT	Do NOT call this command from within a Web Connection process The Quick Report window appears on the 4D machine The Browser will wait until the window is closed

Web Services, HTML and Javascript Encapsulation

Preliminary Note

Before working with this section, read the following sections on Web Services:

• Web Services, Overview
• Web Services, Configuration
• Web Services, Your First time (Part 1)
• Web Services, Your First time (Part 2)
• Web Services, Web Connection Processes
• Web Services, HTML Support

There are three ways in which you can encapsulate HTML code into your 4D application:

1. Using the command SEND HTML FILE, you can send a Web page stored on disk.
2. A 4D form static text object, such as "{anyPage.HTM}", inserts the HTML document "anyPage.HTM" into the 4D form at the location of the static text object.
3. Any 4D text variable in a form can encapsulate HTML code into a 4D form, provided its first character is ASCII code 1 (i.e., vtHTML:=Char(1)+"...HTML code...").

In the last two cases, the resulting form on the Web brower side is the combination of the 4D and HTML objects. Note that when using a static text object, you insert a document in its entirety. When using a text variable, you insert pieces of code.

Using SEND HTML FILE or form static text object, you can either use an existing HTML document or refer to a document that you have programmatically built and then saved on disk. You can build the HTML code in memory, using a text variable located in a form.

Binding HTML Objects with 4D Methods

No matter how you encapsulate HTML in your 4D application, you can link an HTML object to a 4D Method by creating a link for that object. The URL of the object must be /4DMETHOD/Method_Name, where Method_Name is the name of the 4D project method to be executed when the HTML object is clicked. Examples of 4D methods bound to HTML objects are provided in the section Web Services, Your First time (Part 2).

Important

• If you send an HTML file, you can bind 4D methods with any type of linkable HTML object. Remember, in order to have a POST action 4D Method that will issue a SEND HTML FILE("") call that stops the HTML mode, you must have a Submit button in your HTML page to execute the POST action of the HTML page (the page being submitted to 4D Web Server).
• On the other hand, if you encapsulate HTML code into a 4D form as static text or as a text variable, you cannot use Submit buttons. You can have links referring to 4D methods, but you do not have the opportunity to specify the POST action because 4D builds the page for you.

Binding HTML Objects with 4D Variables - Part 1

You can bind HTML objects with 4D variables.

Note: You work with process variables.

First, an HTML object can have its value initialized using the value of a 4D variable.

Second, after a Web page is submitted back, the value of an HTML object can be returned into a 4D variable. To do so, within the HTML source of the page, you create an HTML object whose name is the same as the name of the 4D process variable you want to bind. That point is studied further in the section “Binding HTML Objects with 4D Variables - Part 2” in this document.

Since an HTML object value can be initialized with the value of a 4D variable, you can programmatically provide default values to HTML objects by including [VarName] in the **value** field of the HTML object, where VarName is the name of the 4D process variable as defined in the Web Connection process. This is the name that you surround with the square brackets [ ].

In fact, the syntax [VarName] allows you to insert 4D data anywhere in the HTML page. For example, if you write:

```
<P>Welcome to [vtSiteName]!</P>
```

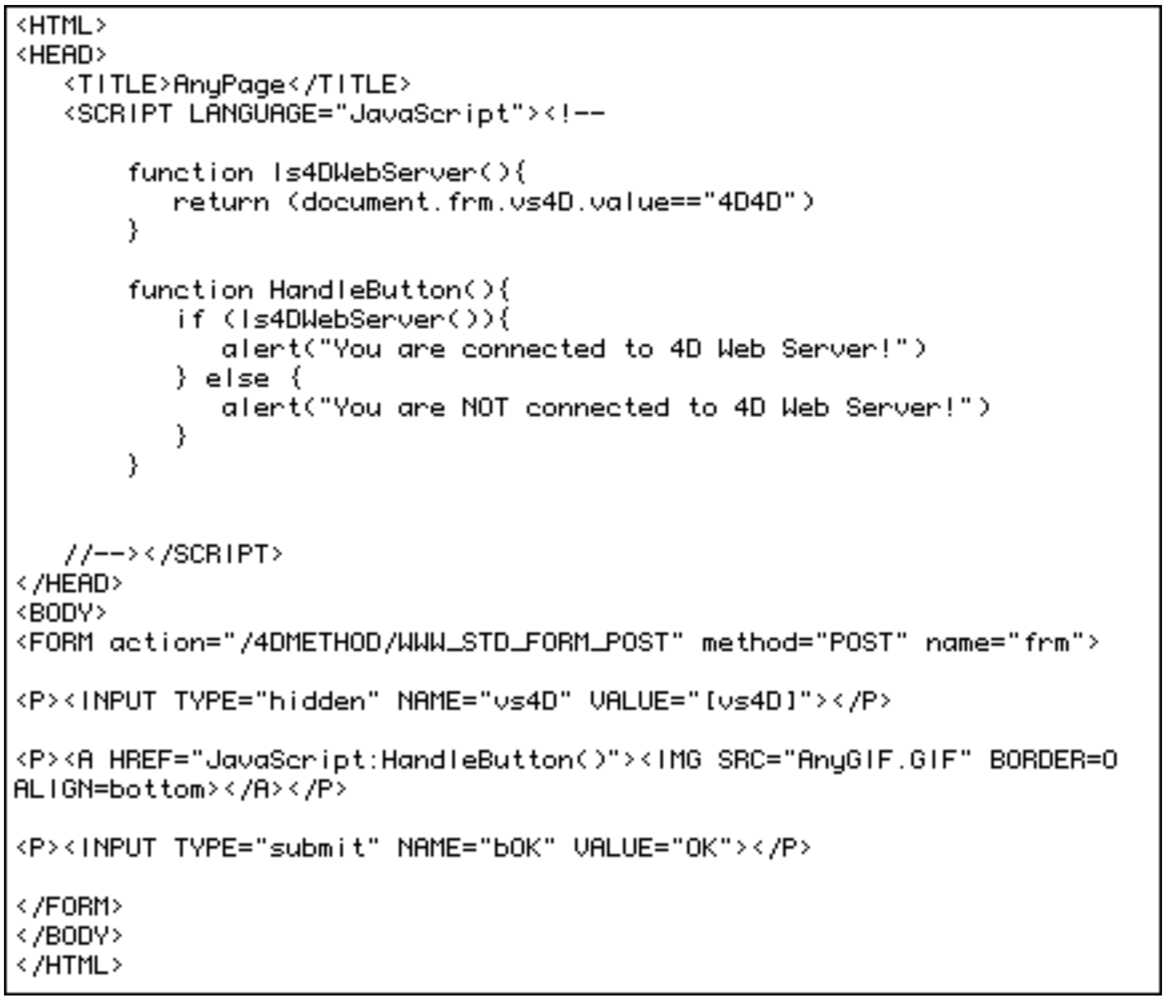
The value of the 4D variable vtSiteName will be inserted in the HTML page.

Tip: The bounding using the syntax [VarName] is treated recursively. If the text value in the variable VarName includes other valid [...] references, 4D will replace them with the values of the variables until no reference is found.

Here is an example:

```
    ` The following piece of 4D code assigns "4D4D" to the process variable vs4D
vs4D:="4D4D"
    ` Then it send the HTML page "AnyPage.HTM"
SEND HTML FILE("AnyPage.HTM")
```

The source of the HTML page AnyPage.HTM is listed here:



```
<HTML>
<HEAD>
   <TITLE>AnyPage</TITLE>
   <SCRIPT LANGUAGE="JavaScript"><!--

       function Is4DWebServer(){
          return (document.frm.vs4D.value=="4D4D")
       }

       function HandleButton(){
          if (Is4DWebServer()){
             alert("You are connected to 4D Web Server!")
          } else {
             alert("You are NOT connected to 4D Web Server!")
          }
       }


   //--></SCRIPT>
</HEAD>
<BODY>
<FORM action="/4DMETHOD/WWW_STD_FORM_POST" method="POST" name="frm">

<P><INPUT TYPE="hidden" NAME="vs4D" VALUE="[vs4D]"></P>

<P><A HREF="JavaScript:HandleButton()"><IMG SRC="AnyGIF.GIF" BORDER=0
ALIGN=bottom></A></P>

<P><INPUT TYPE="submit" NAME="bOK" VALUE="OK"></P>

</FORM>
</BODY>
</HTML>
```

Before sending an HTML page (HTML document or translated 4D form), 4D always parses the HTML source code in order to look for objects referring to 4D variables and to remap the URLs of the links (as we will see later).

In the HTML source code shown, note the **hidden** input object named vs4D. The value of this object is set to the text value "[vs4D]". Since the project method sending the HTML file has previously defined the 4D process variable vs4D, 4D replaces the value of the HTML object and sets it to "4D4D", the value of the 4D variable.

The embedded JavaScript function Is4DWebServer tests the value of the vs4D HTML object. Here is the trick: if the HTML page is served by 4D, the object's value is changed to "4D4D". However, if the HTML page is served by another application (i.e., Web Star on Macintosh), the object stays with its value as defined in the page, "[vs4D]". Bingo! By using JavaScript to test the value of that object, from within the page on the Web Browser side, you can detect whether or not the page is being served by 4D.

This first example shows how you can build "intelligent" HTML pages that provide additional features when being served by 4D, while staying compatible with other Web servers.

Important: You bind process variables only. In addition, the initial version 6.0 release doesnot allow you to bind a 4D array to an HTML SELECT object. On the other hand, each element of a SELECT object can refer to separate 4D variables (i.e., the first element to V1, the second to V2, and so on).

The binding in the direction 4D toward Web Browser works with any encapsulation method (SEND HTML FILE, static text or text variable in a 4D form).

JavaScript Encapsulation

4D supports JavaScript source code embedded into HTML documents. However, the initial 6.0 release does not support the insertion of JavaScript .js files into HTML documents (i.e., <SCRIPT SRC="...").

Using SEND HTML FILE, you send a page that you have prepared in an HTML source editor or built programmatically using 4D and saved as a document on disk. In both cases, you have full control of the page. You can insert JavaScript scripts in the HEAD section of the document as well as use scripts with the FORM markup. In the previous example, the script refers to the form "frm" because you were able to name the form. You can also trigger, accept, or reject the submission of the form at the FORM markup level.

If you you encapsulate HTML in a 4D form, you do not have control over the HEAD section or the FORM declaration. The scope of the scripts is therefore different. For example, you cannot access the HTML form by its name.

However, compare the Is4DWebServer JavaScript function of the previous example with this one:

```
function Is4DWebServer(){
   return (document.forms[0].vs4D.value=="4D4D")
}
```

Both functions do the same thing, but the second example uses the forms property of the HTML document object to access the object through the element forms[0]. As a result, it operates even if you do not know the name that 4D may or may have not given to the translated HTML page (form).

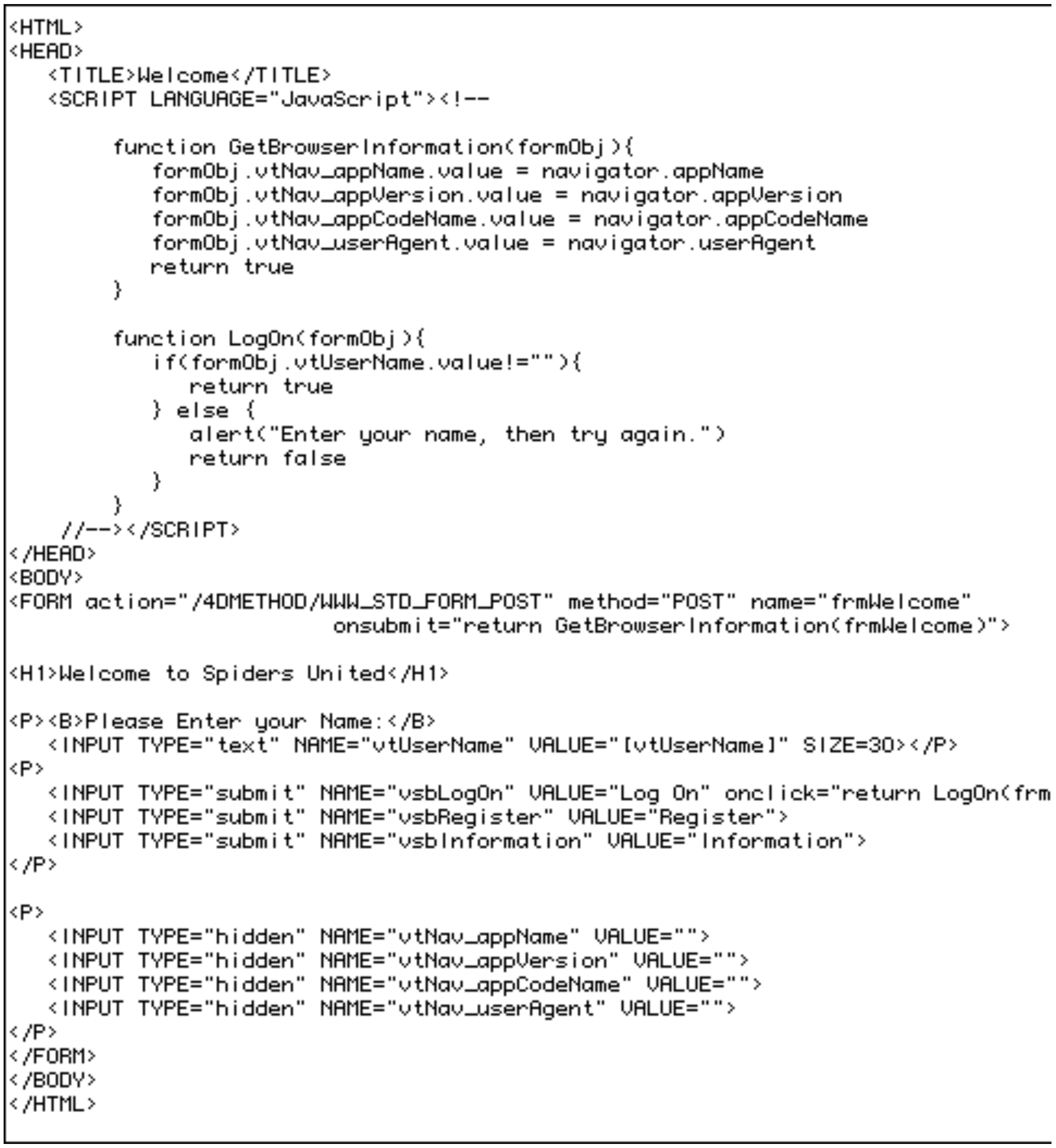
Note: The initial 6.0 release does not provide built-in support for Java applets. However, while this documentation is being written, we learned that ACI Partners are currently developing 4D Plug-ins that integrate Java applets with the 4D environment.

Binding HTML Objects with 4D Variables - Part 2

When you send an HTML page using SEND HTML FILE, you can also bind 4D variables with HTML objects in the “Web Browser toward 4D” direction. The binding works both ways: once the HTML page is submitted, 4D copies back the values of the HTML objects into the 4D process variables.

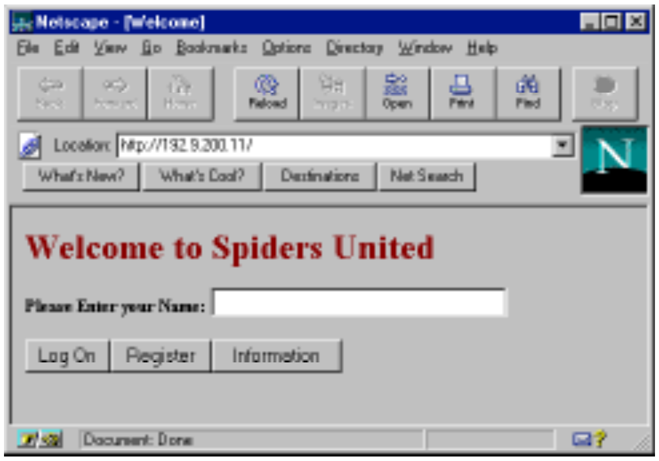
Warning: Getting the values back into the 4D process variables is only possible with HTML pages sent using SEND HTML FILE. With HTML encapsulated in a 4D form, getting back values is restricted to the actual 4D objects located in the form.

Consider the following HTML page source code:



```
<HTML>
<HEAD>
   <TITLE>Welcome</TITLE>
   <SCRIPT LANGUAGE="JavaScript"><!--

        function GetBrowserInformation(formObj){
           formObj.vtNav_appName.value = navigator.appName
           formObj.vtNav_appVersion.value = navigator.appVersion
           formObj.vtNav_appCodeName.value = navigator.appCodeName
           formObj.vtNav_userAgent.value = navigator.userAgent
           return true
        }

        function LogOn(formObj){
           if(formObj.vtUserName.value!=""){
              return true
           } else {
              alert("Enter your name, then try again.")
              return false
           }
        }
    //--></SCRIPT>
</HEAD>
<BODY>
<FORM action="/4DMETHOD/WWW_STD_FORM_POST" method="POST" name="frmWelcome"
                         onsubmit="return GetBrowserInformation(frmWelcome)">

<H1>Welcome to Spiders United</H1>

<P><B>Please Enter your Name:</B>
   <INPUT TYPE="text" NAME="vtUserName" VALUE="[vtUserName]" SIZE=30></P>
<P>
   <INPUT TYPE="submit" NAME="vsbLogOn" VALUE="Log On" onclick="return LogOn(frm
   <INPUT TYPE="submit" NAME="vsbRegister" VALUE="Register">
   <INPUT TYPE="submit" NAME="vsbInformation" VALUE="Information">
</P>

<P>
   <INPUT TYPE="hidden" NAME="vtNav_appName" VALUE="">
   <INPUT TYPE="hidden" NAME="vtNav_appVersion" VALUE="">
   <INPUT TYPE="hidden" NAME="vtNav_appCodeName" VALUE="">
   <INPUT TYPE="hidden" NAME="vtNav_userAgent" VALUE="">
</P>
</FORM>
</BODY>
</HTML>
```

When 4D sends the page to a Web Browser, it looks like this:


Netscape - [Welcome]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Location: http://192.9.200.11/
What's New? What's Cool? Destinations Net Search
Welcome to Spiders United

Please Enter your Name:

Log On Register Information
Document: Done

The main features of this page are:

• It includes three Submit buttons: vsbLogOn, vsbRegister and vsbInformation.
• When you click Log On, the submission of the form is first processed by the JavaScript function LogOn. If no name is entered, the form is not even submitted to 4D, and a JavaScript alert is displayed.
• The form has a POST 4D Method as well as a Submit script (GetBrowserInformation) that copies the Navigator properties to the four hidden objects whose names starts with vtNav_App.
• Theinitial value of the object vtUserName is [vtUserName].

