OMNIS Programming

OMNIS Software

August 1998

Table of Contents

About This Manual

This manual describes how you develop an application using OMNIS Studio, focusing on the programming aspects of application development.

The *Using OMNIS Studio* manual describes the primary objects and components in OMNIS Studio, and describes how you can create and modify them. If you have not already looked at *Using OMNIS Studio*, you should do so before starting on this manual.

Other manuals in the set, which are available in on-line format, include

– *OMNIS Studio Conversion*
  describes how you convert your OMNIS 7 applications to OMNIS Studio and, for the benefit of OMNIS 7 users, introduces the new features in OMNIS Studio

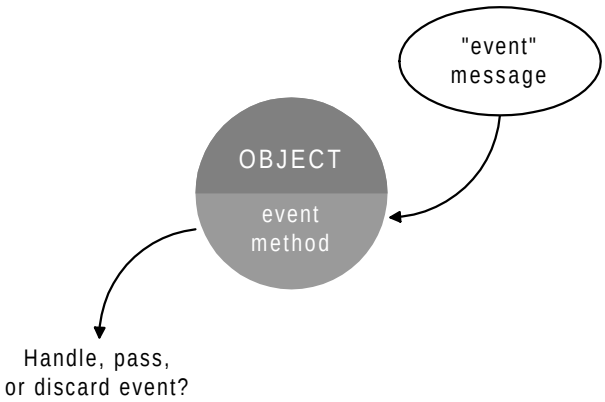
– *OMNIS Graphs*
  describes the Graph external component available in OMNIS

In addition to these manuals, a comprehensive Help system describing the OMNIS Studio commands and functions is available from within the OMNIS Studio development environment.

Your Notes

Chapter 1—Events and Messages

Almost all user actions in OMNIS generate an *event*. For example, if the user clicks on a field in a window an event is generated. If the user tabs from one cell in a grid field to another an event is generated, and if the user closes a window an event is generated. When the event occurs a *message* is sent to the object in which the event occurred. The key to creating an events-based application that properly functions is in the methods you write in the various objects in your library to intercept or handle these events. These methods are called *event handling methods* and you put them behind the fields, objects, and windows in your library. You can also write event handling methods for the objects on a report.



When an event occurs the *default action* normally takes place. For example, when the user presses the tab key to move to the next field on a data entry field, the default action is for the cursor to leave the current field and enter the next field on the window, and normally this is exactly what happens. However you could put a method behind the field that performs any one of a number of alternative actions in response to the tab. That is, the event handling method could use the tab to trigger a particular piece of code and then allow the default action to occur, it could pass the event to somewhere else in your library, or it could discard the event altogether and stop the default action from happening.

Events are reported in OMNIS as *event messages*. These messages are sent to the event handling methods as one or more *event parameters*. The first parameter of an event message, pEventCode, contains an *event code* representing the event. Event messages may contain a second or third parameter that tell you more about the event. For example, a click

on a list box will generate an evClick event plus a second parameter pRow telling you the row clicked on. Note that all event codes are prefixed with the letters “ev”, and all event parameters are prefixed with the letter “p”. You can use the event codes in your event handling methods to detect specific events, and the event parameters to test the contents of event messages.

Event Handling Methods

You can write an event handling method for each field and object contained in window, menu, toolbar, and report classes. The other class types do not generate events. You add the event methods for window and report fields in the *Field Methods* for the class. For menu classes you can add an event method to the *Line Methods* for a menu line, and for toolbar classes you can enter an event method in the *Tool Methods* for each toolbar control.

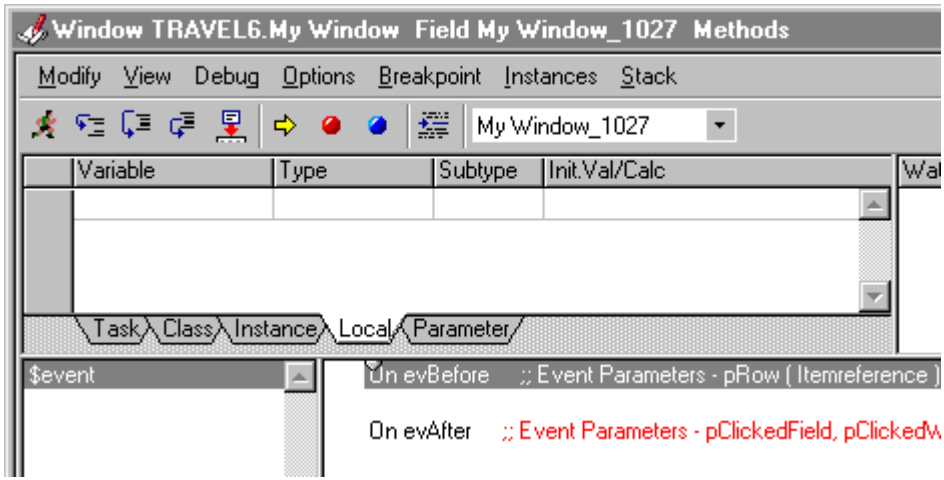
Window fields, toolbar controls, and menu lines contain a default event handling method called $event(), and report fields contain a default event handling method called $print(). If you open the field methods for a window field, toolbar control, or menu line you will see an $event() method, and for each report field you will see a $print() method for the object. These are the default event handling methods for those objects.

To view the event handling method for a field or object

- Show the design screen for the class
- Right-click on the field, menu line or toolbar control
- Choose Field Methods, Line Methods, or Tool Methods, as appropriate

The method editor opens showing the first method in the list for the field or object. If this is not the $event() method, select it from the list to view it. Some event handlers will contain code to handle a range of possible events in the object.

For example, the following screenshot shows the default $event() method for an entry field.



The event handling method for some types of field may be empty, because there is only one possible event for the object. For example, the event handling method for a menu line is empty since you can only select a menu line. Therefore any code you put in the $event() method for a menu line runs automatically when you select the line.

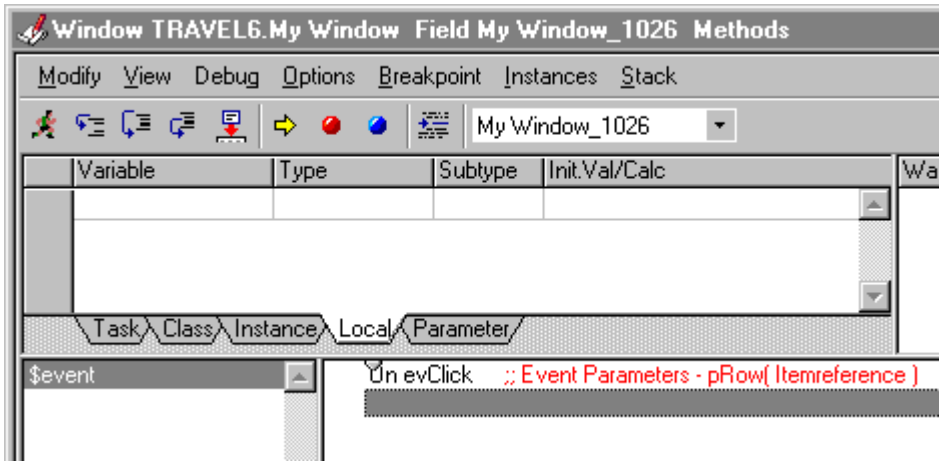
To enter the code for an event handling method

- Assuming you have opened a default $event() method for a field, click on the next command line after the *On* command

or, for an empty $event() method

- Select the first line of the method

For example, you can open the event method for a pushbutton, that contains a single *On evClick* command which will detect a click on the button.



- Enter the code you want to run for that event

You could use the *Do* command and some notation in your event handling method, or you can use the *Do* method command to run another method in the current class or instance, or the *Do code method* command to run a method in a code class; in all cases, you can put literally any code in an event handling method and it will run given the right event.

The On Command

You can use the *On* command to detect events in your event handling methods. Fields from the Component Store may contain a default event handling method with one or more *On* commands to detect different events. For example, an entry field contains the method

```
On evBefore     ;; Event Parameters - pRow ( Itemreference )

On evAfter     ;; Event Parameters - pClickedField,
                  pClickedWindow, pMenuLine,
                  pCommandNumber, pRow
```

These lines detect the events **evBefore** and **evAfter**, which are the event codes contained in the message sent when the user enters or leaves the field, respectively. The in-line comments indicate which event parameters OMNIS supplies for that event. In most cases, the event parameters are references containing values to do with the context of the event: the field clicked on, the list row number, the menu line number, and so on.

There is a summary of the most common event codes at the end of this chapter.

You can use the default event handling method for a field or add your own. The following event handler for a data entry field detects an evBefore as the user enters the field and performs a calculation changing the value of the field.

```
On evBefore                  ;; user tabs into date field
   Calculate cDate as #D     ;; cDate is the dataname of the field
   Redraw {DateField}        ;; the current field
   Quit event handler
```

Code which is common to all events should be placed at the start of the event handling method, *before* any *On* commands. You can use the *On default* command to handle any events not covered by an earlier *On* command line. The general format is

```
; code which will run for all events
On evBefore
   ; code for evBefore events
On evAfter
   ; code for evAfter events
On default
   ; code for any other events
```

When you enter the *On* command in an event handling method, it displays a list of all the available event codes in the command palette. You can click on the one you want, or you can enter more than one event code for a single *On* command, for example *On evClick, evDoubleClick*. *On* commands cannot be nested or contained in an *If* or loop construct.

When you have entered the *On* command line for a particular event and selected the next command line, you can open the Catalog to view the event parameters for that event code.

- Click on the line ***after*** an *On evClick* command line
- Open the Catalog (F9/Cmnd-9)
- Click on Event Parameters under the Variables tab

For example, an *On evClick* command displays the parameters pEventCode and pRow in the Catalog. You can use these event parameters in your event handling methods to test the event message. A click on a list box generates an evClick event message containing a reference to the row clicked on, held in the pRow event parameter. You can test the value of pRow in your code

```
On evClick        ;; method behind a list box
   If pRow=1      ;; if row 1 was clicked on
      ; Do this...
   End If
   If pRow=2      ;; if row 2 was clicked on
      ; Do that...
   End If
```

All events return the parameter pEventCode, which you can also use in your event handling methods.

```
On evAfter,evBefore      ;; method behind field
   ; Do this code for both events
   If pEventCode=evAfter
      ; Do this for evAfter events only
   End If
   If pEventCode=evBefore
      ; Do this for evBefore events only
   End If
```

The parameters for the current event are returned by the *sys(86)* function, which you can use while debugging or monitoring which events are handled by which methods. For example, you could use the *Send to trace log* command and the functions *sys(85*) and *sys(86)*, to report the current method and events, in the $event() method for a field

```
; $event() method for field 10 on the window
Send to trace log {[sys(85)] - [sys(86)]}
; sends the following to the trace log when you tab out of the field
WindowName/10/$event - evAfter,evTab
WindowName/10/$event - evTab
```

You can use any of the parameters reported for an event in your event handling methods. However, if you enter an event parameter not associated with the current event, the parameter will be null and lead to a runtime error.

The Quit event handler Command

If you want to discard or pass an event you can use the *Quit event handler* command to terminate an *On* construct. A field event handling method might have the following structure.

```
; general code for all events
On evBefore
   ; code for evBefore events
On evAfter
   ; code for evAfter events
On evClick,evDoubleClick
   ; code for click events
   Quit event handler (pass event)
On default
   ; code for any other events
```

The *Quit event handler* command has two options

– **Discard event**
  for some events you can discard the event and stop the default action taking place

– **Pass to next handler**
  passes the event to the next handler in the event chain

Discarding Events

In certain circumstances you might want to detect particular events and discard them in order to stop the default action from occurring. You can discard or throw away events using the *Quit event handler* command with the Discard event option enabled. Note however, you *cannot* discard some events or stop the default action from taking place since the event has already occurred by the time it is detected by an event handling method. In this case, a *Quit event handler (Discard event)* has no effect for some events.

Being able to discard an event is useful when you want to validate what the user has entered in a field and stop the cursor leaving the field if the data is invalid. The following method displays an appropriate message and stays in the field if the user does not enter the data in the correct format.

```
On evAfter                ;; as user leaves the field
   If len(CustCode <> 6)   ;; check a value has been entered
      If len(CustCode = 0)  ;; field left blank
         OK message {You must enter a customer code}
      Else                   ;; wrong length code entered
         OK message {The customer code must have 6 digits}
      End If
      Quit event handler (Discard event) ;; stay in the field
   End If
```

You can also handle or discard events using the *Quit method* command with a return value of kHandleEvent or kDiscardEvent, as appropriate.