Let's examine the 4D method WWW Welcome that sends this HTML page using the SEND HTML FILE command. This method is called by the On Web Connection Database Method.

```
    ` WWW Welcome Project Method
    ` WWW Welcome -> Boolean
    ` WWW Welcome -> Yes = Can start a session

C_BOOLEAN($0)
$0:=False

    ` Hidden INPUT HTML objects returning Browser information
C_TEXT(vtNav_appName;vtNav_appVersion;vtNav_appCodeName;vtNav_userAgent)
vtNav_appName:=""
vtNav_appVersion:=""
vtNav_appCodeName:=""
vtNav_userAgent:=""
```

```
    ` Text INPUT HTML object where the user name is entered
C_TEXT(vtUserName)
vtUserName:=""

    ` HTML submit button values
C_STRING(31;vsbLogOn;vsbRegister;vsbInformation)

Repeat
        ` Do not forget to reset the values of the submit buttons!
    vsbLogOn:=""
    vsbRegister:=""
    vsbInformation:=""
        ` Send the Web page
    SEND HTML FILE("Welcome.HTM")
        ` Test the values of the submit buttons in order to detect which one was clicked
    Case of

            ` The Log On button was clicked
        : (vsbLogOn # "")
            QUERY([WWW Users];[WWW Users]User Name=vtUserName)
            $0:=(Records in selection([WWW Users])>0)
            If ($0)
                WWW POST EVENT ("Log On";WWW Log information )
                    ` The method WWW POST EVENT saves information
                    ` to a table of the database
            Else
                CONFIRM("This User Name is unknown, would you like to register?")
                $0:=(OK=1)
                If ($0)
                    $0:=WWW Register
                        ` The method WWW Register allow a new Web User to register
                End if
            End if

            ` The Register button was clicked
        : (vsbRegister # "")
            $0:=WWW Register

                ` The Information button was clicked
        : (vsbInformation # "")
            DIALOG([User Interface];"WWW Information")
    End case
Until (Not(<>vbWebServicesOn) | $0)
```

The features of this method are:

• The 4D variables vtNav_appName, vtNav_appVersion, vtNav_appCodeName, and vtNav_userAgent (bound to the HTML objects having the same names) use the GetBrowserInformation JavaScript script to get back the values assigned to the HTML objects. Simple and direct, the method initializes the variables as strings, then gets back the values after the Web page has been submitted.

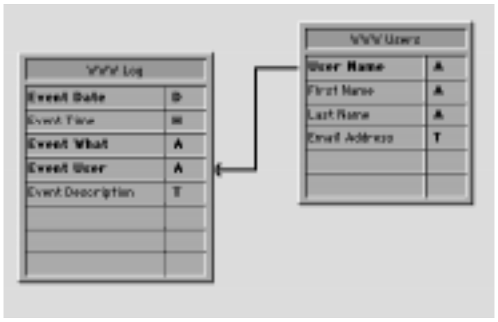
• The 4D variables vsbLogOn, vsbRegister and vsbInformation are bound to the three Submit buttons. Note that these variables are reset each time the page is sent to the browser. When the submit is performed by one of these buttons, the browser returns the value of the clicked button to 4D. The 4D variables are reset each time, so the variable that is no longer equal to the empty string tells you which button was clicked. The two other variables are empty strings, not because the browser returned empty strings, but because the browser "said" nothing about them; consequently, 4D left the variables unchanged. That is why it is necessary to reset those variables to the empty string each time the page is sent to the browser.

This the way to distinguish which Submit button was clicked when several Submit buttons exist on the Web page. Note that 4D buttons in a 4D form are numeric variables. However, with HTML, all objects are text objects. Upon return to 4D, testing if a button was clicked consists of testing the text value of the 4D variable bound to the button.

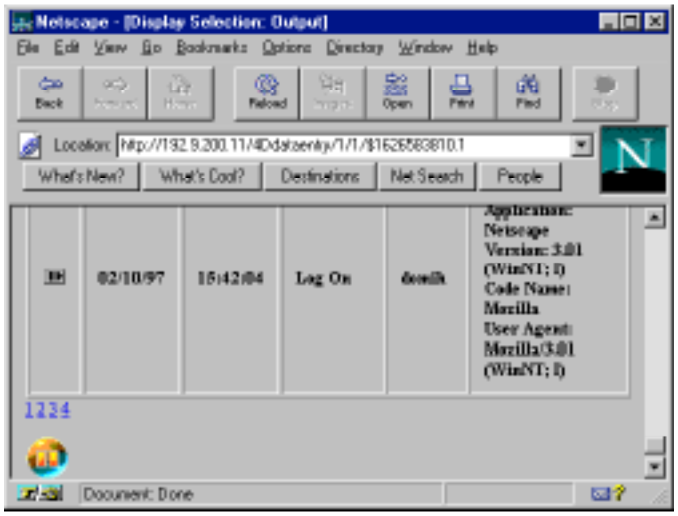
If you bind a 4D variable with a SELECT object, you also bind a text variable. In 4D, to test which element of a drop-down list was chosen, you test the numeric value of the 4D array. With HTML, this is the value of the selected item that is returned in the 4D variable bound to the HTML object.

No matter which object you bind with a 4D variable, the returned value is of type Text, so you bind String or Text 4D process variables.

An interesting point of this example is that after you have obtained information about the Browser, you can store these values in a 4D table, again combining Web and database capabilities. This is what the (unlisted) WWW POST EVENT project method does. It does not "post an event"; it saves the web session information into the tables shown here:



After you have saved the information in a table, you can use other project methods to send the information back to the Web user. To do so, simply use QUERY to find the right information and then use DISPLAY SELECTION to show the [WWW Log] records. The following figure shows the log information available to the registered user of the Web site:


Netscape - [Display Selection: Output]
File Edit View Go Bookmarks Options Directory Window Help
Back Reload Open Print Find
Location: http://192.9.200.11/4Ddataentry/1/1/$1626583810.1
What's New? What's Cool? Destinations Net Search People
1234
Document: Done

[illegible]	02/10/97	15:42:04	Log On	[illegible]	Application: Netscape Version: 3.01 (WinNT; I) Code Name: Mozilla User Agent: Mozilla/3.01 (WinNT; I)

Using the binding features shown in this example, combined with all the information you can give to or gather from users via HTML dialogs or 4D forms, you can add some very interesting administrative capabilities to your database Web site.

Binding HTML Objects with 4D Variables - Part 3

As seen in the section Web Services, HTML Support, when a 4D form is used as a Web page, 4D provides Server-side Image Mapping by means of invisible-like buttons that overlap a static picture.

If you send an HTML document using SEND HTML FILE, you can bind 4D variables with Image Map HTML objects (INPUT TYPE="IMAGE") to retrieve information. For example, you can create an Image Map HTML object named bImageMap. Each time you click on the image on the browser side, a submit with the click position is sent back to the 4D Web Server. To retrieve the coordinates of the click (expressed relative to the top left corner of the image), you just need to bind the 4D process variable bImageMap and the variables bImageMap_X and bImageMap_Y, which return (as text) the horizontal and vertical coordinates of the click.

In the HTML page, you write something like:

```
<P><INPUT TYPE="image" SRC="MyImage.GIF" NAME="bWorldMap" BORDER=0></P>
```

In the 4D method that sends the HTML page, you write:

```
    bImageMap:=""
    bImageMap_X:=""
    bImageMap_Y:=""
    SEND HTML FILE("ThisPage.HTM")
```

Then, in the POST action 4D method or in the current method, after the POST action method issued a SEND HMTL FILE("") call, you retrieve the coordinates of the click in this way:

```
    $vlX:=Num(bImageMap_X)   ` Get horizontal coordinates in numeric form
    $vlY:=Num(bImageMap_Y)   ` Get vertical coordinates in numeric form
    If (($vlX#0)&($vlY#0))
        ` Do something accordingly to the coordinates
    End if
```

File References and URLs

To insure the maintenance of the database context and subcontext IDs, 4D automatically remaps file references and URLs. For example, 4D remaps all IMG and HREF references to local files.

If you insert your own HTML code into a 4D form using a text variable, you must follow the 4D remapping syntax.

Local GIF files are remapped as "/4DPict/_/GIF_file_pathname/$-2", where GIF_file_pathname is the full HTML path name of the GIF file relative to the root of the volume where the file is located.

Example

The following 4D method returns the remapped reference for the pathname received as parameter:

```
        ` WWW Local GIF URL Project Method
        ` WWW Local GIF URL Project ( Text )
        ` WWW Local GIF URL ( Native pathname ) -> URL to local GIF file
    C_TEXT($0;$1)
    $0:="/4DPict/_/"+HTML Pathname ($1)+"/$-2"
```

Note: For details about the method HTML Pathname, see the examples of the command Mac to ISO.

Then, when inserting HTML code into a 4D form using a text variable, you can write:

```
	vtHTML:=Char(1)+"<P><IMG SRC="+Char(34)+WWW Local GIF URL("F:\ThisImage.HTM"+Char(34)
					+" ALIGN=MIDDLE></P>"+Char(13)
```

This will insert the GIF document in the 4D form at the location of the 4D variable vtHTML.

Important: You only need to write this kind of code to insert custom HTML code into a 4D form. If you just send an HTML page using SEND HTML FILE or if you use a command such as ADD RECORD, remember that 4D transparently translates and remaps the HTML.

The remapping does not change links that have the following protocols:
• http:
• ftp:
• mailto:
• news:
• gopher:
• javascript:
• telnet:

Important: The initial version 6.0 of 4th Dimenson does not support Frames. The FRAME markups have recursive HTML file references that 4D does not yet support.

See Also

SEND HTML FILE.

The Text Parameter Passed to 4D Methods Called via URLs Web Server

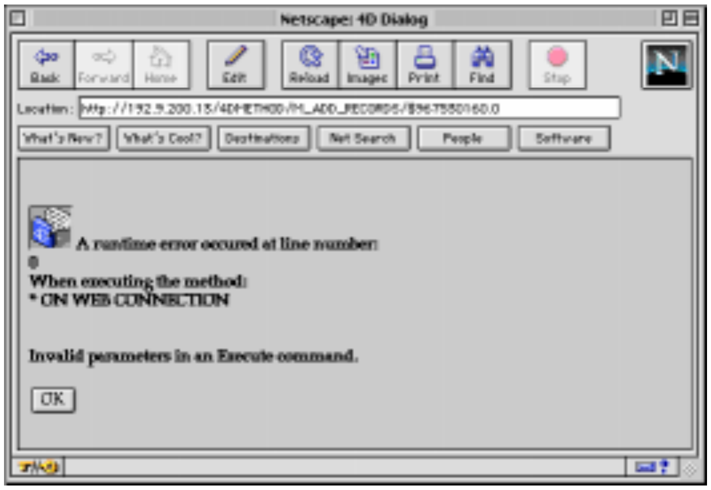
version 6.0.2

4th Dimension sends a text parameter to any 4D method called via a URL. Regarding this text parameter:

• Although you do not use this parameter, you <u>must explicitly declare</u> it with the line C_TEXT($1), otherwise runtime errors will occur while using the Web to access a database that runs in compiled mode.
• This parameter returns the extra data placed at the end of the URL, and can be used as a placeholder for passing values from the HTML environment to the 4D environment.

Runtime Errors in Compiled Mode

Let's consider the following example. You execute a method bound to an HTML object using a link and you obtain the following screen on your Web browser:



http://192.9.200.15/4DMETHOD/M_ADD_RECORDS/$967580160.0

A runtime error occured at line number:
0
When executing the method:
* ON WEB CONNECTION

Invalid parameters in an Execute command.

This runtime error is related to the missing declaration of the text $1 parameter in the 4D method that is called when you click on the HTML link referring to that method. As the context of the execution is the current HTML page, the error refers to the "line 0" of the method that has actually sent the page to the Web browser.

Following the example from the section Web Services, Your First Time (Part I), you eliminate the problem by explicitly declaring the text $1 parameter within the M_ADD_RECORDS and M_LIST_RECORDS methods:

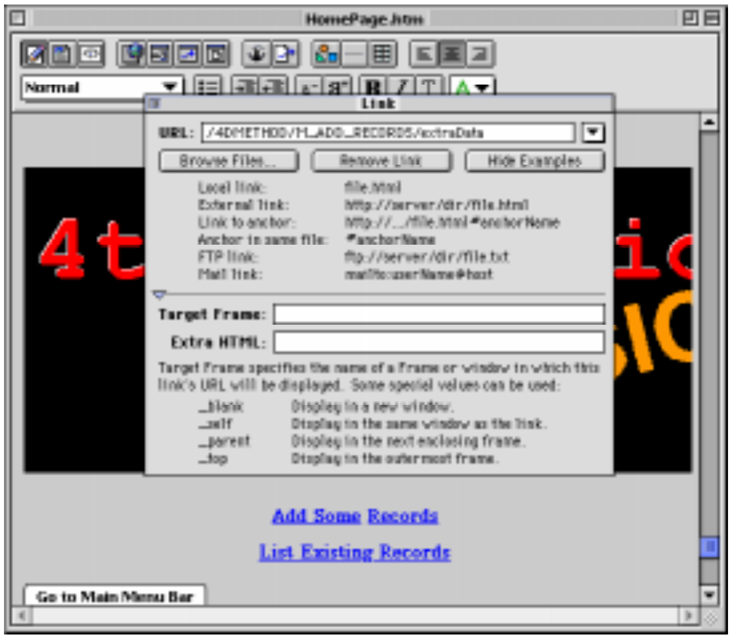
```
     ` M_ADD_RECORDS project method
⇒   C_TEXT($1) ` This parameter MUST be declared explicitely
    Repeat
       ADD RECORD([Customers])
    Until(OK=0)

     ` M_LIST_RECORDS project method
⇒   C_TEXT($1) ` This parameter MUST be declared explicitely
    ALL RECORDS([Customers])
    MODIFY SELECTION([Customers])
```

After these changes have been made, the compiled runtime errors no longer occur.

Working with the URL Extra Data

The text $1 parameter passed to the 4D method returns the extra data appended to the URL.

Again following the example in the Language Reference manual, the change (shown below) is made to the URL of the link that refers to the M_ADD_RECORDS method:



URL: /4DMETHOD/M_ADD_RECORDS/extraData

Local link:	file.html
External link:	http://server/dir/file.html
Link to anchor:	http://.../file.html#anchorName
Anchor in same file:	#anchorName
FTP link:	ftp://server/dir/file.txt
Mail link:	mailto:userName@host

Target Frame:

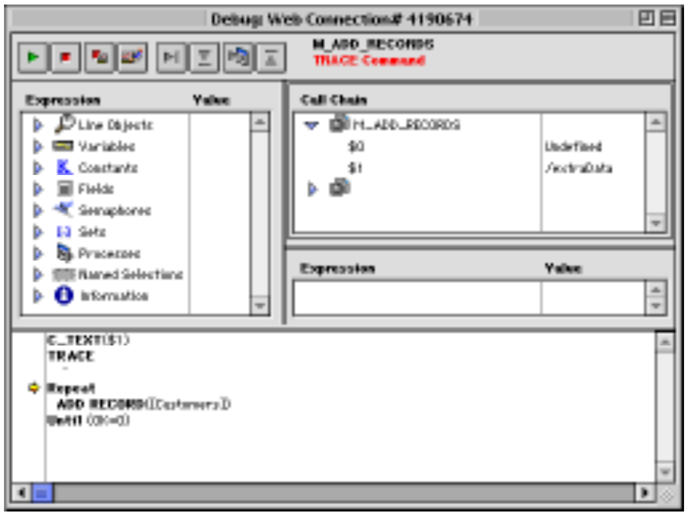
Extra HTML:

Target Frame specifies the name of a Frame or window in which this link's URL will be displayed. Some special values can be used:

_blank	Display in a new window.
_self	Display in the same window as the link.
_parent	Display in the next enclosing frame.
_top	Display in the outermost frame.

Note: The screen shot depicts the change as made using Claris Home Page on MacOS.

The data added to the URL is therefore the string “/extraData”. After this change has been made, you can use the Debugger window, on the 4D side, to quickly check that the $1 parameter actually returns the string “/extraData”:



Call Chain	
$0	Undefined
$1	/extraData

```
C_TEXT($1)
TRACE

Repeat
  ADD RECORD([Customers])
Until (OK=0)
```

By using conventions and algorithms similar to those described in the section On Web Connection Database Method of the Language Reference manual, you therefore have the means to exchange additional data between the HTML and the 4D environments when a 4D method is called by an HTML link.

How to Dynamically Set the URL Extra Data

If you create and write your own HTML files “on the fly” (using, for example, Create document and SEND PACKET), you simply write the URLs accordingly to your needs.

If you work with existing HTML files, you can use JavaScript to dynamically set the link propertie(s) of your object(s).

See Also

Web Services, Your First Time (Part II).

START WEB SERVER

START WEB SERVER

Parameter	Type	Description

This command does not require any parameters

Description

The START WEB SERVER command starts serving your database on your Intranet network or on the Internet using the built-in 4th Dimension Web Server.

If the Web Server is successfully started, OK is set to 1, otherwise OK is set to 0 (zero). For example,
if the TCP/IP network component is missing, OK is set to 0.

See Also

STOP WEB SERVER.

System Variables and Sets

If the Web Services are successfully started, OK is set to 1, otherwise OK is set to 0.

STOP WEB SERVER

Web Server

version 6.0

STOP WEB SERVER

Parameter	Type	Description
This command does not require any parameters		

Description

The command STOP WEB SERVER stops serving your database as a Web Server. If the database was being served as a Web site, all Web connections are stopped, and all Web processes terminated.
If the database was not being served as a Web site, the command does not nothing.

See Also

START WEB SERVER.

SET WEB TIMEOUT

SET WEB TIMEOUT (timeout)

Parameter	Type		Description
timeout	Number	→	Web connections timeout expressed in seconds

Description

The SET WEB TIMEOUT command sets the timeout for the Web Connection processes. The default timeout is 5 minutes.

You reduce or increase this delay by passing, in the timeout parameter, the new timeout expressed in seconds.

The command takes effect immediately, and its scope is the working session; all the Web Connection processes are affected.

See Also

Web Services, Web Connection Processes.

SET WEB DISPLAY LIMITS

Web Server

version 6.0

SET WEB DISPLAY LIMITS (numberRecords{; numberPages{; picRef}})

Parameter	Type		Description
numberRecords	Number	→	Maximum number of records to display in each HTML page
numberPages	Number	→	Maximum number of page references at bottom of each HTML page
picRef	Number	→	Picture reference number for full page record button

Description

The command SET WEB DISPLAY LIMITS modifies the way 4th Dimension displays a selection of records on the Web browser side when you call DISPLAY SELECTION or MODIFY SELECTION.

When you display a selection of records using 4th Dimension or 4D Client, the program does not load all the records of the selection; it only loads (from the disk) the records that are visible in the window at one time. In doing so, although you create a selection of thousands of records, displaying them is quite fast. Then, if you scroll or resize the window, 4D loads the records appropriately.

On the Web, 4D divides the selection of records to be displayed in pages. Without a paging scheme, a selection of thousands of records would result in thousands of records going over the Internet or your Intranet to be displayed in only one Web page. It also would take quite some time to download these records, and your Web browser would more than likely run out of memory.