Window Events

So far the discussion has focused on field events, which you would normally handle in the field using an event handling method. However you can enter methods to handle events that occur in your window as well. Like fields, the event handling method for a window class is called $event(), and you enter this method in the *Class Methods* for the window class.

Window classes do not contain an $event() method by default, but you can insert a method with this name. You enter the code for a window $event() method in exactly the same as for fields using the *On* command to detect events in your window.

Window events affect the window only and not individual fields. They include clicks on the window background, bringing the window to the front or sending it to the back, moving it, sizing it, minimizing or maximizing the window, or closing it. For example, when you click on a window's close box, the evCloseBox and evClose events are generated in the window indicating that the close box has been clicked and the window has been closed. You could enter an $event() method for the window to detect these events and act accordingly.

The following window $event() method detects a click on a window behind the current window, and discards the click if the user is inserting or editing data.

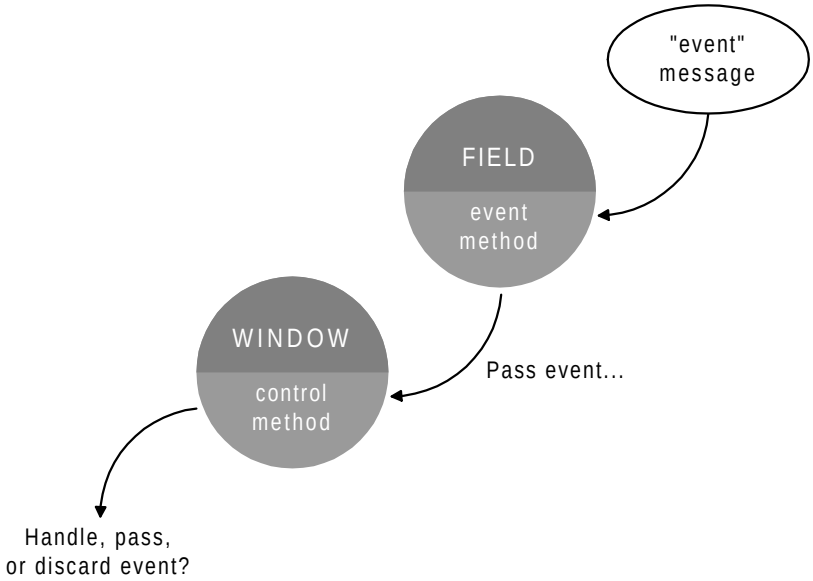
```
On evWindowClick          ;; user has clicked on a window behind
   If cInserting | cEditing     ;; vars to detect current mode
      OK message {You cannot switch windows while entering data}
      Quit event handler (Discard event) ;; keep window on top
   End If
   Quit event handler
```

The following window $event() method checks for events occurring in the window and runs the appropriate methods elsewhere in the class. Note you cannot trap an evResize and discard it since the resizing has already occurred, but you can reverse the resizing by setting the size of the open window back to the size stored in the class.

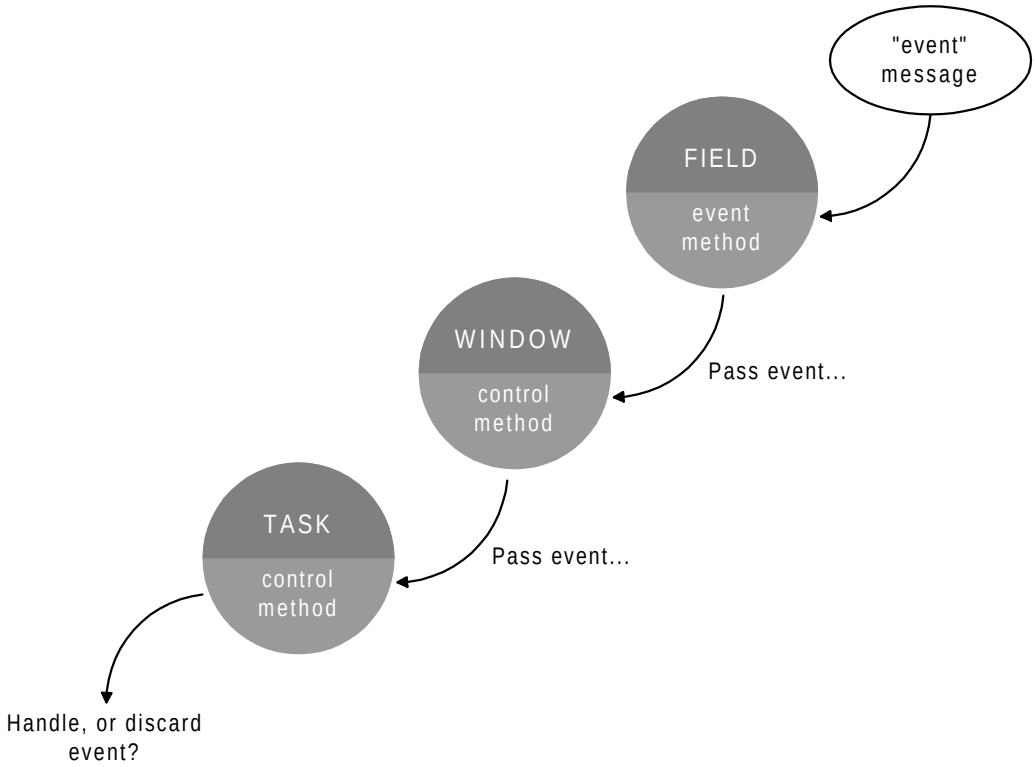
```
On evToTop
   Do method Activate
   Quit event handler
On evWindowClick
   Do method Deactivate
   Quit event handler
On evClose
   Do method Close
   Quit event handler
On evResized
   Do $cwind.$width.$assign($cclass.$width)
   Do $cwind.$height.$assign($cclass.$height)
   Quit event handler (Discard event)
```

Control Methods and Passing Events

As already described, you handle events for fields using an event handling method contained in the field, but you can add a further level of control over field events by adding a method called $control() to your window. This method is called a *window control method*. To allow this method to handle events you must pass events to it from the field event handling methods. You do this by including in your field event handler the *Quit event handler* command with the **Pass to next handler** option enabled.