By default, 4th Dimension displays the first 20 records of the selection and includes, at the end of each HTML page, 20 links to the first 20 pages of the selection. This means that, by default, you can browse the first 400 records of the selection by clicking on the page links located at the end of each selection page. Note that this paging system is transparent to your coding; everything happens within the call to DISPLAY SELECTION or MODIFY SELECTION.

SET WEB DISPLAY LIMITS enables you to change these settings. In the numberRecords parameter, you indicate the maximum number of records you want to display per selection page. In numberPages, you indicate the maximum number of selection page links you want at the end of each selection page.

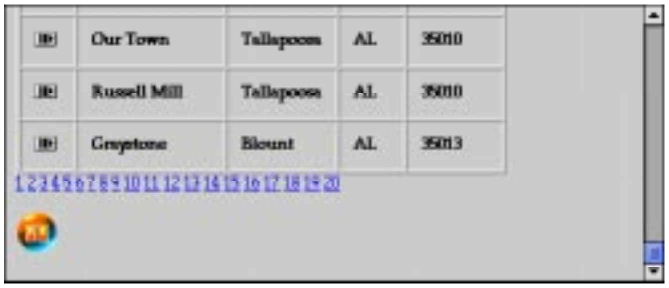
For example, if you have a selection of 10,000 records and want to browse all of them in one display selection, you can pass numberRecords=100 and numberPages=100. However, remember that the data is going over the network or Internet; with the Internet, you must take the speed factor into account when changing the display selection settings.

In addition, SET WEB DISPLAY LIMITS optionally allows you to change the default icon of the full page record button. In the picRef parameter, specify the picture reference number of the picture stored in the database Picture library you want to use as new icon.

SET WEB DISPLAY LIMITS only affects subsequent calls to DISPLAY SELECTION or MODIFY SELECTION, and its scope is local to the current process.

Example

In the following example, a DISPLAY SELECTION or a MODIFY SELECTION is issued for a [Zip Codes] table. By default, 4D displays the records on the Web browser side as shown here:

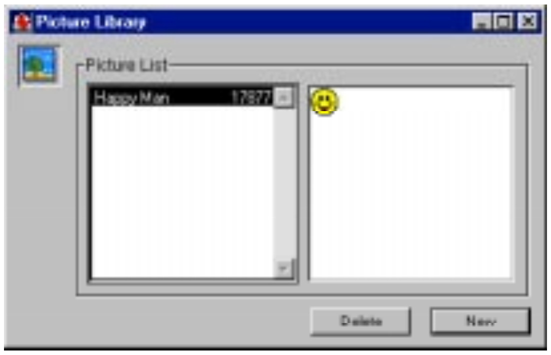


	Our Town	Tallapoosa	AL	35010
	Russell Mill	Tallapoosa	AL	35010
	Graystone	Blount	AL	35013

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Note that the first 400 records can be browsed.

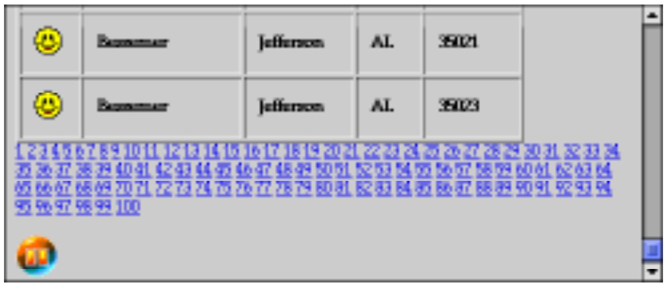
If the following picture is added to the database Picture Library:



And, if the project method that displays the selection performs the SET WEB DISPLAY LIMITS call shown here, prior to the call to DISPLAY SELECTION or MODIFY SELECTION:

SET WEB DISPLAY LIMITS (50;100;17877)

Then the selection on the Web browser side ends up looking like this:



☺	Bessemer	Jefferson	AL	35021
☺	Bessemer	Jefferson	AL	35023

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94
95 96 97 98 99 100

You can now browse the first 50,000 records of the selection.

See Also

DISPLAY SELECTION, MODIFY SELECTION.

SET HTML ROOT

Web Server

version 6.0

SET HTML ROOT (pathnameHTML)

Parameter	Type		Description
pathnameHTML	String	→	HTML Pathname to default directory for HTML files

Description

The command SET HTML ROOT changes the default directory or folder where 4D looks for the HTML file you pass as a parameter to the commande SEND HTML FILE.

By default, 4D looks for the HTML documents in the directory containing the structure file of the database.

The pathname you specify must be an HTML pathname, where the directory or folder names are separated by a slash ("/") character, no matter what the platform. For more information about HTML pathnames, please refer to the Language Reference part of any HTML manual you can find in bookstores.

If you specify an invalid pathname, an OS File manager error is generated. You can intercept the error with an ON ERR CALL method. If you display an alert or a message from within the error method, it will appear on the browser side.

Example

See example for the command SEND HTML FILE.

See Also

ON ERR CALL.

Error Handling

If you specify an invalid pathname, an OS File manager error is generated. You can intercept the error with an ON ERR CALL method.

SEND HTML FILE

SEND HTML FILE (htmlFile)

Parameter	Type		Description
htmlFile	String	→	HTML Pathname to HTML file or empty string for terminating SEND HTML FILE

Description

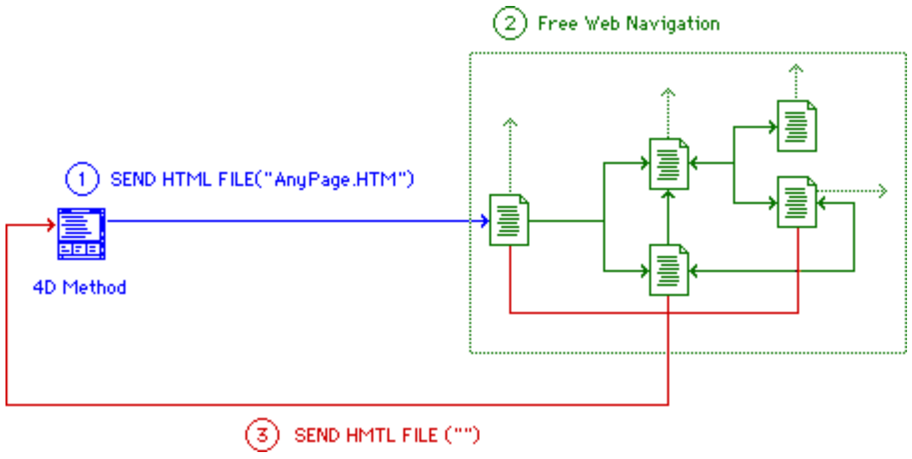
The SEND HTML FILE command sends, to the Web browser, the Web page stored in the HTML document whose pathname you pass in htmlFile.

IMPORTANT: 4th Dimension expects HTML document encoded in ISO Latin-1.

By default, 4th Dimension looks for the HTML document within the directory or folder where the structure file of the database is located, unless you set the default location of the HTML documents to another directory or folder, using the command SET HTML ROOT.

If you specify an invalid HTML pathname, 4D sends the message “The requested HTML page could not be found” to the Web browser.

The alternate syntax SEND HTML FILE(""), in which you pass an empty string in hmtlFile, allows you to terminate the call to SEND HTML FILE, which initiated the HTML mode. This is illustrated in the following diagram:



1. A 4D Method (Project, Object or Database) issues a call to SEND HTML FILE, sending an HTML document to the browser.

2. The initial Web page sent to the browser may have HTML links to other Web pages or can itself refer to 4D Methods that call SEND HTML FILE to send other Web pages. These other pages may have links or refer to 4D Methods for accessing other pages, and so on. While navigating through the Web pages, you can also use browser's navigation controls, such as the Back button.

3. Any of the Web pages can include references to a 4D method that issues a SEND HTML FILE("") call. This call terminates the SEND HTML FILE call that initiated the whole thing, and you go back, pursuing the execution 4D Method that originally started the free Web navigation.

Note: If you call SEND HTML FILE from within a process that is not a Web connection process, the command does nothing and returns no error; the call is simply ignored.

Examples

1. In the directory containing the database structure file, there is an HTML document called "HomePage.HTM". This is the Web page you want Web users to see when they connect to the database, instead of seeing the default menu bar #1 of your database. To present this Web page, in the On Web Connection database method of your application, you write:

```
	` On Web Connection Database Method
⇒	SEND HTML FILE ("HomePage.HTM")
```

2. Your folder database folder is organized as follows:

```
..\Documents\CurrentWork\Databases\MyDB.4DB
..\Documents\CurrentWork\Databases\MyDB.RSR
..\Documents\CurrentWork\Databases\MyDB.4DD
..\Documents\CurrentWork\Databases\WebStuff\HTM\HomePage.HTM
```

You can send the Web page "HomePage.HTM" in this way:

```
⇒	SEND HTML FILE ("WebStuff/HTM/HomePage.HTM")
```

or this way:

```
	SET HTML ROOT ("WebStuff/HTM/")
⇒	SEND HTML FILE ("HomePage.HTM")
```

3. During a 4D Web session, you are adding records using a 4D form. In this form, there is a bHelp button, whose object method is as follows:

```
          ` bHelp button Object Method
⇒     SEND HTML FILE ("Help.HTM")
```

Starting from the Help.HTM document, you can freely navigate between numerous HTML pages which implement the database Help system for your Web site. In each page, you have a submit button titled Done, which allows you to go back to data entry.
To do so, each of the HTML documents must contain the definition of this submit button:

```
      <!-- bDone submit button →
      <P><INPUT TYPE="submit" NAME="bDone" VALUE="Done"></P>
```

as well as the definition of the form post action:

```
      <!-- Execute the 4D htm_Help_Done when a submit button is hit →
      <FORM action="/4DMETHOD/htm_Help_Done" method="POST">
```

On the 4D side, the project method htm_Help_Done terminates the SEND HTML FILE initiated by the bHelp button:

```
          ` htm_Help_Done Project Method
⇒     SEND HTML FILE ("")
```

The call to SEND HTML FILE in the object Method of the bHelp button is the last line of the method. When the method is completed, you return to data entry.

See Also

Web Services, HTML and JavaScript Encapsulation, Web Services, Your First Time (Part II), Web Services, Your First Time (Part II).

55 Windows

Managing Windows

version 6.0

Windows are used to display information to the user. They have three main uses: to enter data, to display data, and to inform the user in messages and dialogs.

There is always at least one window open. Scroll bars are added, when needed, to let the user scroll in a form that is larger than the window. In the User environment, this window displays either the record list (output form) or the data entry screen (input form). In the Custom Menus environment, this window displays a splash screen (a custom graphic).

When you execute a menu command within the Custom Menus process, the splash screen can be replaced with data by commands that display forms. When the commands finish executing, the splash screen is displayed again.

You can open various types of custom windows with the Open Window command. When you no longer need a custom window, you should close it using the CLOSE WINDOW command or by clicking the Control-menu box (Windows) or Close Box (Macintosh), if it exists.

Some commands open their own windows. Commands such as GRAPH TABLE, REPORT, and PRINT LABEL open a window that becomes the frontmost window.

If you start a new process and do not open a window at the beginning of the process method, 4D will automatically open a default one as soon as a form is to be displayed.

See Also

Open window, Window Types.

Open window

version 6.0 (Modified)

Open window (left; top; right; bottom{; type{; title{; controlMenuBox}}}){ → WinRef }

Parameter	Type		Description
left	Number	→	Global left coordinate of window contents area
top	Number	→	Global top coordinate of window contents area
right	Number	→	Global right coordinate of window contents area, or -1 for using form default size
bottom	Number	→	Global bottom coordinate of window contents area, or -1 for using form default size
type	Number	→	Window type
title	String	→	Title of window or "" for using default form title
controlMenuBox	String	→	Method to call when the Control-menu box is double-clicked or the Close box is clicked
Function result	WinRef	←	Window reference number

Description

Open window opens a new window with the dimensions given by the first four parameters:
• left is the distance in pixels from the left edge of the application window to the left internal edge of the window.
• top is the distance in pixels from the top of the application window to the top internal edge of the window.
• right is the distance in pixels from the left edge of the application window to the right internal edge of the window.
• bottom is the distance in pixels from the top of the application window to the bottom internal edge of the window.

If you pass -1 in both right and bottom, you instruct 4D to automatically size the window under the following conditions:
• You have designed a form and set its Sizing Options in the Design environment Form properties window
• Before calling Open window, you selected the form using the command INPUT FORM, to which you passed the optional * parameter.

Important: This automatic sizing of the window will occur only if you made a prior call to INPUT FORM for the form to be displayed, and if you passed the * optional parameter to INPUT FORM.

• The type parameter is optional. It represents the type of window you want to display, and corresponds to the different windows shown in the section Window Types. If the window type is negative, the window created is a floating window. If the type is not specified, type 1 is used by default.

• The title parameter is the optional title for the window

If you pass an empty string ("") in title, you instruct 4D to use the Window Title set in the Design environment Form Properties window for the form to be displayed.

Important: The default form title will be set to the window only if you made a prior call to INPUT FORM for the form to be displayed, and if you passed the * optional parameter to INPUT FORM.

• The controlMenuBox parameter is the optional Control-menu box method for the window. If this parameter is specified, a Control-menu box (Windows) or a Close Box (Macintosh) is added to the window. When the user double-clicks the Control-menu box (Windows) or clicks on the Close Box (Macintosh), the method passed in controlMenuBox is called.

Version 6 Note: You can also manage the closing of the window from within the form method of the form displayed in the window when an On Close Box event occurs. For more information, see the command Form event.

If more than one window is open for a process, the last window opened is the active (frontmost) window for that process. Only information within the active window can be modified. Any other windows can be viewed. When the user types, the active window will always come to the front, if it is not already there.

Forms are displayed inside an open window. Text from the MESSAGE command also appears in the window.

Examples

1. The following project method opens a window centered in the main window (Windows) or in the main screen (Macintosh). Note that it can accept two, three, or four parameters:

```
	` OPEN CENTERED WINDOW project method
	` $1 – Window width
	` $2 – Window height
	` $3 – Window type (optional)
	` $4 – Window title (optional)
$SW:=Screen width\2
$SH:=(Screen height\2)
$WW:=$1\2
$WH:=$2\2
Case of
	: (Count parameters=2)
⇒		Open window($SW-$WW;$SH-$WH;$SW+$WW;$SH+$WH)
	: (Count parameters=3)
⇒		Open window($SW-$WW;$SH-$WH;$SW+$WW;$SH+$WH;$3)
	: (Count parameters=4)
⇒		Open window($SW-$WW;$SH-$WH;$SW+$WW;$SH+$WH;$3;$4)
End case
```

After the project method is written, you can use it this way:

```
OPEN CENTERED WINDOW (400;250;Movable dialog box;"Update Archives")
DIALOG([Utility Table];"UPDATE OPTIONS")
CLOSE WINDOW
If (OK=1)
	` ...
End if
```

2. The following example opens a floating window that has a Control-menu box (Windows) or Close Box (Macintosh) method. The window is opened in the upper right hand corner of the application window.

```
⇒	Open window(Screen width-149;33;Screen width-4;178;- Palette
window;"";"CloseColorPalette")
	DIALOG([Dialogs];"Color Palette")
```

The CloseColorPalette method calls the CANCEL command:

```
	CANCEL
```

3. The following example opens a window whose size and title come from the properties of the form displayed in the window:

```
       INPUT FORM([Customers];"Add Records";*)
⇒      Open window(10;80;-1;-1;Plain window;"")
       Repeat
          ADD RECORD([Customers])
       Until (OK=0)
```

Reminder: In order to have Open window automatically use the properties of the form, you must call INPUT FORM with the optional * parameter, and the properties of the form must have been set accordingly in the Design environment.

See Also

CLOSE WINDOW, Open external window.

Window Types

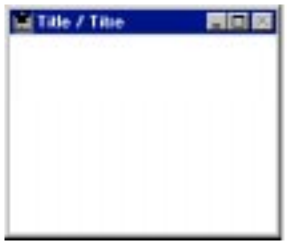
You can use one of the following predefined constants to specify the type of window that you open with Open window:

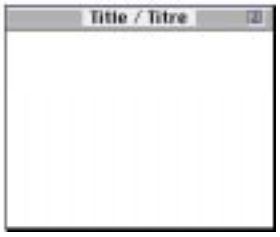
Constant	Type	Value	Can be a floating window
Plain window	Long Integer	8	No
Plain no zoom box window	Long Integer	0	No
Plain fixed size window	Long Integer	4	No
Modal dialog box	Long Integer	1	No
Alternate dialog box	Long Integer	3	Yes
Movable dialog box	Long Integer	5	Yes
Plain dialog box	Long Integer	2	Yes
Palette window	Long Integer	720	Yes
Round corner window	Long Integer	16	No

Floating Windows: If you pass one of these constants to Open window, you open a regular windows. To open a floating windows, pass a negative window type value to Open window.

The following table shows each window type, on Windows (left) and on Macintosh (right).

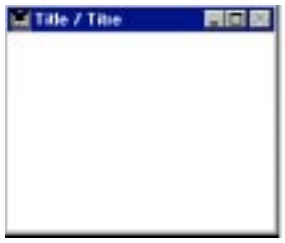
Plain window (8)

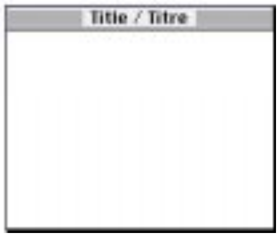




- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: Yes
- Can be minimized/maximized or zoomed: Yes
- Suitable for scroll bars: Yes
- Usage: data entry with scrollbars, DISPLAY SELECTION, MODIFY SELECTION, etc.

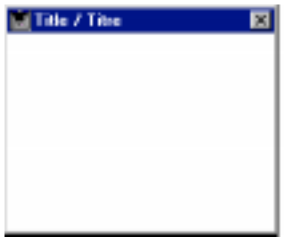
Plain no zoom box window (0)

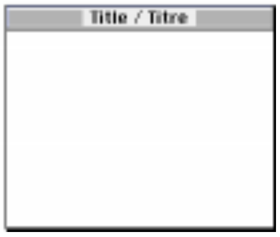




- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: Yes
- Can be minimized/maximized or zoomed: No on Macintosh
- Suitable for scroll bars: Yes
- Usage: data entry with scrollbars, DISPLAY SELECTION, MODIFY SELECTION, etc.

Plain fixed size window (4)





- Can have a title: Yes
- Can have a close box or equivalent: Yes
- Can be resized: No on Macintosh
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: Yes and No
- Usage: data entry with ADD RECORD(...;...*) or equivalent

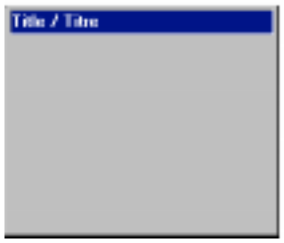
Modal dialog box (1)

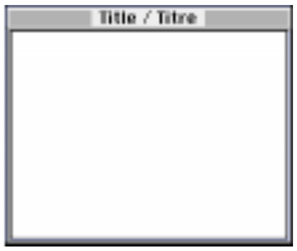
- Can have a title: No
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: DIALOG, ADD RECORD(...;...;*) or equivalent
- Windows of this type are modal

Alternate dialog box (3)

- Can have a title: No
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: DIALOG, ADD RECORD(...;...;*) or equivalent
- Windows of this type are modal, unless used as floating windows

Movable dialog box (5)



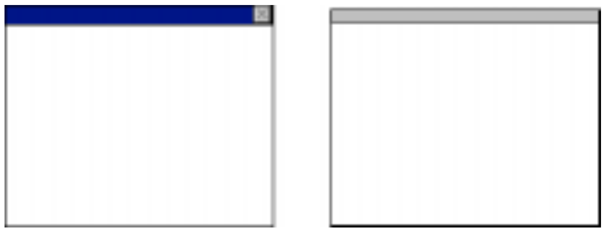


- Can have a title: Yes
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: DIALOG, ADD RECORD(...;...;*) or equivalent
- Windows of this type are modal, but can be moved and can be used as floating windows

Plain dialog box (2)

- Can have a title: No
- Can have a close box or equivalent: No
- Can be resized: No
- Can be minimized/maximized or zoomed: No
- Suitable for scroll bars: No
- Usage: DIALOG, ADD RECORD(...;...;*) or equivalent, splashscreens
- Windows of this type are modal, unless used as floating windows

Palette window (720 {+ 1} {+ 2} {+ 4} {+ 8})

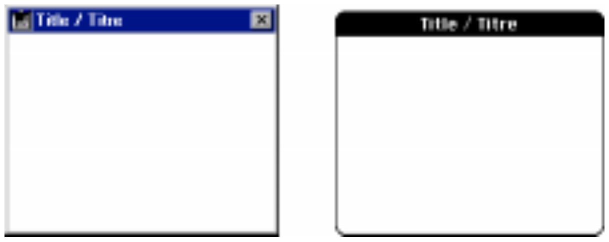


When you call Open window, you can add one or several of the following constants to Palette window in order to obtain variations in the behavior of the window:

Constant	Type	Value
Has zoom box	Long Integer	8
Has grow box	Long Integer	4
Has window title	Long Integer	2
Has highlight	Long Integer	1

• Can have a title: Yes, if Has window title variation is specified
• Can have a close box or equivalent: Yes
• Can be resized: Yes, if Has grow box variation is specified
• Can be minimized/maximized or zoomed: Yes, if Has zoom box variation is specified
• Suitable for scroll bars: Yes, if Has grow box variation is specified
• Usage: Floating windows with DIALOG or DISPLAY SELECTION (no data entry)

Round corner window (16)



• Can have a title: Yes
• Can have a close box or equivalent: Yes
• Can be resized: No on Macintosh
• Can be minimized/maximized or zoomed: No
• Suitable for scroll bars: No on Macintosh
• Usage: Rare

See Also

Open external window, Open window.

Open external window

version 6.0 (Modified)

Open external window (left; top; right; bottom; type; title; plugInArea) → Number

Parameter	Type		Description
left	Number	→	Global left coordinate of window contents area
top	Number	→	Global top coordinate of window contents area
right	Number	→	Global right coordinate of window contents area
bottom	Number	→	Global bottom coordinate of window contents area
type	Number	→	Window type
title	String	→	Title of window
plugInArea	String	→	External area command
Function result	Number	←	Window reference number

Description

Open external window opens a new window and displays the external area supported by the command plugInArea provided by a 4D plug-in.

Open external window returns a Long Integer value that can be used both as a window reference number (that can be used with other Windows commands) and as a reference to the external area displayed in the window (that can be used with other routines provided by the 4D plug-in).

The first six arguments are the same as those of the the Open window command. However, none of the parameters are optional.

Open external window creates modeless windows. The command does not wait for user input, so you can have several active windows open at once. You can click between each window and edit the one in front. If the window type has a title bar, a Control-menu box (Windows) or a Close Box (Macintosh) will be added to enable the user to close the window.

Examples

The following example opens an external window and displays the 4D Write external area:

```
⇒     wrWind:=Open external window (50; 50; 350; 450; 8; "Letter Writing"; "_4D WRITE")
```

The following example closes the external window opened in the previous example:

```
      CLOSE WINDOW (wrWind)
```

See Also

CLOSE WINDOW, Open window.

CLOSE WINDOW

CLOSE WINDOW {(extWindowRef)}

Parameter	Type		Description
extWindowRef	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

CLOSE WINDOW closes the active window opened by an Open window command in the current process. CLOSE WINDOW has no effect if a custom window is not open; it does not close standard windows. CLOSE WINDOW also has no effect if called while a form is active in the window. You must call CLOSE WINDOW when you are done using a window opened by Open window.

If you pass an external window reference number in the extWindowRef parameter, CLOSE WINDOW closes the specified external window. For more information about external windows, refer to the Open external window function.

Example

The following example opens a window and adds new records with the ADD RECORD command. When the records have been added, the window is closed with CLOSE WINDOW:

```
Open window (5; 40; 250; 300; 0; "New Employee")
Repeat
   ADD RECORD ([Employees])   ` Add a new employee record
Until (OK = 0)   ` Loop until the user cancels
CLOSE WINDOW   ` Close the window
```

See Also

Open external window, Open window.

ERASE WINDOW

version 6.0 (Modified)

ERASE WINDOW {(window)}

Parameter	Type		Description
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

The command ERASE WINDOW clears the contents of the window whose reference number is passed in window.

If you omit the window parameter, ERASE WINDOW clears the contents of the frontmost window for the current process.

Usually, you will use ERASE WINDOW in combination with MESSAGE and GOTO XY. In this case, ERASE WINDOW clears the contents of the window and moves the cursor to the upper-left corner of the window, the GOTO XY (0; 0) position.

Do not confuse ERASE WINDOW, which clears the contents of a window, with CLOSE WINDOW, which removes the window from the screen.

See Also

GOTO XY, MESSAGE.

REDRAW WINDOW

REDRAW WINDOW {(window)}

Parameter	Type		Description
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

The command REDRAW WINDOW provokes a graphical update of the window whose reference number you pass in window.

If you omit the window parameter, REDRAW WINDOW applies to the frontmost window for the current process.

Note: 4th Dimension handles the graphical updates of the windows each time you move a window, resize it, or bring it to the front, as well as when you change the form and/or the values displayed in the window. You will rarely use this command.

See Also

ERASE WINDOW.

DRAG WINDOW

Windows

version 6.0

DRAG WINDOW

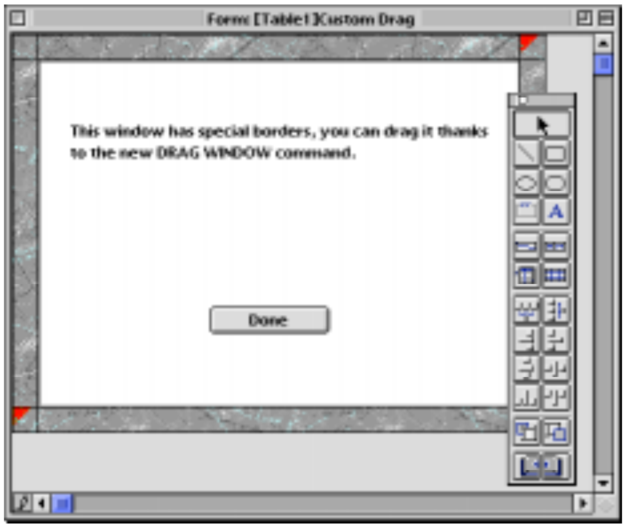
Parameter	Type	Description
This command does not require any parameters		

Description

The command DRAG WINDOW drags the current frontmost window following the movements of the mouse. Usually you call this command from within an object method of an object that can respond instantaneously to mouse clicks (i.e., invisible buttons).

Example

The following form, shown here in the Design Environment, contains a frame created with a static picture, above which are four invisible buttons for each side:



This window has special borders, you can drag it thanks to the new DRAG WINDOW command.

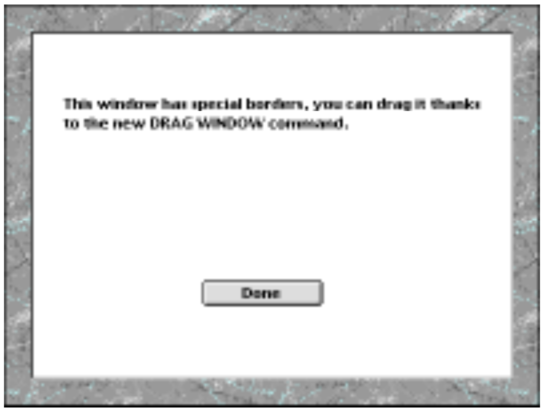
Each button has the following method:

```
DRAG WINDOW ` Start dragging window when clicked
```

In the User or Custom Menus environment, after executing the following project method:

```
Open window(50;50;50+400;50+300;2)
DIALOG([Table1];"Custom Drag")
CLOSE WINDOW
```

You obtain a window similar to this:



This window has special borders, you can drag it thanks to the new DRAG WINDOW command.

Then you can drag the window by clicking anywhere on the borders.

See Also

GET WINDOW RECT, SET WINDOW RECT.

Get window title

Get window title {(window)} → String

Parameter	Type		Description
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted
Function result	String	←	Window title

Description

The command Get window title returns the title of the window whose reference number is passed in window. If the window does not exist, an empty string is returned.

If you omit the window parameter, Get window title returns the title of the frontmost window for the current process.

Example

See example for the command SET WINDOW TITLE.

See Also

SET WINDOW TITLE.

SET WINDOW TITLE

version 6.0 (Modified)

SET WINDOW TITLE (title{; window})

Parameter	Type		Description
title	String	→	Window title
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

The command SET WINDOW TITLE changes the title of the window whose reference number is passed in window to the text passed in title (max. length 80 characters). If the window does not exist, SET WINDOW TITLE does nothing. If you omit the window parameter, SET WINDOW TITLE changes the title of the frontmost window for the current process.

Note: In the User environment, 4th Dimension changes the window titles automatically —i.e., "Entry for Table" when you perform data entry. If you change a window title, 4D will probably override it. On the other hand, in the Custom Menus environment, 4th Dimension does not change the titles of the windows.

Example

While performing data entry in a form, you click on a button that executes a lengthy operation (i.e., browsing programmatically related records shown in a subform). You keep informed about the progress of the operation using the title of the current window:

```
   ` bAnalysis button Object Method
Case of
   : (Form event=On Clicked)
      $vsCurTitle:=Get window title ` Save current window title in a local variable
      FIRST RECORD([Invoice Line Items])  ` Start the lengthy operation
      For($vlRecord;1;Records in selection([Invoice Line Items]))
         DO SOMETHING
            ` Show progress information
         SET WINDOW TITLE("Processing Line Item #"+String($vlRecord))
      End for
         ` Restore original window title
      SET WINDOW TITLE($vsCurTitle)
End case
```

See Also

Get window title.

HIDE TOOL BAR

Windows

version 6.0

HIDE TOOL BAR

Parameter	Type	Description
This command does not require any parameters		

Description

The command HIDE TOOL BAR makes the toolbar invisible.

If the toolbar was already hidden, HIDE TOOL BAR does nothing.

See Also

HIDE MENU BAR, SHOW MENU BAR, SHOW TOOL BAR.

SHOW TOOL BAR

Windows

version 6.0

SHOW TOOL BAR

Parameter	Type	Description
This command does not require any parameters		

Description

The command SHOW TOOL BAR makes the toolbar visible.

If the toolbar was already visible, SHOW TOOL BAR does nothing.

See Also

HIDE MENU BAR, HIDE TOOL BAR, SHOW MENU BAR.

WINDOW LIST

WINDOW LIST (windows{; *})

Parameter	Type		Description
windows	Array	←	Array of window reference numbers
*	*	→	If specified, take floating windows into account If omitted, ignore floating windows

Description

The command WINDOW LIST populates the array windows with the window reference numbers of the windows currently open in all running processes (kernel or user processes).

If you do not pass the optional * parameter, floating windows are ignored.

Example

The following project method tiles all the current open window, except floating windows and dialog boxes:

```
          ` TILE WINDOWS project method

⇒    WINDOW LIST($alWnd)
     $vlLeft:=10
     $vlTop:=80 ` Leave enough room for the Tool bar
     For ($vlWnd;1;Size of array($alWnd))
        If (Window kind($alWnd{$vlWnd}) # Modal Dialog)
           GET WINDOW RECT($vlWL;$vlWT;$vlWR;$vlWB;$alWnd{$vlWnd})
           $vlWR:=$vlLeft+($vlWR-$vlWL)
           $vlWB:=$vlTop+($vlWB-$vlWT)
           $vlWL:=$vlLeft
           $vlWT:=$vlTop
           SET WINDOW RECT($vlWL;$vlWT;$vlWR;$vlWB;$alWnd{$vlWnd})
           $vlLeft:=$vlLeft+10
           $vlTop:=$vlTop+25
        End if
     End for
```

Note: This method could be improved by adding tests on the size of the main window (on Windows) or the size and location of the screens (on Macintosh).

See Also

Window kind, Window process.

Window kind

Windows

version 6.0

Window kind {(window)}

Parameter	Type		Description
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

The command Window kind returns the 4th Dimension type of the window whose reference number is passed in window. If the window does not exist, Window kind returns 0 (zero).

Otherwise, Window kind may return one of the following values:

Constant	Type	Value
Regular window	Long Integer	8
Modal dialog	Long Integer	9
External window	Long Integer	5
Floating window	Long Integer	14

If you omit the window parameter, Window kind returns the type of the frontmost window for the current process.

Example

Set example for the command WINDOW LIST.

See Also

GET WINDOW RECT, Get window title, Window process.

Window process

Window process {(window)} → Number

Parameter	Type		Description
window	WinRef	→	Window reference number
Function result	Number	←	Process reference number

Description

The command Window process returns the process number that runs the window whose reference number is passed in window. If the window does not exist, 0 (zero) is returned.

If you omit the window parameter, Window process returns the process of the current frontmost window.

See Also

Current process.

GET WINDOW RECT

GET WINDOW RECT (left; top; right; bottom{; window})

Parameter	Type		Description
left	Number	←	Global left coordinate of window's contents area
top	Number	←	Global top coordinate of window's contents area
right	Number	←	Global right coordinate of window's contents area
bottom	Number	←	Global bottom coordinate of window's contents area
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

The command GET WINDOW RECT returns the global coordinates of the window whose reference number is passed in window. If the window does not exist, the variable parameters are left unchanged.

If you omit the window parameter, GET WINDOW RECT applies to the frontmost window for the current process.

The coordinates are expressed relative to the top left corner of the contents area of the application window (on Windows) or of the main screen (on Macintosh). The coordinates return the rectangle corresponding to the contents area of the window (excluding title bars and borders).

Example

See example for the command WINDOW LIST.

See Also

SET WINDOW RECT.

SET WINDOW RECT

Windows

version 6.0

SET WINDOW RECT (left; top; right; bottom{; window})

Parameter	Type		Description
left	Number	→	Global left coordinate of window's contents area
top	Number	→	Global top coordinate of window's contents area
right	Number	→	Global right coordinate of window's contents area
bottom	Number	→	Global bottom coordinate of window's contents area
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted

Description

The command SET WINDOW RECT changes the global coordinates of the the window whose reference number is passed in window. If the window does not exist, the command does nothing.

If you omit the window parameter, SET WINDOW RECT applies to the frontmost window for the current process.

This command can resize and move the window, depending on the new coordinates passed.

The coordinates must be expressed relative to the top left corner of the contents area of the application window (on Windows) or to the main screen (on Macintosh). The coordinates indicate the rectangle corresponding to the contents area of the window (excluding title bars and borders).

Warning: Be aware that by using this command, you may move a window beyond the limits of the main window (on Windows) or of the screens (on Macintosh). To prevent this, use commands such as Screen width and Screen height to double-check the new coordinates of the window.

Example

See example for the command WINDOW LIST.

See Also

DRAG WINDOW, GET WINDOW RECT.

Frontmost window

version 6.0

Frontmost window {(*)} → WinRef

Parameter	Type		Description
*	*	→	If specified, take floating windows into account If omitted, ignore floating windows
Function result	WinRef	←	Window reference number

Description

The command Frontmost window returns the window reference number of the frontmost window.

See Also

Frontmost process, Next window.

Next window

Next window (window) → Number

Parameter	Type		Description
window	WinRef	→	Window reference number, or Frontmost window of current process, if omitted
Function result	Number	←	Window reference number

Description

The command Next window returns the window reference number of the window "behind" the window you pass in window (based on the front-to-back order of the windows).

See Also

Frontmost window.

Find window

Find window (left; top{; windowPart}) → WinRef

Parameter	Type		Description
left	Number	→	Global left coordinate
top	Number	→	Global top coordinate
windowPart	Number	←	Window part ID number
Function result	WinRef	←	Window reference number

Description

The command Find window returns (if any) the reference number of the first window "touched" by the point whose coordinates passed in left and top.

The coordinates must be expressed relative to the top left corner of the contents area of the application window (Windows) or to the main screen (Macintosh).

If you specify the windowPart parameter, whether or not a window has been found, the parameter returns one of the following values:

Constants	Type	Value	Platform
In menu bar	Long Integer	1	Macintosh only
In system window	Long Integer	2	Macintosh only
In contents	Long Integer	3	Windows or Macintosh
In drag	Long Integer	4	Macintosh only
In grow	Long Integer	5	Macintosh only
In go away	Long Integer	6	Macintosh only
In zoom box	Long Integer	7	Macintosh only

See Also

Frontmost window, Next window.

56 Error Codes

Syntax Errors

Error Codes

version 6.0

The following table lists the syntax error codes for errors that may occur during code execution in the User or Custom Menus environment. Some of these errors may occur in interpreted mode only, some in compiled mode only, some in both modes. You can intercept these errors using an error interruption method installed using ON ERR CALL.

Code	Description
1	A “(” was expected.
2	A field was expected.
3	The command may be executed only on a field in a subtable.
4	Parameters in the list must all be of the same type.
5	There is no table to which to apply the command.
6	The command may only be executed on a Subtable type field.
7	A Numeric argument was expected.
8	An Alphanumeric argument was expected.
9	The result of a conditional test was expected.
10	The command cannot be applied to this field type.
11	The command cannot be applied between two conditional tests.
12	The command cannot be applied between two Numeric arguments.
13	The command cannot be applied between two Alphanumeric arguments.
14	The command cannot be applied between two Date arguments.
15	The operation is not compatible with the two arguments.
16	The field has no relation.
17	A table was expected.
18	Field types are incompatible.
19	The field is not indexed.
20	An “=” was expected.
21	The method does not exist.
22	The fields must belong to the same table or subtable for a sort or graph.
23	A “<” or “>” was expected.
24	A “;” was expected.
25	There are too many fields for a sort.
26	The field type cannot be Text, Picture or Subtable.
27	The field must be prefixed by the name of its table.
28	The field type must be Numeric.
29	The value must be 1 or 0.
30	A variable was expected.
31	There is no menu bar with this number.
32	A date was expected.
33	Unimplemented command or function.
34	
35	The sets are from different tables.