As a further level of control, you can add a $control() method to your tasks. This method is called a *task control method*. Events are passed to the task control method from the window control method contained in the window belonging to the task. Therefore, an event may be generated in the field, passed to the window control method, and then passed to the task control method.



Window events that are handled in the $event() method for a window can be passed to the task $control() method as well.

At each level an event handling method can discard the event or pass it on to the next event handler. At the task level, the highest level of control, the event can be processed and the default action takes place, or the event can be discarded and no further action occurs.

The OMNIS event processing mechanism gives you absolute control over what is going on in your application, but it also means you need to design your event handling methods with care. It is important not to pass on an event to higher levels unnecessarily and to keep control methods short, to limit the time spent processing each event.

In the following example, the $control() method is contained in an OMNIS data entry window. It sets the main file for the window when it is opened or comes to the top, and does not let the user close the window if OMNIS is in data entry mode.

```
On evToTop
   ; window comes to the top or is opened
   Set main file {FCUSTOMERS}
   Quit event handler
On evClose
   If cInserting | cEditing     ;; vars to detect current mode
      ; User closes window when in enter data mode
      OK message {You can't close in enter data mode}
      Quit event handler (Discard event)
   End If
```

Event Processing and Enter Data Mode

Normally, the default processing for an event takes place when all the event handler methods dealing with the event have finished executing. It is not possible to have active unprocessed events when waiting for user input so the default processing is carried out for any active events after an *Enter data* command has been executed or at a debugger break. Therefore if required, you can use the *Process event and continue* command to override the default behavior and force events to be processed allowing an event handling method to continue.

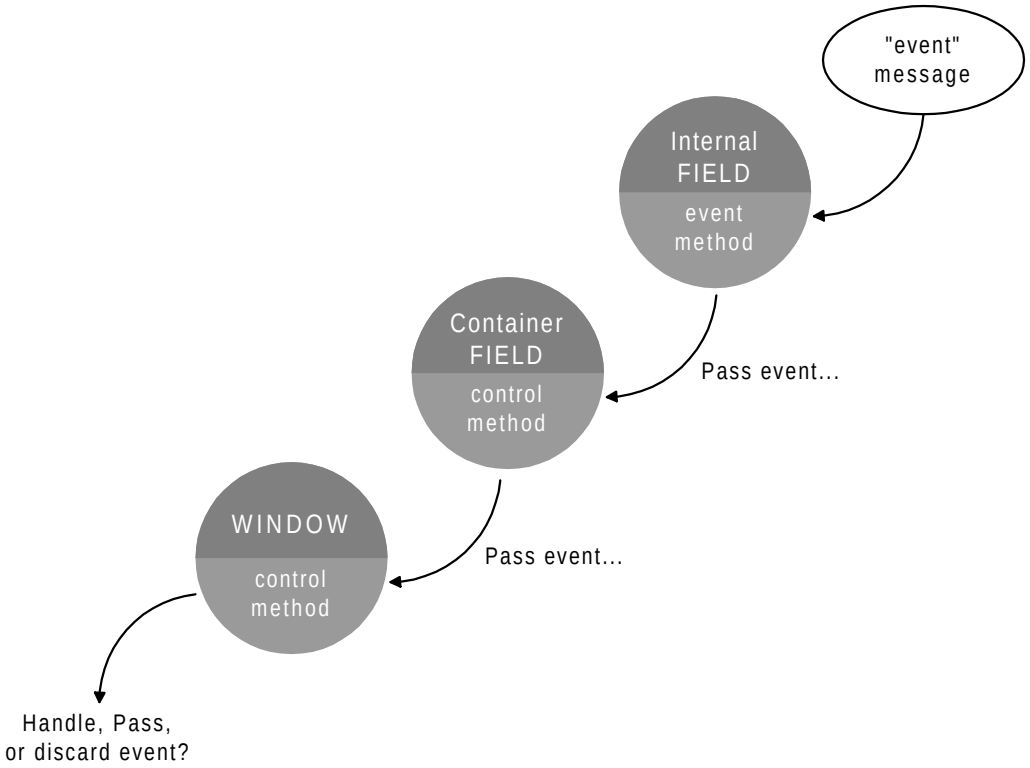
The *Process event and continue (Discard event)* option lets you discard the active event. For example, in an event handler for evOK the following code would cause the OK event to be thrown away before the subsequent enter data starts.

```
On evOK
   Process event and continue (Discard event)
   Open window instance {window2}
   Enter data
```

Container Fields and Events

Container fields are fields that contain other fields; examples of container fields include subwindows, tab panes, page panes, scroll boxes, and complex grid fields. The logic for handling and passing events within a container field is the same as for simple fields, it just has more levels of control.

For the purposes of event handling, you can regard the container field as both a field on the parent window, and a window since it contains other fields. In this respect, a container field can have an $event() method that handles events for the container field itself, and a $control() method that handles events passed to it from the individual fields inside the container field. Each field in the container field has a $event() method to handle its own events. If the control method for your container field allows it, events are passed to the parent window control method, which in turn can be passed onto the task control method or discarded as appropriate.



You can nest container fields such as subwindows and tab panes, but nested container fields do not pass events.

Queuing Events

Some user actions generate a single event which is handled as it occurs by your event handling methods. The event may be dealt with completely in the field or it may be passed up the event chain as required. However some user actions generate a whole series of events, one after another. These events are placed in an *event queue*. Each event is handled by your event handling methods strictly in turn on a first-in, first-out basis. For example, when the user tabs from one field to another the current field is sent an evAfter and then an evTab event, then the new field is sent an evBefore event: all these events are placed in the event queue in response to a single user action, the tab. Similarly when you close a window, the current field is sent an evAfter, the window is sent an evCloseBox event, then it is sent an evClose event. Each one of these events is sent to the appropriate object and is handled by your event handling methods *before* the next event in the queue is handled.

In addition to events generated by user actions, you can append an event to the event queue using the *Queue* commands in the *Events...* group.

```
Queue bring to top
Queue close
Queue cancel
Queue set current field
Queue click
Queue double-click
Queue keyboard event
Queue OK
Queue scroll (Left|Right|Up|Down)
Queue tab
Queue quit
```

These commands let you simulate user actions such as key presses and clicks on buttons or windows. For example, the *Queue bring to top {WINDOWNAME}* command brings the specified window instance to the top and simulates a user clicking behind the current window. Events generated by these commands are handled after those that are currently queued. You can queue several events in succession.