36	Invalid table name.
37	A ":=" was expected.
38	
39	The set does not exist.
40	This is a procedure, not a function.
41	A variable or field belonging to a subtable was expected.
42	The record cannot be pushed onto the stack.
43	The function cannot be found.
44	The method cannot be found.
45	Field or variable expected.
46	A Numeric or Alphanumeric argument was expected.
47	The field type must be Alphanumeric.
48	Syntax error.
49	This operator cannot be used here.
50	These operators cannot be used together.
51	Module not implemented.
52	An array was expected.
53	Indice out of range.
54	Argument types are incompatible.
55	A Boolean argument was expected.
56	Field, variable, or table expected.
57	An operator was expected.
58	A ")" was expected.
59	This kind of argument was not expected here.
60	A parameter or a local variable cannot be used in an EXECUTE statement in a compiled database.
61	The type of an array cannot be modified in a compiled database.
62	The command cannot be applied to a subtable.
63	The field is not indexed.
64	A picture field or variable was expected.
65	
66	
67	This command cannot be executed on 4D Server.
68	A list was expected.

Tips

Some of these error codes denote plain syntax errors due to mistyping. For example, you get an error #37 if you execute the statement v=0 when you actually meant v:=0. You can eliminate the error by fixing your code in the Design Method Editor.

Some of these error codes are due to simple programming errors. For example, you get an error #5 if you execute an ADD RECORD command, when you have not first set the default table (using the DEFAULT TABLE command), and do not pass the table parameter. In this case, there is no table to which to apply the command. You eliminate the error by checking to see if you forgot to set the default table or if you forgot to pass the table parameter to the command for this occurrence.

Some of these error codes denote errors due to a flaw in the design. For example, you get an error #16 if you apply RELATE ONE to a field that is not related to any other field. You eliminate the error by checking to see if your code is actually wrong or if you simply forgot to create the relation starting from the field.

Some errors, when they occur, are not always located exactly where your code breaks. For example, if in a subroutine you get an error #53 (indice out of range) on the line vpFld:=Field($1;$2), the error is due to a wrong table and/or field number that has been passed to the subroutine as a parameter. Therefore, the error is located in the caller method and not where the error actually occurs. In this case, trace your code in the Debugger window to determine which line of code is the real culprit, then fix it in the Design Method Editor.

See Also

ON ERR CALL.

Database Engine Errors

Error Codes

version 6.0

This table lists the error codes generated by the 4th Dimension Database Engine. These codes cover errors that occur at a low level of the database engine, such as user interruption, privilege errors, and damaged objects.

Code	Description
1006	Program interrupted by user—user pressed Alt-click (Windows) or Option-click (MacOS)
-9937	Password System is locked by another user.
-9938	The current record has been changed from within the trigger.
-9939	External routine not found.
-9940	4D Extension initialization failed.
-9941	Unknown EX_GESTALT selector.
-9942	4D Client licensing scheme is incompatible with this version of 4D Server.
-9943	4D Passport version error.
-9944	The user does not belong to the 4D Open access group.
-9945	CD-ROM 4D Runtime error, writing operations are not allowed.
-9946	Unable to clear the named selection because it does not exist.
-9947	The “Allow 4D Client connections only” check box has been selected.
-9950	Invalid data segment number.
-9951	This field has no relation.
-9952	Invalid data segment header.
-9953	There is no Log file.
-9954	There is no current record.
-9955	QuickTime is not installed.
-9956	Versions of 4D Client and 4D Server are different.
-9957	The choice list is locked.
-9958	The process could not be started.
-9959	The backup process has already been started by another user or process.
-9960	4D Backup is not installed on the server.
-9961	The backup process is not currently running.
-9962	The backup cannot be run because the server is shutting down.
-9963	Invalid record number requested by a workstation.
-9964	Bad sort definition table sent by a workstation.
-9965	Bad search definition table sent by a workstation.
-9966	Invalid type requested by a workstation.
-9967	The record could not be modified because it could not be loaded.
-9968	Invalid selected record number requested by workstation.
-9969	Invalid field type requested by workstation.
-9970	Field is not indexed.
-9971	Field number is out of range requested by workstation.
-9972	File number is out of range requested by workstation.

-9973	The TRIC resources are not the same.
-9974	Record has already been deleted.
-9975	Transaction index page could not be loaded.
-9976	Backup in progress, no modification allowed.
-9977	The selection does not exist.
-9978	Bad user password.
-9979	Unknown user.
-9980	The file cannot be created because the structure is locked.
-9981	Invalid field name/field number definition table sent by the workstation.
-9982	The record was not loaded because it is not in the selection on the workstation.
-9983	The same external package is installed twice.
-9984	Transaction has been cancelled because of a duplicated index key error.
-9985	Recursive integrity.
-9986	Record locked during an automatic deletion action.
-9987	Some other records are already related to this record.
-9989	Invalid structure (database needs to be repaired).
-9990	Time-out error.
-9991	Privileges error.
-9992	Wrong password.
-9993	Menu bar is damaged (database needs to be repaired).
-9994	Serial communication interrupted by the user, user pressed Ctrl-Alt-Shift (Windows) or Command-Option-Shift (MacOS).
-9995	Demo limit has been reached.
-9996	Stack is full (too much recursion or nested calls).
-9997	Maximum number of records has been reached.
-9998	Duplicated key.
-9999	No more room to save the record. (see note 4)
-10500	Invalid record address (database needs to be repaired) .
-10501	Invalid index page (index needs to be repaired).
-10502	Invalid record structure (data file needs to be repaired).
-10503	Record # is out of range. (during GOTO RECORD, or data file needs to be repaired) (see note 3)
-10504	Index page # is out of range (index needs to be repaired).

-1	Unkown entry point requested by a Plug-In
4001	Invalid table number requested by a Plug-In
4002	Invalid record number requested by a Plug-In
4003	Invalid field number requested by a Plug-In
4004	Access to a table's current record requested by a Plug-in while there is no current record

Notes

1. While some of the errors listed reflect serious problems, i.e., -10502 Invalid record structure (data file needs to be repaired), other errors may occur on a regular basis and can be managed using an ON ERR CALL project method. For example, it is common to handle the error –9998 Duplicated key if your application offers opportunities to create duplicated values for a table that includes an indexed field whose Unique property is set.

2. Some of the errors listed never occur at the 4D language level. They can occur and be handled at a low level by database engine routines or when using 4D Backup or 4D Open.

3. The error -10503 Record # is out of range does NOT always mean that the database needs to be repaired. This error may occur if you attempt to use the record number (i.e., the command GOTO RECORD) for a newly created record in transaction. The reason is that newly created records, while in a transaction, are assigned temporary record numbers until the transaction is validated. If this error occurs in that context, your database is fine, but your algorithm is not.

4. The error -9999 No more room to save the record occurs when all the segments of your database are full or located on full volumes. This error can also be generated if the data file is locked or located on a locked volume. This allows you, for example, to detect locked data files from within the On Startup Database Method of your application. For more information on this subject, see Testing the locked status of the data file.

See Also

ON ERR CALL.

Network Components Errors

The following table describes the errors that can occur with a network component.

Code	Description
-10001	The actual connection to the database has been disrupted.
-10002	The connection for this process has been disrupted.
-10003	Bad connection parameters.
-10020	No server was selected while using OP Select 4D server.
-10021	No server was found while using OP Find 4D server.
-10050	Unknown option in Get/SetOption.
-10051	Incorrect value in Get/SetOption.
-10130	The state of the connection does not allow you to continue this session.
-10131	The connection has been aborted.
-10132	Some connection parameters are invalid.
2	The user clicked the Other button while using OP Select 4D Server.

OS File Manager Errors

The following table lists codes returned by the Operating System File Manager. These codes can be returned when you are using, for example, the System Documents commands. In this list, the word “file” indicates a document on disk and not a file in your database structure.

Code	Description
-33	File directory full. You cannot create new files on disk.
-34	Disk is full. There is no more room available on the disk.
-35	Specified volume doesn’t exist.
-36	I/O error. There is probably a bad block on the disk.
-37	Bad filename or volume name.
-38	Tried to read or write to a file that is not open.
-39	Logical end-of-file reached during read operation.
-40	Attempt to position before start of file.
-41	Not enough memory to open a new file on the disk.
-42	Too many files open at the same time.
-43	File not found.
-44	Volume is locked by a hardware setting.
-45	File is locked.
-46	Volume is locked by an application.
-47	Tried to access a file that has been deleted.
-48	Tried to rename a file with the name of an already deleted file.
-49	Tried to open a file already open with write permission.
-51	Tried to access a document with an invalid document reference number.
-52	Internal file manager error (position of file marker is lost).
-53	Volume not on line.
-54	Attempt to open locked file for writing.
-57	Tried to work with a non-Windows disk.
-58	Tried to work with a non-Windows disk.
-60	Bad master directory block. Your disk is damaged.
-61	Read/write permission doesn’t allow writing.
-64	There is a hardware problem with the disk (bad installation, incorrect formatting,...).
-84	There is a hardware problem with the disk (bad installation, incorrect formatting,...).
-120	Tried to access a file by using a pathname that specifies a non existing directory.
-121	An access path could not be created.
-124	Tried to access a disconnected shared volume.

See Also

ON ERR CALL.

OS Memory Manager Errors

The following table lists the error codes returned by the Operating System Memory Manager.

Code	Description
-108	Not enough memory to perform an operation. Give more memory to your 4D application.
-109	Internal Memory problem. Memory is probably logically corrupted. Exit as soon as possible. Restart your machine and reopen the database.
-111	Internal Memory problem. Memory is probably logically corrupted. Exit as soon as possible. Restart your machine and reopen the database. (*)
-117	Internal Memory problem. Memory is probably logically corrupted. Exit as soon as possible. Restart your machine and reopen the database.

Tip: When allocating and working with large arrays, BLOBs, pictures, as well as sets (objects that can hold large amount of data), use an ON ERR CALL project method to test the error -108.

(*) Error -111 can also occur when you attempt to read a value from a BLOB with an offset out of range. In this case, the error is minor and you do not need to terminate the working session. Just fix the offset you pass to the BLOB command.

See Also

ON ERR CALL.

OS Printing Manager Errors

The following table lists the error codes returned by the Operating System Printing Manager. These codes can be returned during printing.

Code	Description
-1	Problem saving file to be printed
-27	Problem opening or closing connection with printer
-128	Printing interrupted by the user
-193	Resource file not found
-4100	Printer connection has been interrupted
-4101	Printer busy or not connected
-8150	A LaserWriter is not selected
-8151	The printer has been initialized with a different driver version
-8192	LaserWriter time-out

See Also

ON ERR CALL.

OS Resource Manager Errors

The following table lists the error codes returned by the Operating System Resource Manager.

Code	Description
-1	Resource file could not be opened
-192	Resource not found
-193	Resource map is damaged (file needs to be repaired)
-194	Resource could not be added
-196	Resource could not be deleted

See Also

ON ERR CALL.

SANE NaN Errors

The following table lists the NaN codes returned by the Operating System. NaN is a Standard Apple Numeric Environment (SANE) representation which means “Not a Number.” NaN appears when an operation produces a result that is beyond SANE’s scope.

Code	Description
1	Invalid square root
2	Invalid addition
4	Invalid division
8	Invalid multiplication
9	Invalid remainder
17	Converting an invalid ASCII string
20	Converting a Comp type number to floating-point
21	Creating a NaN with a zero code
33	Invalid argument to a trig function
34	Invalid argument to an inverse trig function
36	Invalid argument to a log function
37	Invalid argument to an xi or xy function
38	Invalid argument to a financial function
255	Uninitialized storage

OS Sound Manager Errors

The following table lists the codes returned by the Operating System Sound Manager.

Code	Description
-203	Too many sound commands
-204	The sound resource cannot be loaded
-205	The sound channel is logically corrupted
-206	The format of the sound resource is wrong
-207	Not enough memory to perform the sound
-209	The sound channel is busy

See Also

ON ERR CALL.

OS Serial Ports Manager Errors

The following table lists error codes returned by the Operating System Serial Ports Manager.

Code	Description
-28	There is no open serial port

See Also

ON ERR CALL.

MacOS System Errors

The following table lists some of the MacOS system errors. It is usually not possible to recover from these errors.

Code	Description
4	Zero divide
15	Segment Loader Error: 4th Dimension failed in loading one of its own code segments. You must allocate more memory to 4th Dimension.
17 to 24	A system package is missing. Check if your system directory has been correctly installed
25	Out of memory You must allocate more memory to 4th Dimension
28	Stack has moved into the application heap. You must allocate more memory to 4th Dimension

Testing the locked status of the data file

Testing the locked status of the data file and the volume on which it is located requires calls to the Operating System File Manager. Doing so for each database operation that modifies the data file would significantly affect the performance of the database engine.

It is your responsibility to test this status at the beginning of a working session. Usually, you do so in the On Startup Database Method of your database. In versions up to 3.2.5 of 4D and 1.2.5 of 4D Server, testing the locked status of the data file required the use of external routines that returned the File Manager attributes of the data file and the volume where it is located. Now, 4D and 4D Server simplify this test by signalling a locked data file each time you try to create a new record. If the data file or its volume are locked, the database engine generates the error -9999 No more space available to save the record.

The following code is an example of how to test the locked status of the data file:

```
   ` Is data locked project method
   ` Is data locked -> Boolean
   ` Is data locked -> True if locked or full
gError:=0
ON ERR CALL("ERROR HANDLING")
CREATE RECORD([Any File])
SAVE RECORD([Any File])
ON ERR CALL("")
If (gError=0)
   DELETE RECORD([Any File])
End if
$0:=(gError=-9999)
```

The ON ERR CALL method, named ERROR HANDLING, is listed here:

```
   ` ERROR HANDLING project method
gError:=Error
```

Then, in the On Startup Database Method of your database, you can write:

```
   ` STARTUP global procedure
   ` ...
If (Is data locked)
      ` Displays a dialog box explaining that:
      ` - the data file is locked,
      ` - or the volume on which the data file is locked or full,
      ` Whatever the case, you may want to use the database in read only:
      ` the data file may be located on a CD-ROM volume.
      ` So the dialog box may have a “Use in read only” button and a “Quit” button.
```

```
        If (bQuit=1)
            QUIT 4D ` Leave the database and check what's going on at the Finder level
        Else
                ` Set an interprocess flag to signal that the data file is locked
            <>gREADONLY:=True
                ` ... Continue Startup execution
        End if
    End if
```

If you authorize use of the database with a locked data file, be sure to restrict access to the operations that might modify the database contents. To do so, you can call your test function again or maintain an interprocess flag as in the previous example. For example, if you have a project method M_ADD_CUST that starts a process in which you add Customer records, you know in advance if this will be possible. If it is not, it is worthless to start the process. For example:

```
        ` M_ADD_CUST project method
    If (Can write data)
            ` Go ahead
        <>vlPID_CUST:=New process(...;...;...)
            ` ...
    End if
```

The method Can write data is listed here:

```
        ` Can write data project method
        ` Can write data -> Boolean
        ` Can write data -> True if data file is not locked
    $0:=True
    If (<>gREADONLY) ` or If (Is data locked)
        ALERT("This operation cannot be performed with a read only data file.")
        $0:=False
    End if
```

Important note: In order to preserve the database engine performance in 4D 3.2.5 and 4D Server 1.2.5 (and earlier versions), DO NOT check the locked status of the data file during each database operation that might modify data file contents. For example, if you are adding records in a transaction, remember that the data file is left untouched until you try to validate the transaction. Testing the locked status of the data while making modifications in a transaction would add needless testing overhead. Consequently, the locked status of the data file is tested only when you try to create a new record when not in a transaction. This includes adding or importing new records in the User environment. This also includes the commands ADD RECORD, SAVE RECORD (applied to a new record), and ARRAY TO SELECTION (that creates new records). This does not include the creation of new records in a transaction, or modification or deletion of existing records.

Note: Testing the locked status of the data file does not prevent you from continuing to test the other I/O errors that may occur while writing to the data file.

57 ASCII Codes

version 6.0

ASCII Code Tables

• The standard ASCII codes, 0 through 127, are common to Windows and Macintosh. These standard ASCII codes are listed in ASCII Codes 0..63 and ASCII Codes 64..127.

• The ASCII codes 128 through 255 are different on Windows and Macintosh. In order to maintain platform independence, the Windows version of 4th Dimension automatically converts ASCII codes from Windows to Macintosh ASCII maps when characters are entering the 4D environment (Data entry, Edit/Paste, Import, etc.) and from Macintosh to Windows ASCII maps when characters are leaving the 4D environment (Edit/Cut or Copy, Export, etc.).

The ASCII codes 128 through 255 are listed in ASCII Codes 128..191 and ASCII Codes 192..255.

Understanding ASCII Codes and 4th Dimension

On both Macintosh and Windows, the internal database engine and the 4D language work with the Macintosh extended ASCII set. When you enter data using the keyboard (adding records, editing procedures, etc.), 4th Dimension uses the internal Altura ASCII conversion scheme to convert what comes from the keyboard (expressed using the Windows set) to the Macintosh set. For example, to enter an “é”, you type ALT+0233, and 4th Dimension stores ASCII code 142 in the record. This is transparent to the end user, because when you create a search, you actually type (in the Search editor) the value for which you are looking. Therefore, the value that you typed (ALT+0233) is also translated into ASCII code 142, and you find the value.

The codes work the same when you type ALT+0233 in the Procedure editor. However, to look for a character using its ASCII code, you use the Macintosh ASCII code of the character.

For example:

```
QUERY (...; [MyFile]MyField="é") ` é is Alt+0233
```

is the same as:

```
QUERY (...;[MyFile]MyField=Char(142)) ` é is ASCII 142
```

See Also

Ascii, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

ASCII Codes 0..63

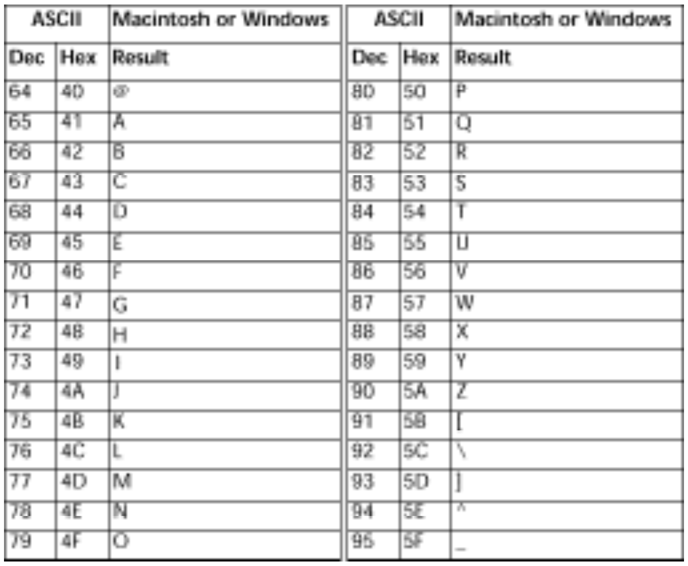
The standard ASCII codes (0 through 127) are common to Windows and Macintosh.