Types of Events

The following sections list the events generated by the different types of fields and windows. You can use the event codes described below in your field and window $event() methods, and/or your window and task $control() methods. The events and parameters are described in more detail in the OMNIS Help.

Field Events

For most types of entry field, button, list and grid field you can detect when the user enters and leaves the field, or when the user clicks on the field. The following events are reported by many types of fields

– **evAfter**
the focus is about to leave the field. For example, the user has clicked outside the current field, or they have pressed tab, or they have selected a line in a menu. The parameters for the event tell you what other field or window was clicked on, or which menu or list line was selected

– **evBefore**
the cursor has entered the field

– **evClick** and **evDoubleClick**
the field or window has been clicked or double-clicked on: not reported for entry fields. For lists, the second event parameter tells you which row was selected

– **evOpenContextMenu**
a context menu is about to open over the field

– **evSent**
sent to a field when its value has changed due to DDE or Apple event

Window Events

The following events are sent to the current top window.

– **evClose**
the window is about to be closed

– **evCloseBox**
the user has clicked the close box of the window

– **evCustomMenu**
the user has selected a line in a custom menu; the second event parameter tells you the number of the menu line selected

– **evMinimized** and **evMaximized**
the window has been minimized, or maximized

– **evMoved**
the window has been moved

– **evOK** and **evCancel**
the user has clicked the OK or Cancel button, or has pressed the equivalent key(s)

– **evResized** and **evRestored**
the window has been resized, or restored to its normal size

– **evStandardMenu**
the user has selected a line in a standard menu, or has clicked on one of the standard OMNIS database buttons (Find, Next, Previous, etc.); the second event parameter is an internal number for the standard menu line selected

– **evToTop**
the window has come to the top

– **evWindowClick**
the user has clicked on another window; the second event parameter is a reference to the window clicked on

Scroll Events

These events can occur for a field or window provided they have a vertical or horizontal scroll bar as appropriate.

– **evHScrolled** and **evVScrolled**
the field or window has been scrolled horizontally, or vertically

Mouse Events

The following mouse events are sent to a field or window background. Mouse and right-mouse button events are generated only if the $mouseevents and $rmouseevents library preferences are enabled. Under MacOS, right-mouse events are generated when you hold down the Ctrl key and click the mouse.

– **evMouseDouble** and **evRMouseDouble**
the mouse, or right-mouse button is double-clicked in a field or window

– **evMouseDown** and **evRMouseDown**
evMouseUp and **evRMouseUp**
the mouse, or right-mouse button is held down in a field or window, or the mouse button is released

– **evMouseEnter** and **evMouseLeave**
the mouse pointer enters, or leaves a field

– **evDrag**
the mouse is held down in a field and a drag operation is about to start; the parameters report the type and value of the data

– **evCanDrop**
whether the field or window containing the mouse can accept a drop; the parameters reference the object being dropped, the type and value of the data

– **evWillDrop**
the mouse is released at the end of a drag operation. The parameters reference the object being dropped, the type and value of the data

– **evDrop**
the mouse is released over the destination field or window at the end of a drag operation. The parameters reference the object being dropped, the type and value of the data

Complex Grid Events

These events are generated when a complex grid is changed in some way by the user.

– **evExtend**
an extra line has been added to the end of the grid. The second event parameter contains a reference to the new row

– **evRowChange**
the row in the grid has changed: a reference to the row is generated

String and Data Grid Events

These events are generated when a string or data grid is changed.

– **evCellChanging** and **evCellChanged**
the grid cell is about to change, or has changed; for example, the user has tabbed. The parameters tell you the position of the cell and its data

– **evScrollTip**
sent to a string or data grid when scrolled and lets you intercept and change the scrolltip text

Tab Pane and Tab Strip Events

A tab pane or tab strip can have a number of tabs. This event is generated when one of the tabs is selected.

– **evTabSelected**
a tab has been selected. The second event parameter is the number of the tab selected

Tree List Events

A tree list can have a number expandable and collapsable nodes. The following events are generated when a node is clicked on.

– **evTreeExpand** and **evTreeCollapse**
a node has been expanded or collapsed; the second event parameter is a reference to the node expanded or collapsed

– **evTreeExpandCollapseFinished**
a node has expanded or collapsed; sent after an evTreeCollapse or evTreeExpand message

– **evTreeNodeIconClicked**
a node icon has been clicked; the second parameter is a reference to the node

– **evTreeNodeNameFinishing** and **evTreeNodeNameFinished**
a node name is about to change or has changed; the second parameter is a reference to the node; the third parameter contains the new text for the node

Headed List Box Events

The following events are generated when the user edits a cell or clicks on a column header in a headed list box.

– **evHeadedListEditFinished**
a cell has been edited; the second and third parameters are the line and column numbers of the selected cell

– **evHeadedListEditStarting**
the cell is put into edit mode; discarding this event prevents editing; the second and third parameters are the line and column numbers of the selected cell

– **evHeadedListEditFinishing**
the user has edited the cell and pressed Return; discarding this event leaves the field in edit mode; the second and third parameters are the line and column numbers of the selected cell; the fourth parameter is the new text in the cell

– **evHeaderClick**
a header button has been clicked on; the second parameter contains the column number

Icon Array Events

The following events are generated when the user edits the text in an icon array.

– **evIconDeleteStarting** and **evIconDeleteFinished**
the user has pressed the delete key, or the delete has finished completely

– **evIconEditStarting**
the icon text is put into edit mode; discarding the event prevents editing; the second parameter contains the line number of the list that is to be edited

– **evIconEditFinishing**
the user has edited the icon text and pressed Return; discarding the event leaves the field in edit mode; the second parameter contains the line number of the list being edited; the third is the new text entered

– **evIconEditFinished**
the user has finished editing; the second parameter contains the line number of the list that has been edited

Key Events

These events are generated when the user presses a key. Key events are generated only if the $keyevents library preference is enabled.

– **evKey**
any key is pressed. Contains the letter key and system key pressed

– **evTab** and **evShiftTab**
the tab key, or shift-tab key is pressed

Field Status Events

The following events are reported for fields only and reflect the current status of a field. They are generated only if the $statusevents library preference is enabled.

– **evDisabled** and **evEnabled**
a field is disabled or enabled

– **evHidden** and **evShown**
a field is hidden or shown

Chapter 2—Methods and Notation

This chapter describes how you write methods to perform operations in your application. It introduces the different commands and programming constructs you can use to control program flow or perform complex calculations in OMNIS. You add methods to the classes and objects in your library using the *method editor*. Methods let you

– Manipulate classes and other library objects

– Handle events and control program flow

– Send SQL to a server and process the results

– Interface with external software

OMNIS provides a complete 4GL programming language comprising over 400 commands, each command performing a specific function or operation. In addition OMNIS provides a means to manipulate the objects in your library called the *notation*: this accesses the standard properties and methods contained in the objects in your library.

A method can contain one or more OMNIS commands, or some notation, or in practice a combination of these. For example, to open a window from a menu line method you only need one command, *Open window instance*, which as the name suggests instantiates or opens a window. A method that connects you to a server database requires several commands executed in a particular order. You can perform most operations using the notation and the *Do* command. For example, you can open a window using the *Do* command and the $open() method.

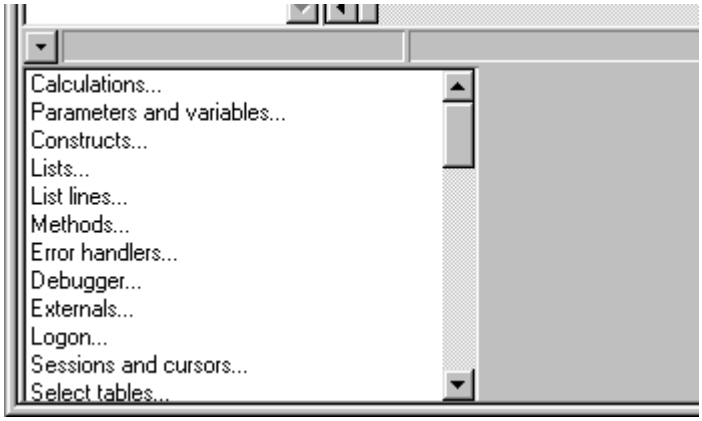
For further details about specific commands used throughout this chapter, see the OMNIS Studio Help. When you start to program methods you will need to use the debugger which is described in the *Debugging Methods* chapter.

The *Variables and Methods* chapter in Using OMNIS Studio tells you how to add methods to the objects in your library, and the *Events and Messages* chapter deals specifically with event handling methods. In addition, commands that you use with list variables are dealt with in the *List Programming* chapter.

Commands

The following sections outline the more important commands or groups of commands in OMNIS. The commands that you can use in your methods are listed in the *command list* at the bottom of the method editor. If the command list is not showing in the method editor you can show it using View>>Show Command Palette, or by pressing Shift-F6 under Windows or Shift-Cmnd-6 under MacOS.

Double-click on each group in the command list to get an idea of the full range of commands available in OMNIS.



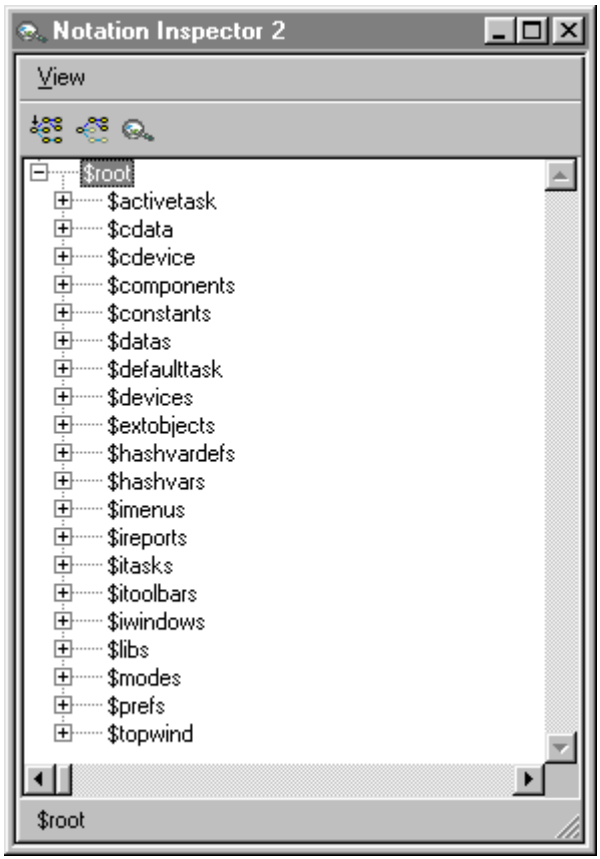
Each group in the command list contains a number of commands that manipulate a particular type of object or perform related operations. For example, the *Calculations...* group contains the *Calculate* command that lets you do calculations and assign a value to a variable, and the *Do* command that lets you execute and modify objects using the notation. The *Constructs...* group contains programming constructs such as *If...Else If*, *Repeat...Until*, and *For* loops.

The Flag

Some of the commands set a Boolean OMNIS variable called the *flag*, or #F, to true or false depending on the success of an operation. Other commands test the current value of the flag and branch accordingly. The *OMNIS Studio* Help documents whether or not a command affects the flag.