ASCII		Macintosh or Windows
Dec	Hex	Result
0	0	NUL
1	1	SOH
2	2	STX
3	3	ETX
4	4	EOT
5	5	ENQ
6	6	ACK
7	7	BEL
8	8	BS
9	9	HT
10	A	LF
11	B	VT
12	C	FF
13	D	CR
14	E	SO
15	F	SI
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
3D	1E	RS
31	1F	US

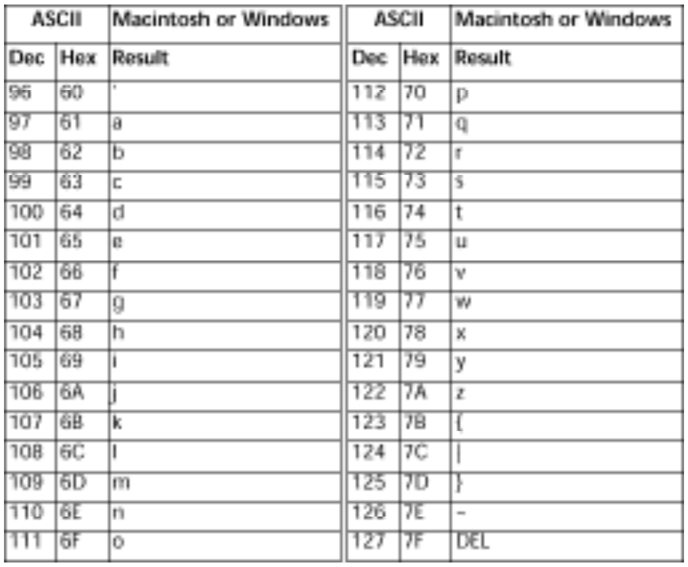
ASCII		Macintosh or Windows
Dec	Hex	Result
32	20	sp
33	21	!
34	22	"
35	F	SI
36	10	DLE
37	11	DC1
38	12	DC2
39	13	DC3
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII Codes 64..127

The standard ASCII codes (0 through 127) are common to Windows and Macintosh.



ASCII		Macintosh or Windows	ASCII		Macintosh or Windows
Dec	Hex	Result	Dec	Hex	Result
64	40	@	80	50	P
65	41	A	81	51	Q
66	42	B	82	52	R
67	43	C	83	53	S
68	44	D	84	54	T
69	45	E	85	55	U
70	46	F	86	56	V
71	47	G	87	57	W
72	48	H	88	58	X
73	49	I	89	59	Y
74	4A	J	90	5A	Z
75	4B	K	91	5B	[
76	4C	L	92	5C	\
77	4D	M	93	5D	]
78	4E	N	94	5E	^
79	4F	O	95	5F	_



ASCII		Macintosh or Windows	ASCII		Macintosh or Windows
Dec	Hex	Result	Dec	Hex	Result
96	60	`	112	70	p
97	61	a	113	71	q
98	62	b	114	72	r
99	63	c	115	73	s
100	64	d	116	74	t
101	65	e	117	75	u
102	66	f	118	76	v
103	67	g	119	77	w
104	68	h	120	78	x
105	69	i	121	79	y
106	6A	j	122	7A	z
107	6B	k	123	7B	{
108	6C	l	124	7C	|
109	6D	m	125	7D	}
110	6E	n	126	7E	~
111	6F	o	127	7F	DEL

ASCII Codes 128..191

The following tables list the characters displayed by 4th Dimension for each ASCII code, on Macintosh and Windows. In addition, the tables present the key combination required to produce each character, using a US keyboard.

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
144	90	ê	Option-i, e	ê	Alt+0234
145	91	ë	Option-u, e	ë	Alt+0235
146	92	í	Option-e, i	í	Alt+0237
147	93	ì	Option-`, i	ì	Alt+0236
148	94	î	Option-i, i	î	Alt+0238
149	95	ï	Option-u, i	ï	Alt+0239
150	96	ñ	Option-n, n	ñ	Alt+0241
151	97	ó	Option-e, o	ó	Alt+0243
152	98	ò	Option-`, o	ò	Alt+0242
153	99	ô	Option-i, o	ô	Alt+0244
154	9A	ö	Option-u, o	ö	Alt+0246
155	9B	õ	Option-n, o	õ	Alt+0245
156	9C	ú	Option-e, u	ú	Alt+0250
157	9D	ù	Option-`, u	ù	Alt+0249
158	9E	û	Option-i, u	û	Alt+0251
159	9F	ü	Option-u, u	ü	Alt+0252

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
128	80	Ä	Option-u, Shift-a	Ä	Alt+0196
129	81	Å	Option-Shift-a	Å	Alt+0197
130	82	Ç	Option-Shift-c	Ç	Alt+0199
131	83	É	Option-e, Shift-e	É	Alt+0201
132	84	Ñ	Option-n, Shift-n	Ñ	Alt+0209
133	85	Ö	Option-u, Shift-o	Ö	Alt+0214
134	86	Ü	Option-u, Shift-u	Ü	Alt+0220
135	87	á	Option-e, a	á	Alt+0225
136	88	à	Option-`, a	à	Alt+0224
137	89	â	Option-i, a	â	Alt+0226
138	8A	ä	Option-u, a	ä	Alt+0228
139	8B	ã	Option-n, a	ã	Alt+0227
140	8C	å	Option-a	å	Alt+0229
141	8D	ç	Option-c	ç	Alt+0231
142	8E	é	Option-e, e	é	Alt+0233
143	8F	è	Option-`, e	è	Alt+0232

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
160	A0	†	Shift-Option-7	▨	▨
161	A1	°	Shift-Option-7	°	Alt+0176
162	A2	¢	Option-4	¢	Alt+0162
163	A3	£	Option-3	£	Alt+0163
164	A4	§	Option-6	§	Alt+0167
165	A5	•	Option-8	▨	▨
166	A6	¶	Option-7	¶	Alt+0182
167	A7	ß	Option-s	ß	Alt+0223
168	A8	®	Option-r	®	Alt+0174
169	A9	©	Option-g	©	Alt+0169
170	AA	™	Option-2	™	Alt-0153
171	AB	´	Shift-Option-e	´	Alt+0145
172	AC	¨	Shift-Option-u	¨	Alt+0168
173	AD	≠	Option-=	▨	▨
174	AE	Æ	Shift-Option-"	Æ	Alt+0198
175	AF	Ø	Shift-Option-o	Ø	Alt+0216

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
176	B0	∞	Option-5	▨	▨
177	B1	±	Shift-Option-=	±	Alt+0177
178	B2	≤	Option-,	▨	▨
179	B3	≥	Option-. (period)	▨	▨
180	B4	¥	Option-y	¥	Alt+0165
181	B5	µ	Option-m	µ	Alt+0181
182	B6	∂	Option-d	▨	▨
183	B7	∑	Option-w	▨	▨
184	B8	∏	Shift-Option-p	▨	▨
185	B9	π	Option-p	▨	▨
186	BA	∫	Option-b	▨	▨
187	BB	ª	Option-9	ª	Alt+0170
188	BC	º	Option-0 (zero)	º	Alt+0186
189	BD	Ω	Option-z	▨	▨
190	BE	æ	Option-'	æ	Alt+0230
191	BF	ø	Option-o	ø	Alt+0248

Note: The cells in the Windows column that are greyed out denote characters that are not available on Windows or that are different from the Macintosh characters.

See Also

Char, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

ASCII Codes 192..255

The following tables list the characters displayed by 4th Dimension for each ASCII code, on Macintosh and Windows. In addition, the tables present the key combination required to produce each character, using a US keyboard.

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
192	C0	¿	Shift-Option-?	¿	Alt+0191
193	C1	¡	Option-1	¡	Alt+0161
194	C2	¬	Option-l (L)	¬	Alt+0172
195	C3	√	Option-v	▨	▨
196	C4	ƒ	Option-f	▨	▨
197	C5	≈	Option-x	▨	▨
198	C6	∆	Option-j	▨	▨
199	C7	«	Option-\	«	Alt+0171
200	C8	»	Shift-Option-\	»	Alt+0187
201	C9	…	Option-;	…	Alt+0133
202	CA	(space)	Spacebar	(space)	Alt+0160
203	CB	À	Option-`, Shift-a	À	Alt+0192
204	CC	Ã	Option-n, Shift-a	Ã	Alt+0195
205	CD	Õ	Option-n, Shift-o	Õ	Alt+0213
206	CE	Œ	Shift-Option-q	▨	▨
207	CF	œ	Option-q	▨	▨

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
208	D0	–	Option--(dash)	▨	▨
209	D1	—	Shift-Option-(dash)	▨	▨
210	D2	“	Option-[	“	Alt+0147
211	D3	”	Shift-Option-[	”	Alt+0148
212	D4	‘	Option-]	‘	Alt+0145
213	D5	’	Shift-Option-]	’	Alt+0146
214	D6	÷	Option-/	÷	Alt+0247
215	D7	◊	Shift-Option-v	▨	▨
216	D8	ÿ	Option-u, y	ÿ	Alt+0255
217	D9	Ÿ	Option-u, Shift-y	▨	▨
218	DA	⁄	Shift-Option-1	▨	▨
219	DB	¤	Shift-Option-2	¤	Alt+0164
220	DC	‹	Shift-Option-3	▨	▨
221	DD	›	Shift-Option-4	▨	▨
222	DE	ﬁ	Shift-Option-5	▨	▨
223	DF	ﬂ	Shift-Option-6	▨	▨

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
224	E0	‡	Shift-Option-7		
225	E1	·	Shift-Option-9	·	Alt+0183
226	E2	‚	Shift-Option-0		
227	E3	„	Shift-Option-w		
228	E4	‰	Shift-Option-r		
229	E5	Â	Option-i, Shift-a	Â	Alt+0194
230	E6	Ê	Option-i, Shift-e	Ê	Alt+0202
231	E7	Á	Option-e, Shift-a	Á	Alt+0193
232	E8	Ë	Option-u, Shift-e	Ë	Alt+0203
233	E9	È	Option-`, Shift-e	È	Alt+0200
234	EA	Í	Shift-Option-s	Í	Alt+0205
235	EB	Î	Shift-Option-d	Î	Alt+0206
236	EC	Ï	Shift-Option-f	Ï	Alt+0207
237	ED	Ì	Option-`, Shift-i	Ì	Alt+0204
238	EE	Ó	Shift-Option-h	Ó	Alt+0211
239	EF	Ô	Shift-Option-j	Ô	Alt+0212

ASCII		Macintosh		Windows	
Dec	Hex	Result (in Times)	Key Combination	Result (in Arial)	Key Combination
240	F0		Shift-Option-k	▨	▨
241	F1	Ò	Shift-Option-l (L)	Ò	Alt+0210
242	F2	Ú	Shift-Option-;	Ú	Alt+0218
243	F3	Û	Option-i, Shift-u	Û	Alt+0219
244	F4	Ù	Option-`, Shift-u	Ù	Alt+0217
245	F5	ı	Shift-Option-b	▨	▨
246	F6	ˆ	Shift-Option-i	▨	▨
247	F7	˜	Shift-Option-n	▨	▨
248	F8	¯	Shift-Option-,	¯	Alt+0175
249	F9	˘	Shift-Option-.	▨	▨
250	FA	˙	Option-h	▨	▨
251	FB	˚	Option-k	▨	▨
252	FC	¸	Shift-Option-z	¸	Alt+0184
253	FD	˝	Shift-Option-g	▨	▨
254	FE	˛	Shift-Option-x	▨	▨
255	FF	ˇ	Shift-Option-t	▨	▨

Note: The cells in the Windows column that are greyed out denote characters that are not available on Windows or that are different from the Macintosh characters.

See Also

Ascii, ISO to Mac, Mac to ISO, Mac to Win, ON EVENT CALL, Win to Mac.

Function Key Codes

version 6.0

4th Dimension returns values for Function keys in the KeyCode system variable, which is used within project methods installed by the ON EVENT CALL command. These project methods are used to catch events. The values for Function keys are not based on ASCII codes. They are:

Function key	KeyCode
F1	-122
F2	-120
F3	-99
F4	-118
F5	-96
F6	-97
F7	-98
F8	-100
F9	-101
F10	-109
F11	-103
F12	-111
F13	-105
F14	-107
F15	-113

Reminder: The KeyCode system variable is to be used in a project method installed using ON EVENT CALL.

In addition to the function keys, the following table lists the values returned in KeyCode when you press one of the common keys, such as Return or Enter.

Code	Key
3	Enter
13	Return
8	Backspace or Delete
9	Tab
27	Escape or Clear
127	Del
5	Help
1	Home
4	End
11	Page Up
12	Page Down
28	Left Arrow
29	Right Arrow
30	Up Arrow
31	Down Arrow

58 Command Syntax

Command Syntax by Name

Command Syntax

version 6.0.2

A

ABORT
Abs (number) → Number
ACCEPT
ACCUMULATE (data{; data2; ...; dataN})
ACI folder → String
Activated → Boolean
ADD DATA SEGMENT
ADD RECORD ({table}{; }{*})
ADD SUBRECORD (subtable; form{; *})
Add to date (date; years; months; days) → Date
ADD TO SET ({table; }set)
After → Boolean
ALERT (message{; ok button title})
ALL RECORDS {(table)}
ALL SUBRECORDS (subtable)
Append document (document{; type}) → DocRef
APPEND MENU ITEM (menu; itemText{; process})
APPEND TO CLIPBOARD (dataType; data)
APPEND TO LIST (list; itemText; itemRef{; sublist{; expanded}})
Application file → String
Application type → Long Integer
Application version {(*)} → String
APPLY TO SELECTION ({table; }statement)
APPLY TO SUBSELECTION (subtable; statement)
Arctan (number) → Number
ARRAY BOOLEAN (arrayName; size{; size2})
ARRAY DATE (arrayName; size{; size2})
ARRAY INTEGER (arrayName; size{; size2})
ARRAY LONGINT (arrayName; size{; size2})
ARRAY PICTURE (arrayName; size{; size2})
ARRAY POINTER (arrayName; size{; size2})
ARRAY REAL (arrayName; size{; size2})
ARRAY STRING (strLen; arrayName; size{; size2})
ARRAY TEXT (arrayName; size{; size2})
ARRAY TO LIST (array; list{; itemRefs})
ARRAY TO SELECTION (array; field{; array2; field2; ...; arrayN; fieldN})
ARRAY TO STRING LIST (strings; resID{; resFile})
Ascii (character) → Number

AUTOMATIC RELATIONS (one; many)
Average (series) → Number

B

BEEP
Before → Boolean
Before selection {(table)} → Boolean
Before subselection (subtable) → Boolean
BLOB PROPERTIES (blob; compressed{; expandedSize{; currentSize}})
BLOB size (blob) → Number
BLOB TO DOCUMENT (document; blob{; *})
BLOB to integer (blob; ordreOctet{; offset}) → Number
BLOB to list (blob{; offset}) → ListRef
BLOB to longint (blob; byteOrder{; offset}) → Number
BLOB to real (blob; realFormat{; offset})
BLOB to text (blob; formatTexte{; offset{; longueurTexte}})
BLOB TO VARIABLE (blob; variable{; offset})
BREAK LEVEL (level{; pageBreak})
BRING TO FRONT (process)
BUTTON TEXT ({*; }object; buttonText)

C

CALL PROCESS (process)
CANCEL
CANCEL TRANSACTION
Caps lock down → Boolean
CHANGE ACCESS
CHANGE PASSWORD (password)
Change string (source; newChars; where) → String
Char (asciiCode) → String
CLEAR CLIPBOARD
CLEAR LIST (list{; *})
CLEAR NAMED SELECTION (name)
CLEAR SEMAPHORE (semaphore)
CLEAR SET (set)
CLEAR VARIABLE (variable)
CLOSE DOCUMENT (docRef)
CLOSE RESOURCE FILE (resFile)
CLOSE WINDOW {(extWindowRef)}
Command name (command) → String
Compiled application → Boolean
COMPRESS BLOB (blob{; compression})

COMPRESS PICTURE (picture; method; quality)
COMPRESS PICTURE FILE (document; method; quality)
CONFIRM (message{; OK button title{; cancel button title}})
COPY ARRAY (source; destination)
COPY BLOB (srcBLOB; dstBLOB; srcOffset; dstOffset; len)
COPY DOCUMENT (sourceName; destinationName{; *})
Copy list (list) → ListRef
COPY NAMED SELECTION ({table; }name)
COPY SET (srcSet; dstSet)
Cos (number) → Number
Count fields (tableNum | tablePtr) → Number
Count list items (List) → Long
Count menu items (menu{; process}) → Number
Count menus {(process)} → Number
Count parameters → Number
Count screens → Number
Count tables → Number
Count tasks → Integer
Count user processes → Integer
Count users → Integer
Create document (document{; type}) → DocRef
CREATE EMPTY SET ({table; }set)
CREATE FOLDER (folderPath)
CREATE RECORD {(table)}
CREATE RELATED ONE (field)
Create resource file (resFilename{; fileType}) → DocRef
CREATE SET ({table; }set)
CREATE SUBRECORD (subtable)
Current date {(*)} → Date
Current default table → Pointer
Current form page → Number
Current form table → Pointer
Current machine → String
Current machine owner → String
Current process → Number
Current time {(*)} → Time
Current user → String
CUT NAMED SELECTION ({table; }name)
C_BLOB ({method; }variable{; variable2; ...; variableN})
C_BOOLEAN ({method; }variable{; variable2; ...; variableN})
C_DATE ({method; }variable{; variable2; ...; variableN})
C_GRAPH ({method; }variable{; variable2; ...; variableN})
C_INTEGER ({method; }variable{; variable2; ...; variableN})

C_LONGINT ({method; }variable{; variable2; ...; variableN})
C_PICTURE ({method; }variable{; variable2; ...; variableN})
C_POINTER ({method; }variable{; variable2; ...; variableN})
C_REAL ({method; }variable{; variable2; ...; variableN})
C_STRING ({method; }size; variable{; variable2; ...; variableN})
C_TEXT ({method; }variable{; variable2; ...; variableN})
C_TIME ({method; }variable{; variable2; ...; variableN})

D

Data file {(segment)} → String
DATA SEGMENT LIST (Segments)
Database event → Number
Date (dateString) → Date
Day number (date) → Number
Day of (date) → Number
Deactivated → Boolean
Dec (number) → Number
DEFAULT TABLE (table)
DELAY PROCESS (process; duration)
DELETE DOCUMENT (document)
DELETE ELEMENT (array; where{; howMany})
DELETE FROM BLOB (blob; offset; len)
DELETE LIST ITEM (list; itemRef | *{; *})
DELETE MENU ITEM (menu; menuItem{; process})
DELETE RECORD {(table)}
DELETE RESOURCE (resType; resID{; resFile})
DELETE SELECTION {(table)}
Delete string (source; where; numChars) → String
DELETE SUBRECORD (subtable)
DELETE USER (UserID)
DIALOG ({table; }form)
DIFFERENCE (set; subtractSet; resultSet)
DISABLE BUTTON ({*; }object)
DISABLE MENU ITEM (menu; menuItem{; process})
DISPLAY RECORD {(table)}
DISPLAY SELECTION ({table}{; *}{; *})
DISTINCT VALUES (field; array)
Document creator (document) → String
DOCUMENT LIST (pathname; documents)
DOCUMENT TO BLOB (document; blob{; *})
Document type (document) → String
DRAG AND DROP PROPERTIES (srcObject; srcElement; srcProcess)
DRAG WINDOW
Drop position → Number