Notation

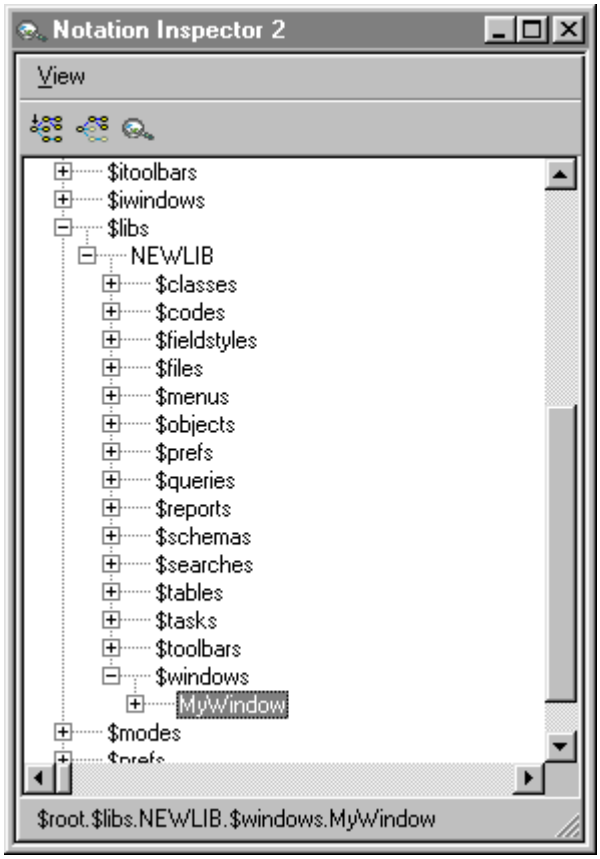
OMNIS structures its objects in an object tree, or hierarchical arrangement of objects and groups that contain other objects. The complete tree contains all the objects in OMNIS itself, together with your design libraries, classes, and other objects created at runtime. You can view the complete object tree in the Notation Inspector.



The object at the base of the tree is called $root. The $libs group contains all the current open libraries and lets you access each library and its classes at design time. The classes and objects in each library are stored in their own separate groups; for example the $windows group contains all the window classes in a library. Most of the other groups directly under $root contain the objects created at runtime when you run your application; for example the $iwindows group contains all the window instances currently open.

When you want to reference a particular object, a class or instance perhaps, you must access the right branch of the object tree. For example, you must access the $windows group to

reference a window class: the following screenshot shows a window called MyWindow in a library called NEWLIB.



To access a window instance, say an instance of the same Window class, you must reference the $iwindows group, directly under the $root object.

To facilitate a system of naming or referring to an object in the object tree, and its properties and methods, OMNIS uses a system called the *notation*. The notation for an object is really the path to the object within the object tree. The full notation for an object is shown in the status bar of the Notation Inspector. You can use the notation to execute a method or to change the properties of an object, and you can use a notation string anywhere you need to reference a variable or field name.

In the notation all property and standard method names begin with a dollar sign “$”, and methods are further distinguished from properties by having parentheses after their name. Standard objects and group names also begin with a dollar sign. To write the full notation for an object you need to include each object and group in the path to the object, separating

each object using "." a dot. For example, to refer to a window class in a library you would use the following notation

```
$root.$libs.LIBRARYNAME.$windows.Windowname
```

This notation includes $root as the base object, the $libs group containing all the open libraries, the name of your library, the $windows group containing all the window classes in your library, and lastly the name of the window itself. If you want to refer to a particular object on your window you need to add the $objs group and the name of the object

```
$root.$libs.Libraryname.$windows.Windowname.$objs.Objectname
```

You can omit certain object names from a notation string to make it shorter, and when you have only one library open usually you can omit the library name as well. You can omit the following objects: $root, $constants, $clib, $hashvars, $libs, $extobjects, $tvars, $datas, $cvars, $files, $lvars, $vals. In most cases therefore, you can refer to an object on a window as

```
$windows.Windowname.$objs.Objectname
```

In addition, there are a number of shortcuts that let you reference objects, without always referring right back to the $root object, and certain global objects that you can use to make your code more generic. These are described below.

Item References

To save you time and effort, and to make your code more efficient, you can create an alias or reference to an object which you can use in place of the full notation for the object. To do this, you create a variable of type *item reference* and use the *Set reference* command to assign the notation to the variable. The item reference variable can be of any scope, and the notation can be any valid OMNIS notation for an object, a group, or even an object property. For example

```
; Declare variable WinRef of type Item reference
Set reference WinRef to Libraryname.$windows.Windowname
; creates a reference to the window which you can use in your code
Do WinRef.$forecolor.$assign(kBlue)  ;; changes the window forecolor
```

You can enter the notation for an object in the initial value field for the item reference variable. You can also find the full notation for an object in the Notation Inspector and drag it to the notation field when you enter the *Set reference* command.

You can also use an item reference variable to return a reference to a new object, when using methods to create a new class, instance, or object. Furthermore OMNIS contains a special property called $ref which you can use to return an item reference to an object. Both these features are used in the section describing the *Do* command below.

Current Objects

Under $root, OMNIS contains a number of global state variables that tell you about how OMNIS is currently executing, or what objects, instances, and methods are currently being used. These objects provide a shortcut to the *current object* or instance that is currently executing. Mostly their names begin with "$c", and they include

- $cclass
  the current class
- $cdata
  the current open data file
- $cinst
  the current instance; usually the instance containing the currently executing method
- $cfield
  the field where the current method is executing
- $clib
  the current library
- $cmethod
  the current executing method
- $cobj
  the current object within a class or instance
- $crecipient
  the current recipient of an event; if a custom method is being processed, $crecipient is the recipient of that method
- $ctarget
  a reference to the target field, that is, the field which currently has the focus (shows the caret and is sent keyboard events)
- $ctask
  the current task; is usually the startup or default task until you open another task
- $cwind
  the current window instance
- $topwind
  the topmost open window instance

You can use the current objects in place of the full notation for a specific object to make the object and its code reusable and portable between libraries. For example, you can use $cinst in a method within a window instance to refer to itself, rather than referring to it by name

```
$cinst
; rather than
$root.$iwindows.WindowInstanceName
```

You can refer to the current library using $clib. For example, to make the current library private use

```
Do $clib.$isprivate.$assign(kTrue)
; is more generic than
Do $libs.MyLibrary.$isprivate.$assign(kTrue)
```

Do Command and Executing Methods

While you can use *Calculate* to change an object property or evaluate an expression, you can use the *Do* command for all expressions that execute some notation. In this respect, the *Do* command is the single-most powerful command in OMNIS. You can use the *Do* command to set the value of a property, or to run any standard or custom method. The *Do* command has several variants which include

– *Do*
sends a message to an object in your library, or assigns a value to an object property. Normally you should execute the *Do* command in the current object to execute one of its methods or assign to one of its properties. There are a number of common methods that you can use with the *Do* command including $open() to open an instance of a class, $assign() to change an object property, $redraw() to redraw an object, and so on

– *Do inherited*
executes the inherited method for the current method

– *Do default*
runs the default processing for a custom method

– *Do redirect*
redirects method execution to a custom method with the same name as the current method contained elsewhere in your library

– *Do method*
calls a method in the current class and returns a value

– *Do code method*
runs a method in a code class and returns a value

Note that you can display a list of built-in methods for an object or object group by clicking on the object in the Notation Inspector and opening the Property Manager. The methods for

an object are listed under the Methods tab in the Property Manager. See *OMNIS Studio* Help for a complete list of methods for all the objects in OMNIS. The Show Runtime Properties option in the Property Manager context menu lets you view properties that are normally available in runtime only, that is, properties of an instance rather than a design class. When runtime properties are visible in the Property Manager the methods for the instance are also shown. You cannot set runtime properties or use methods shown in the Property Manager, they are there as a convenient reference when you are writing code.