DUPLICATE RECORD {(table)}
During → Boolean

E

EDIT ACCESS
ENABLE BUTTON ({*; }object)
ENABLE MENU ITEM (menu; menuItem{; process})
End selection {(table)} → Boolean
End subselection (subtable) → Boolean
ERASE WINDOW {(window)}
EXECUTE (statement)
Execute on server (procedure; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number
Exp (number) → Number
EXPAND BLOB (blob)
EXPORT DIF ({table; }document)
EXPORT SYLK ({table; }document)
EXPORT TEXT (table; document)

F

False → Boolean
Field (tableNum | fieldPtr{; fieldNum}) → Number | Pointer
Field name ({tableNum; }fieldNum | fieldPtr) → String
FILTER EVENT
FILTER KEYSTROKE (filteredChar)
Find in array (array; value{; start}) → Number
Find window (left; top{; windowPart}) → WinRef
FIRST PAGE
FIRST RECORD {(table)}
FIRST SUBRECORD (subtable)
FLUSH BUFFERS
FOLDER LIST (pathname; directories)
FONT ({*; }object; font)
FONT LIST (fonts)
Font name (fontNumber) → String
Font number (fontName) → Number
FONT SIZE ({*; }object; size)
FONT STYLE ({*; }object; styles)
Form event → Number
Frontmost process {(*)} → Integer
Frontmost window {(*)} → WinRef

G

Gestalt (selector; value) → Number
GET CLIPBOARD (dataType; data)
Get document position (docRef) → Number
GET DOCUMENT PROPERTIES (document; locked; invisible; created on; created at; modified on; modified at)
Get document size (document{; *}) → Number
GET FIELD PROPERTIES ({tableNum; }fieldNum | fieldPtr; fieldType{; fieldLen{; indexed}})
GET GROUP LIST (groupNames; groupNumbers)
GET GROUP PROPERTIES (groupID; name; owner{; members})
GET HIGHLIGHT (area; startSel; endSel)
GET ICON RESOURCE (resID; resData{; fileRef})
Get indexed string (resID; strID{; resFile}) → String
GET LIST ITEM (list; itemPos; itemRef; itemText{; sublist{; expanded}})
GET LIST ITEM PROPERTIES (list; itemRef; enterable{; styles{; icon}})
GET LIST PROPERTIES (list; appearance{; icon{; lineHeight}})
Get menu item (menu; menuItem{; process}) → String
Get menu item key (menu; menuItem{; process}) → Number
Get menu item mark (menu; menuItem{; process}) → String
Get menu item style (menu; menuItem{; process}) → Number
Get menu title (menu{; process}) → String
GET MOUSE (mouseX; mouseY; mouseButton{; *})
GET PICTURE FROM CLIPBOARD (picture)
GET PICTURE FROM LIBRARY (picRef; picture)
GET PICTURE RESOURCE (resID; resData{; resFile})
Get platform interface → Number
Get pointer (varName) → Pointer
GET PROCESS VARIABLE (process; srcVar; dstVar{; srcVar2; dstVar2; ...; srcVarN; dstVarN})
GET RESOURCE (resType; resID; resData{; resFile})
Get resource name (resType; resID{; resFile}) → String
Get resource properties (resType; resID{; resFile}) → Number
Get string resource (resID{; resFile}) → String
Get text from clipboard → String
Get text resource (resID{; resFile}) → Text
GET USER LIST (userNames; userNumbers)
GET USER PROPERTIES (userID; name; startup; password; nbLogin; lastLogin{; memberships})
GET WINDOW RECT (left; top; right; bottom{; window})
Get window title {(window)} → String
GOTO AREA (area)
GOTO PAGE (pageNumber)
GOTO RECORD ({table; }record)
GOTO SELECTED RECORD ({table; }record)
GOTO XY (x; y)

GRAPH (graphArea; graphNumber; xLabels; yElements{; yElements2; ...; yElementsN})
GRAPH SETTINGS (graph; xmin; xmax; ymin; ymax; xprop; xgrid; ygrid; title{; title2; ...; titleN})
GRAPH TABLE ({table; }graphType; x field; y field{; y field2; ...; y fieldN})

H

HIDE MENU BAR
HIDE PROCESS (process)
HIDE TOOL BAR
HIGHLIGHT TEXT (area; startSel; endSel)

I

IDLE
IMPORT DIF ({table; }document)
IMPORT SYLK ({table; }document)
IMPORT TEXT ({table; }document)
In break → Boolean
In footer → Boolean
In header → Boolean
In transaction → Boolean
INPUT FORM ({table; }form{; *})
INSERT ELEMENT (array; where{; howMany})
INSERT IN BLOB (blob; offset; len{; filler})
INSERT LIST ITEM (list; beforeItemRef | *; itemText; itemRef{; sublist{; expanded}})
INSERT MENU ITEM (menu; beforeItem; itemText{; process})
Insert string (source; what; where) → String
Int (number) → Number
INTEGER TO BLOB (integer; blob; byteOrder{; offset | *})
INTERSECTION (set1; set2; resultSet)
INVERT BACKGROUND (textVar | textField)
Is a list (list) → Boolean
Is a variable (aPointer) → Boolean
Is in set (set) → Boolean
Is user deleted (userNumber) → Boolean
ISO to Mac (text) → String

K

Keystroke → string

L

Last object → Pointer
LAST PAGE
LAST RECORD {(table)}
LAST SUBRECORD (subtable)
Length (string) → Number
Level → Number
List item parent (list; itemRef) → Number
List item position (list; itemRef) → Number
LIST TO ARRAY (list; array{; itemRefs})
LIST TO BLOB (list; blob{; offset | *})
LOAD COMPRESS PICTURE FROM FILE (document; method; quality; picture)
Load list (listName) → ListRef
LOAD RECORD {(table)}
LOAD SET ({table; }set; document)
LOAD VARIABLES (document; variable{; variable2; ...; variableN})
Locked {(table)} → Boolean
LOCKED ATTRIBUTES ({table; }process; user; machine; processName)
Log (number) → Number
LONGINT TO BLOB (longInt; blob; byteOrder{; offset | *})
Lowercase (string) → String

M

Mac to ISO (text) → String
Mac to Win (text) → String
Macintosh command down → Boolean
Macintosh control down → Boolean
Macintosh option down → Boolean
MAP FILE TYPES (macOS; windows; context)
Max (series) → Number
MENU BAR (menuBar{; process}{; *})
Menu bar height → Number
Menu bar screen → Number
Menu selected → Number
MESSAGE (message)
MESSAGES OFF
MESSAGES ON
Milliseconds → Number
Min (series) → Number
Mod (number1; number2) → Number

Modified (field) → Boolean
Modified record {(table)} → Boolean
MODIFY RECORD ({table}{; }{*})
MODIFY SELECTION ({table}{; *}{; *})
MODIFY SUBRECORD (subtable; form{; *})
Month of (date) → Number
MOVE DOCUMENT (srcPathname; dstPathname)

N

New list → ListRef
New process (method; stack{; name{; param{; param2; ...; paramN}{; *}}}) → Number
NEXT PAGE
NEXT RECORD {(table)}
NEXT SUBRECORD (subtable)
Next window (window) → Number
Nil (aPointer) → Boolean
NO TRACE
Not (boolean) → Boolean
Num (expression) → Number

O

Old (field) → Expression
OLD RELATED MANY (field)
OLD RELATED ONE (field)
ON ERR CALL (errorMethod)
ON EVENT CALL (eventMethod{; processName})
ON SERIAL PORT CALL (serialMethod{; process})
ONE RECORD SELECT {(table)}
Open document (document{; fileType}) → DocRef
Open external window (left; top; right; bottom; type; title; plugInArea) → Number
Open resource file (resFilename{; fileType}) → DocRef
Open window (left; top; right; bottom{; type{; title{; controlMenuBox}}}){ → WinRef }
ORDER BY ({table}{; field}{; > or <}{; field2; > or <2; ...; fieldN; > or <N})
ORDER BY FORMULA ({table}{; expression}{; > or <}{; expression2; > or <2; ...; expressionN;
> or <N})
ORDER SUBRECORDS BY (subtable; subfield{; > or <}{; subfield2; > or <2; ...; subfieldN;
> or <N})
OUTPUT FORM ({table; }form)
Outside call → Boolean

P

PAGE BREAK {(* | >)}
PAGE SETUP ({table; }form)
PAUSE PROCESS (process)
PICTURE LIBRARY LIST (picRefs; picNames)
PICTURE PROPERTIES (picture; width; height{; hOffset{; vOffset{; mode}}})
Picture size (picture) → Number
PLATFORM PROPERTIES (platform; system; machine)
PLAY (objectName{; channel})
POP RECORD {(table)}
Pop up menu (contents{; default}) → Number
Position (find; string) → Number
POST CLICK (mouseX; mouseY{; process}{; *})
POST EVENT (what; message; when; mouseX; mouseY; modifiers{; process})
POST KEY (code{; modifiers{; process}})
PREVIOUS PAGE
PREVIOUS RECORD {(table)}
PREVIOUS SUBRECORD (subtable)
PRINT FORM ({table; }form)
PRINT LABEL ({table}{; document}{; *})
PRINT RECORD ({table}{; }{*})
PRINT SELECTION ({table}{; }{*})
PRINT SETTINGS
Printing page → Number
Process number (name) → Number
PROCESS PROPERTIES (process; procName; procState; procTime{; procVisible})
Process state (process) → Number
PUSH RECORD {(table)}

Q

QUERY ({table}{; queryArgument}{; *})
QUERY BY EXAMPLE {(table)}
QUERY BY FORMULA ({table}{; }{queryFormula})
QUERY SELECTION ({table}{; queryArgument}{; *})
QUERY SELECTION BY FORMULA ({table}{; }{queryFormula})
QUERY SUBRECORDS (subtable; queryFormula)
QUIT 4D

R

Random → Number
READ ONLY {(table | *)}
Read only state {(table)} → Boolean

READ WRITE {(table | *)}
REAL TO BLOB (real; blob; realFormat{; offset | *})
RECEIVE BUFFER (receiveVar)
RECEIVE PACKET ({docRef; }receiveVar; stopChar | numChars)
RECEIVE RECORD {(table)}
RECEIVE VARIABLE (variable)
Record number {(table)} → Number
Records in selection {(table)} → Number
Records in set (set) → Number
Records in subselection (subtable) → Number
Records in table {(table)} → Number
REDRAW (object)
REDRAW LIST (list)
REDRAW WINDOW {(window)}
REDUCE SELECTION ({table; }number)
REJECT {(field)}
RELATE MANY (oneTable | Field)
RELATE MANY SELECTION (field)
RELATE ONE (manyTable | Field{; choiceField})
RELATE ONE SELECTION (manyTable; oneTable)
REMOVE FROM SET ({table; }set)
REMOVE PICTURE FROM LIBRARY (picRef)
Replace string (source; oldString; newString{; howMany}) → String
REPORT ({table; }document{; *})
Request (message{; default response{; OK button title{; Cancel button title}}}) → String
RESOLVE POINTER (pointer; varName; tableNum; fieldNum)
RESOURCE LIST (resType; resIDs; resNames{; resFile})
RESOURCE TYPE LIST (resTypes{; resFile})
RESUME PROCESS (process)
Round (round; places) → Number

S

SAVE LIST (list; listName)
SAVE OLD RELATED ONE (field)
SAVE PICTURE TO FILE (document; picture)
SAVE RECORD {(table)}
SAVE RELATED ONE (field)
SAVE SET (set; document)
SAVE VARIABLES (document; variable{; variable2; ...; variableN})
SCAN INDEX (field; number; > or <)
SCREEN COORDINATES (left; top; right; bottom{; screen})
SCREEN DEPTH (depth; color{; screen})
Screen height {(*)} → Number
Screen width {(*)} → Number

SEARCH BY INDEX
SELECT LIST ITEM (list; itemPos)
SELECT LIST ITEM BY REFERENCE (list; itemRef)
SELECT LOG FILE (logFile)
Selected list item (list) → Long
Selected record number {(table)} → Number
SELECTION TO ARRAY (field | table; array{; field2 | table2; array2; ...; fieldN | tableN; arrayN})
Self → Pointer
Semaphore (semaphore) → Boolean
SEND HTML FILE (htmlFile)
SEND PACKET ({docRef; }packet)
SEND RECORD {(table)}
SEND VARIABLE (variable)
Sequence number {(table)} → Number
SET ABOUT (itemText; method)
SET BLOB SIZE (blob; size{; filler})
SET CHANNEL (port | operation{; settings | document})
SET CHOICE LIST ({*; }object; list)
SET COLOR ({*; }object; color)
SET CURSOR {(cursor)}
SET DEFAULT CENTURY (century{; pivotYear})
SET DOCUMENT CREATOR (document; fileCreator)
SET DOCUMENT POSITION (docRef; offset{; anchor})
SET DOCUMENT PROPERTIES (document; locked; invisible; created on; created at;
modified on; modified at)
SET DOCUMENT SIZE (document; size)
SET DOCUMENT TYPE (document; fileType)
SET ENTERABLE ({*; }entryArea; enterable)
SET FIELD TITLES (table | subtable; fieldTitles; fieldNumbers)
SET FILTER ({*; }object; entryFilter)
SET FORMAT ({*; }object; displayFormat)
SET GROUP PROPERTIES (groupID; name; owner{; menbers})
SET HTML ROOT (pathnameHTML)
SET INDEX (field; index{; *})
SET LIST ITEM (list; itemRef; newItemText; newItemRef{; sublist{; expanded}})
SET LIST ITEM PROPERTIES (list; itemRef; enterable; styles; icon)
SET LIST PROPERTIES (list; appearance{; icon{; lineHeight}})
SET MENU ITEM (menu; menuItem; itemText{; process})
SET MENU ITEM KEY (menu; menuItem; itemKey{; process})
SET MENU ITEM MARK (menu; item; mark{; process})
SET MENU ITEM STYLE (menu; menuItem; itemStyle{; process})
SET PICTURE RESOURCE (resID; resData{; resFile})
SET PICTURE TO CLIPBOARD (picture)
SET PICTURE TO LIBRARY (picture; picRef; picName)
SET PLATFORM INTERFACE (interface)
SET PRINT PREVIEW (preview)

SET PROCESS VARIABLE (process; dstVar; expr{; dstVar2; expr2; ...; dstVarN; exprN})
SET QUERY DESTINATION (destinationType{; destinationObject})
SET QUERY LIMIT (limit)
SET REAL COMPARISON LEVEL (epsilon)
SET RESOURCE (resType; resID; resData{; resFile})
SET RESOURCE NAME (resType; resID; resName{; resFile})
SET RESOURCE PROPERTIES (resType; resID; resAttr{; resFile})
SET RGB COLOR ({*; }object; foregroundColor; backgroundColor)
SET SCREEN DEPTH (depth; color{; screen})
SET STRING RESOURCE (resID; resData{; resFile})
SET TABLE TITLES (tableTitles; tableNumbers)
SET TEXT RESOURCE (resID; resData{; resFile})
SET TEXT TO CLIPBOARD (text)
SET TIMEOUT (seconds)
SET USER PROPERTIES (userID; name; startup; password; nbLogin; lastLogin{; memberships})
SET VISIBLE ({*; }object; visible)
SET WEB DISPLAY LIMITS (numberRecords{; numberPages{; picRef}})
SET WEB TIMEOUT (timeout)
SET WINDOW RECT (left; top; right; bottom{; window})
SET WINDOW TITLE (title{; window})
Shift down → Boolean
SHOW MENU BAR
SHOW PROCESS (process)
SHOW TOOL BAR
Sin (number) → Number
Size of array (array) → Number
SORT ARRAY (array{; array2; ...; arrayN}{; > or <})
SORT BY INDEX
SORT LIST (list{; > or <})
Square root (number) → Number
START TRANSACTION
START WEB SERVER
Std deviation (series) → Number
STOP WEB SERVER
String (expression{; format}) → String
STRING LIST TO ARRAY (resID; strings{; resFile})
Structure file → String
SUBSELECTION TO ARRAY (start; end; field | table; array{; field2 | table2; array2; ...;
fieldN | tableN; arrayN})
Substring (source; firstChar{; numChars}) → String
Subtotal (data{; pageBreak}) → Number
Sum (series) → Number
Sum squares (series) → Number
System folder → String

T

Table (tableNum | aPtr) → Pointer | Number
Table name (tableNum | tablePtr) → String
Tan (number) → Number
Temporary folder → String
Test clipboard (dataType) → Number
Test path name (pathname) → Number
TEXT TO BLOB (texte; blob; formatTexte{; offset | *})
Tickcount → Number
Time (timeString) → Time
Time string (seconds) → String
TRACE
Trigger level → Number
TRIGGER PROPERTIES (triggerLevel; dbEvent; tableNum; recordNum)
True → Boolean
Trunc (number; places) → Number
Type (fieldVar) → Number

U

Undefined (variable) → Boolean
UNION (set1; set2; resultSet)
UNLOAD RECORD {(table)}
Uppercase (chaîne) → String
USE ASCII MAP (map | *; mapInOut)
USE NAMED SELECTION (name)
USE SET (set)
User in group (user; group) → Boolean

V

VALIDATE TRANSACTION
VARIABLE TO BLOB (variable; blob{; offset | *})
VARIABLE TO VARIABLE (process; dstVar; srcVar{; dstVar2; srcVar2; ...; dstVarN; srcVarN})
Variance (series) → Number
Version type → Long Integer
VOLUME ATTRIBUTES (volume; size; used; free)
VOLUME LIST (volumes)

W

Win to Mac (text) → String
Window kind {(window)}
WINDOW LIST (windows{; *})
Window process {(window)} → Number
Windows Alt down → Boolean
Windows Ctrl down → Boolean

Y

Year of (date) → Number

Constants

4D Environment

Constant	Type	Value
4D Client	Long Integer	4
4D Engine	Long Integer	1
4D First	Long Integer	6
4D Runtime	Long Integer	2
4D Runtime Classic	Long Integer	3
4D Server	Long Integer	5
4th Dimension	Long Integer	0
Demo Version	Long Integer	1
Full Version	Long Integer	0

ASCII Codes

Constant	Type	Value
ACK ASCII code	Long Integer	6
At sign	Long Integer	64
Backspace	Long Integer	8
BEL ASCII code	Long Integer	7
BS ASCII code	Long Integer	8
CAN ASCII code	Long Integer	24
Carriage return	Long Integer	13
CR ASCII code	Long Integer	13
DC1 ASCII code	Long Integer	17
DC2 ASCII code	Long Integer	18
DC3 ASCII code	Long Integer	19
DC4 ASCII code	Long Integer	20
DEL ASCII code	Long Integer	127
DLE ASCII code	Long Integer	16
Double quote	Long Integer	34
EM ASCII code	Long Integer	25
ENQ ASCII code	Long Integer	5
Enter	Long Integer	3
EOT ASCII code	Long Integer	4
ESC ASCII code	Long Integer	27
Escape	Long Integer	27
ETB ASCII code	Long Integer	23
ETX ASCII code	Long Integer	3
FF ASCII code	Long Integer	12
FS ASCII code	Long Integer	28
GS ASCII code	Long Integer	29
HT ASCII code	Long Integer	9
LF ASCII code	Long Integer	10
Line feed	Long Integer	10
NAK ASCII code	Long Integer	21
NBSP	Long Integer	202
NUL ASCII code	Long Integer	0
Period	Long Integer	46
Quote	Long Integer	39
RS ASCII code	Long Integer	30

ASCII Codes (continued)

Constant	**Type**	**Value**
SI ASCII code	Long Integer	15
SO ASCII code	Long Integer	14
SOH ASCII code	Long Integer	1
SP ASCII code	Long Integer	32
Space	Long Integer	32
STX ASCII code	Long Integer	2
SUB ASCII code	Long Integer	26
SYN ASCII code	Long Integer	22
Tab	Long Integer	9
US ASCII code	Long Integer	31
VT ASCII code	Long Integer	11

BLOB

Constant	Type	Value
C string	Long Integer	0
Compact compression mode	Long Integer	1
Extended real format	Long Integer	1
Fast compression mode	Long Integer	2
Is not compressed	Long Integer	0
Macintosh byte ordering	Long Integer	1
Macintosh double real format	Long Integer	2
Native byte ordering	Long Integer	0
Native real format	Long Integer	0
Pascal string	Long Integer	1
PC byte ordering	Long Integer	2
PC double real format	Long Integer	3
Text with length	Long Integer	2
Text without length	Long Integer	3

Clipboard

Constant	Type	Value
No such data in clipboard	Long Integer	-102
Picture data	String	PICT
Text data	String	TEXT

Colors

Constant	Type	Value
Black	Long Integer	15
Blue	Long Integer	6
Brown	Long Integer	13
Dark Blue	Long Integer	5
Dark Brown	Long Integer	10
Dark Green	Long Integer	9
Dark Grey	Long Integer	11
Green	Long Integer	8
Grey	Long Integer	14
Light Blue	Long Integer	7
Light Grey	Long Integer	12
Orange	Long Integer	2
Purple	Long Integer	4
Red	Long Integer	3
White	Long Integer	0
Yellow	Long Integer	1

Communications

Constant	Type	Value
Data bits 5	Long Integer	0
Data bits 6	Long Integer	2048
Data bits 7	Long Integer	1024
Data bits 8	Long Integer	3072
MacOS Printer Port	Long Integer	0
MacOS Serial Port	Long Integer	1
Parity Even	Long Integer	12288
Parity None	Long Integer	0
Parity Odd	Long Integer	4096
Protocol DTR	Long Integer	30
Protocol None	Long Integer	0
Protocol XONXOFF	Long Integer	20
Speed 115200	Long Integer	1022
Speed 1200	Long Integer	94
Speed 1800	Long Integer	62
Speed 19200	Long Integer	4
Speed 230400	Long Integer	1021
Speed 2400	Long Integer	46
Speed 300	Long Integer	380
Speed 3600	Long Integer	30
Speed 4800	Long Integer	22
Speed 57600	Long Integer	0
Speed 600	Long Integer	189
Speed 7200	Long Integer	14
Speed 9600	Long Integer	10
Stop bits One	Long Integer	16384
Stop bits One and a half	Long Integer	-32768
Stop bits Two	Long Integer	-16384

Database Engine

Constant	Type	Value
Is new record	Long Integer	-3
No current record	Long Integer	-1

Database Events

Constant	Type	Value
Delete Record Event	Long Integer	3
Load Record Event	Long Integer	4
Save Existing Record Event	Long Integer	2
Save New Record Event	Long Integer	1

Date Display Formats

Constant	Type	Value
Abbr Month Date	Long Integer	6
Abbreviated	Long Integer	2
Long	Long Integer	3
MM DD YYYY	Long Integer	4
MM DD YYYY Forced	Long Integer	7
Month Date Year	Long Integer	5
Short	Long Integer	1

Days and Months

Constant	Type	Value
April	Long Integer	4
August	Long Integer	8
December	Long Integer	12
February	Long Integer	2
Friday	Long Integer	6
January	Long Integer	1
July	Long Integer	7
June	Long Integer	6
March	Long Integer	3
May	Long Integer	5
Monday	Long Integer	2
November	Long Integer	11
October	Long Integer	10
Saturday	Long Integer	7
September	Long Integer	9
Sunday	Long Integer	1
Thursday	Long Integer	5
Tuesday	Long Integer	3
Wednesday	Long Integer	4

Events (Modifiers)

Constant	Type	Value
Activate window bit	Long Integer	0
Activate window mask	Long Integer	1
Caps Lock key bit	Long Integer	10
Caps Lock key mask	Long Integer	1024
Command key bit	Long Integer	8
Command key mask	Long Integer	256
Control key bit	Long Integer	12
Control key mask	Long Integer	4096
Mouse button bit	Long Integer	7
Mouse button mask	Long Integer	128
Option key bit	Long Integer	11
Option key mask	Long Integer	2048
Right control key bit	Long Integer	15
Right control key mask	Long Integer	32768
Right option key bit	Long Integer	14
Right option key mask	Long Integer	16384
Right shift key bit	Long Integer	13
Right shift key mask	Long Integer	8192
Shift key bit	Long Integer	9
Shift key mask	Long Integer	512

Events (What)

Constant	Type	Value
Activate event	Long Integer	8
Auto key event	Long Integer	5
Disk event	Long Integer	7
Key down event	Long Integer	3
Key up event	Long Integer	4
Mouse down event	Long Integer	1
Mouse up event	Long Integer	2
Null event	Long Integer	0
Operating system event	Long Integer	15
Update event	Long Integer	6

Expressions

Constant	Type	Value
MAXINT	Long Integer	32767
MAXLONG	Long Integer	2147483647
MAXTEXTLEN	Long Integer	32000

Field and Variable Types

Constant	Type	Value
Array 2D	Long Integer	13
Boolean array	Long Integer	22
Date array	Long Integer	17
Integer array	Long Integer	15
Is Alpha Field	Long Integer	0
Is BLOB	Long Integer	30
Is Boolean	Long Integer	6
Is Date	Long Integer	4
Is Integer	Long Integer	8
Is LongInt	Long Integer	9
Is Picture	Long Integer	3
Is Pointer	Long Integer	23
Is Real	Long Integer	1
Is String Var	Long Integer	24
Is Subtable	Long Integer	7
Is Text	Long Integer	2
Is Time	Long Integer	11
Is Undefined	Long Integer	5
LongInt array	Long Integer	16
Picture array	Long Integer	19
Pointer array	Long Integer	20
Real array	Long Integer	14
String array	Long Integer	21
Text array	Long Integer	18

Find window

Constant	Type	Value
In contents	Long Integer	3
In drag	Long Integer	4
In go away	Long Integer	6
In grow	Long Integer	5
In menu bar	Long Integer	1
In system window	Long Integer	2
In zoom box	Long Integer	8

Font Styles

Constant	Type	Value
Bold	Long Integer	1
Condensed	Long Integer	32
Extended	Long Integer	64
Italic	Long Integer	2
Outline	Long Integer	8
Plain	Long Integer	0
Shadow	Long Integer	16
Underline	Long Integer	4

Form Events

Constant	Type	Value
On Activate	Long Integer	11
On Clicked	Long Integer	4
On Close Box	Long Integer	22
On Close Detail	Long Integer	26
On Data Change	Long Integer	20
On Deactivate	Long Integer	12
On Display Detail	Long Integer	8
On Double Clicked	Long Integer	13
On Drag Over	Long Integer	21
On Drop	Long Integer	16
On External Area	Long Integer	19
On Getting Focus	Long Integer	15
On Keystroke	Long Integer	17
On Load	Long Integer	1
On Losing Focus	Long Integer	14
On Menu Selected	Long Integer	18
On Open Detail	Long Integer	25
On Outside Call	Long Integer	10
On Printing Break	Long Integer	6
On Printing Detail	Long Integer	23
On Printing Footer	Long Integer	7
On Printing Header	Long Integer	5
On Unload	Long Integer	24
On Validate	Long Integer	3

Function Keys

Constant	Type	Value
Backspace Key	Long Integer	8
Down Arrow Key	Long Integer	31
End Key	Long Integer	4
Enter Key	Long Integer	3
Escape Key	Long Integer	27
F1 Key	Long Integer	-122
F10 Key	Long Integer	-109
F11 Key	Long Integer	-103
F12 Key	Long Integer	-111
F13 Key	Long Integer	-105
F14 Key	Long Integer	-107
F15 Key	Long Integer	-113
F2 Key	Long Integer	-120
F3 Key	Long Integer	-99
F4 Key	Long Integer	-118
F5 Key	Long Integer	-96
F6 Key	Long Integer	-97
F7 Key	Long Integer	-98
F8 Key	Long Integer	-100
F9 Key	Long Integer	-101
Help Key	Long Integer	5
Home Key	Long Integer	1
Left Arrow Key	Long Integer	28
Page Down Key	Long Integer	12
Page Up Key	Long Integer	11
Return Key	Long Integer	13
Right Arrow Key	Long Integer	29
Tab Key	Long Integer	9
Up Arrow Key	Long Integer	30

Hierarchical Lists

Constant	Type	Value
Ala Macintosh	Long Integer	1
Ala Windows	Long Integer	2
Macintosh node	Long Integer	860
Use PicRef	Long Integer	131072
Use PICT resource	Long Integer	65536
Windows node	Long Integer	138

ISO Latin Character Entities

Constant	Type	Value
ISO L1 a acute	String	&aacute;
ISO L1 a circumflex	String	&acirc;
ISO L1 a grave	String	&agrave;
ISO L1 a ring	String	&aring;
ISO L1 a tilde	String	&atilde;
ISO L1 a umlaut	String	&auml;
ISO L1 ae ligature	String	&aelig;
ISO L1 Ampersand	String	&amp;
ISO L1 c cedilla	String	&ccedil;
ISO L1 Cap A acute	String	&Aacute;
ISO L1 Cap A circumflex	String	&Acirc;
ISO L1 Cap A grave	String	&Agrave;
ISO L1 Cap A ring	String	&Aring;
ISO L1 Cap A tilde	String	&Atilde;
ISO L1 Cap A umlaut	String	&Auml;
ISO L1 Cap AE ligature	String	&AELig;
ISO L1 Cap C cedilla	String	&Ccedil;
ISO L1 Cap E acute	String	&Eacute;
ISO L1 Cap E circumflex	String	&Ecirc;
ISO L1 Cap E grave	String	&Egrave;
ISO L1 Cap E umlaut	String	&Euml;
ISO L1 Cap Eth Icelandic	String	&ETH;
ISO L1 Cap I acute	String	&Iacute;
ISO L1 Cap I circumflex	String	&Icirc;
ISO L1 Cap I grave	String	&Igrave;
ISO L1 Cap I umlaut	String	&Iuml;
ISO L1 Cap N tilde	String	&Ntilde;
ISO L1 Cap O acute	String	&Oacute;
ISO L1 Cap O circumflex	String	&Ocirc;
ISO L1 Cap O grave	String	&Ograve;
ISO L1 Cap O slash	String	&Oslash;
ISO L1 Cap O tilde	String	&Otilde;
ISO L1 Cap O umlaut	String	&Ouml;
ISO L1 Cap THORN Icelandic	String	&THORN;
ISO L1 Cap U acute	String	&Uacute;

ISO Latin Character Entities (continued)

Constant	Type	Value
ISO L1 Cap U circumflex	String	&Ucirc;
ISO L1 Cap U grave	String	&Ugrave;
ISO L1 Cap U umlaut	String	&Uuml;
ISO L1 Cap Y acute	String	&Yacute;
ISO L1 Copyright	String	&copy;
ISO L1 e acute	String	&eacute;
ISO L1 e circumflex	String	&ecirc;
ISO L1 e grave	String	&egrave;
ISO L1 e umlaut	String	&euml;
ISO L1 eth Icelandic	String	&eth;
ISO L1 Greater than	String	&gt;
ISO L1 i acute	String	&iacute;
ISO L1 i circumflex	String	&icirc;
ISO L1 i grave	String	&igrave;
ISO L1 i umlaut	String	&iuml;
ISO L1 Less than	String	&lt;
ISO L1 n tilde	String	&ntilde;
ISO L1 o acute	String	&oacute;
ISO L1 o circumflex	String	&ocirc;
ISO L1 o grave	String	&ograve;
ISO L1 o slash	String	&oslash;
ISO L1 o tilde	String	&otilde;
ISO L1 o umlaut	String	&ouml;
ISO L1 Quotation mark	String	&quot;
ISO L1 Registered	String	&reg;
ISO L1 sharp s German	String	&szlig;
ISO L1 thorn Icelandic	String	&thorn;
ISO L1 u acute	String	&uacute;
ISO L1 u circumflex	String	&ucirc;
ISO L1 u grave	String	&ugrave;
ISO L1 u umlaut	String	&uuml;
ISO L1 y acute	String	&yacute;
ISO L1 y umlaut	String	&yuml;

Math

Constant	Type	Value
Degree	Real	0.0174532925199432958
e number	Real	2.71828182845904524
Pi	Real	3.141592653589793239
Radian	Real	57.29577951308232088

Open window

Constant	Type	Value
Alternate dialog box	Long Integer	3
Has grow box	Long Integer	4
Has highlight	Long Integer	1
Has window title	Long Integer	2
Has zoom box	Long Integer	8
Modal dialog box	Long Integer	1
Movable dialog box	Long Integer	5
Palette window	Long Integer	720
Plain dialog box	Long Integer	2
Plain fixed size window	Long Integer	4
Plain no zoom box window	Long Integer	0
Plain window	Long Integer	8
Round corner window	Long Integer	16

Picture Display Formats

Constant	**Type**	**Value**
On Background	Long Integer	3
Scaled to Fit	Long Integer	2
Scaled to fit prop centered	Long Integer	6
Scaled to fit proportional	Long Integer	5
Truncated Centered	Long Integer	1
Truncated non Centered	Long Integer	4

Platform Interfaces

Constant	Type	Value
Automatic interface	Long Integer	-1
Copland interface	Long Integer	3
Macintosh interface	Long Integer	0
Windows 3.1 interface	Long Integer	1
Windows 95 interface	Long Integer	2

Platform Properties

Constant	Type	Value
INTEL 386	Long Integer	386
INTEL 486	Long Integer	486
Macintosh 68K	Long Integer	1
Pentium	Long Integer	586
Power Macintosh	Long Integer	2
PowerPC 601	Long Integer	601
PowerPC 603	Long Integer	603
PowerPC 604	Long Integer	604
Windows	Long Integer	3

Process state

Constant	Type	Value
Aborted	Long Integer	-1
Delayed	Long Integer	1
Does not exist	Long Integer	-100
Executing	Long Integer	0
Hidden modal dialog	Long Integer	6
Paused	Long Integer	5
Waiting for input output	Long Integer	3
Waiting for internal flag	Long Integer	4
Waiting for user event	Long Integer	2

Query Destinations

Constant	Type	Value
Into current selection	Long Integer	0
Into named selection	Long Integer	2
Into set	Long Integer	1
Into variable	Long Integer	3

Resources Properties

Constant	Type	Value
Changed resource bit	Long Integer	1
Changed resource mask	Long Integer	2
Locked resource bit	Long Integer	4
Locked resource mask	Long Integer	16
Preloaded resource bit	Long Integer	2
Preloaded resource mask	Long Integer	4
Protected resource bit	Long Integer	3
Protected resource mask	Long Integer	8
Purgeable resource bit	Long Integer	5
Purgeable resource mask	Long Integer	32
System heap resource bit	Long Integer	6
System heap resource mask	Long Integer	64

SCREEN DEPTH

Constant	Type	Value
Black and white	Long Integer	0
Four colors	Long Integer	2
Is color	Long Integer	1
Is gray scale	Long Integer	0
Millions of colors 24 bit	Long Integer	24
Millions of colors 32 bit	Long Integer	32
Sixteen colors	Long Integer	4
Thousands of colors	Long Integer	16
Two fifty six colors	Long Integer	8

SET RGB COLOR

Constant	Type	Value
Default background color	Long Integer	-2
Default dark shadow color	Long Integer	-3
Default foreground color	Long Integer	-1
Default light shadow color	Long Integer	-4

Standard System Signatures

Constant	Type	Value
Picture Document	String	PICT
QT Animation compressor	String	rle
QT Compact video compressor	String	cdvc
QT Graphics compressor	String	smc
QT Photo compressor	String	jpeg
QT Raw compressor	String	raw
QT Video compressor	String	rpza
Text Document	String	TEXT
Windows MIDI Document	String	MID
Windows Sound Document	String	WAV
Windows Video Document	String	AVI

TCP Port Numbers

Constant	Type	Value
TCP Authentication	Long Integer	113
TCP DNS	Long Integer	53
TCP Finger	Long Integer	79
TCP FTP Control	Long Integer	21
TCP FTP Data	Long Integer	20
TCP Gopher	Long Integer	70
TCP HTTP WWW	Long Integer	80
TCP IMAP3	Long Integer	220
TCP Kerberos	Long Integer	88
TCP KLogin	Long Integer	543
TCP Nickname	Long Integer	43
TCP NNTP	Long Integer	119
TCP NTalk	Long Integer	518
TCP NTP	Long Integer	123
TCP PMCP	Long Integer	1643
TCP PMD	Long Integer	1642
TCP POP3	Long Integer	110
TCP Printer	Long Integer	515
TCP RADACCT	Long Integer	1646
TCP RADIUS	Long Integer	1645
TCP Remote Cmd	Long Integer	514
TCP Remote Exec	Long Integer	512
TCP Remote Login	Long Integer	513
TCP Router	Long Integer	520
TCP SMTP	Long Integer	25
TCP SNMP	Long Integer	161
TCP SNMPTRAP	Long Integer	162
TCP SUN RPC	Long Integer	111
TCP Talk	Long Integer	517
TCP Telnet	Long Integer	23
TCP TFTP	Long Integer	69
TCP UUCP	Long Integer	540
TCP UUCP RLOGIN	Long Integer	541

Test path name

Constant	Type	Value
Is a directory	Long Integer	0
Is a document	Long Integer	1

Time Display Formats

Constant	Type	Value
HH MM	Long Integer	2
HH MM AM PM	Long Integer	5
HH MM SS	Long Integer	1
Hour Min	Long Integer	4
Hour Min Sec	Long Integer	3

Window kind

Constant	Type	Value
External window	Long Integer	5
Floating window	Long Integer	14
Modal dialog	Long Integer	9
Regular window	Long Integer	8

Command Index