Do command

You can use the *Do* command in OMNIS to do almost anything: execute some notation, evaluate an expression, and so on. Specifically, you can use it to execute a method for an object or assign a value to one of its properties. The *Do* command returns a value to indicate whether the operation was successful or not, or for some methods a reference to the object operated upon. This section shows you how you can use the *Do* command and introduces some of the most useful methods.

$open() method

Using the *Do* command with the notation you can perform many operations that are otherwise performed with a command. For example, the class types that you can open contain an $open() method which you can execute using the *Do* command. For example, you can open a window using

```
Do $windows.WINDOWNAME.$open('INSTANCENAME',kWindowCenter)
; opens a window in the center of the screen
```

The $open() method returns a reference to the instance created. For example

```
; Declare variable WindRef of type Item reference
Set reference WindRef to LIB1.$windows.WindowName
Do WindRef.$open('WindowInstance') Returns WindRef
; WindRef now contains a reference to the window instance
; '$root.$iwindows.WindowInstance' which you can use elsewhere, e.g.
Do WindRef.$forecolor.$assign(kBlue)    ;; changes the instance
```

You can use a null value instead of an instance name; therefore CLASS.$open('') would force OMNIS to use the class name as the instance name. Alternatively you can use an asterisk in place of the instance name and OMNIS assigns a unique name to the instance, using the notation CLASSNAME_number. You can return the instance name in an item reference variable and use the reference in subsequent code. For example

```
; Declare variable iMenuRef of type Item reference
Do $menus.MCUSTOMERS.$open('*') Returns iMenuRef
; iMenuRef now contains a reference to the menu instance, which
; will be something like '$root.$imenus.MCUSTOMERS_23'
```

You can close an instance using the $close() method. For example, the following method opens a window instance, lets the user do something, and closes the instance

```
; initially WindRef contains a reference to the window class
Do WindRef.$open('WindowInstance') Returns WindRef
; let the user do something
Do WindRef.$close()
```

You can close the current window from inside the instance using

```
Do $cwind.$close()
```

Classes that contain the $open() methods also have the $openonce() method. This method opens an instance if one does not already exist (excluding window menus, window toolbars, and cascaded menus). In the case of a window, $openonce() brings the window to the top if it is already open. $openonce() returns an item reference to the new or existing instance, like $open().

$assign() method

You can change the properties of an object, including the properties of a library, class, or field, using the *Do* command and the $assign() method. The syntax for the $assign() method is NOTATION.PROPERTY.$assign(VALUE) where NOTATION is the notation for the object, PROPERTY is the property of the object you want to change, and VALUE is a value depending on the context of the object being changed. Usually you can use an OMNIS constant to represent a preset value, and for boolean properties, such as preferences, you can use kTrue or kFalse to set the property as appropriate. For example

```
Do $clib.$prefs.$mouseevents.$assign(kTrue)
; turns on mouse events for the current library
Do $cclass.$closebox.$assign(kTrue)
; adds a close box to the current window class
Do $cfield.$textcolor.$assign(kGreen)
; makes the text in the current field green
```

$add() method

You can create a new object in your library using the $add() method. In the notation you are really adding a new object to a particular group of objects. For example, to create a new field on a window you need to add the object to the $objs group of objects for the window, as follows

```
Do $cwind.$objs.$add(kPushbutton,iTop,iLeft,iHeight,iWidth)
; adds a pushbutton to the window with the
; specified size and position
```

When using $add(), you can return a reference to the new object in a return field of type item reference. You can use the reference to change the properties of the new object. For example

```
; Declare variable WindRef of type Item reference
Do $windows.$add('NewWindowName') Returns WindRef
;  now use the reference to change the new window
Do WindRef.$style.$assign(kPalette)
Do WindRef.$title.$assign('Window title')
Do WindRef.$clickbehind.$assign(kTrue)
Do WindRef.$keepclicks.$assign(kFalse)
Do WindRef.$modelessdata.$assign(kTrue)
Do WindRef.$backcolor.$assign(kRed)
Do WindRef.$forecolor.$assign(kWhite)
Do WindRef.$backpattern.$assign(2)
```

$redraw() method

When you change an object or several objects on an open window using the *Do* command, you often need to redraw the window. However if you change an object before $construct() completes execution for the window instance, you don't need to redraw the window. You can redraw an object, window, or all open windows using the $redraw() method. For example

```
Do $cfield.$redraw()
; redraws the current field
Do $cwind.$redraw()
; redraws the current window
Do $root.$redraw()
; redraws all window instances
```

$redraw(setcontents,refresh) redraws the contents and/or refreshes the field or window; 'setcontents' defaults to true, 'refresh' defaults to false.

$sendall() method

You can send a message to all the objects in a group using the *Do* command and the $sendall() method. For example, you can redraw all the objects in a group, you can assign a value to all the members of an object group, or you can hide all the members of a group using the $sendall() method and the appropriate message. The syntax for this method is $sendall(MESSAGE,CONDITION) where MESSAGE is the message you want to send to all the objects and CONDITION is a calculation which the objects must satisfy to receive the message. For example

```
Do $iwindows.$sendall($ref.$objs.FIELDNAME.$redraw())
; redraws the specified field on all window instances
Do $cwind.$objs.$sendall($ref.$textcolor.$assign(kYellow))
; makes the text yellow for all the fields on the current window
Do $cwind.$objs.$sendall($ref.$visible.$assign(kFalse),$ref.$order<=5)
; hides the first five objects on the current window; useful
; for window subclasses if you want to hide inherited objects
```

$makelist() method

Quite often you need to build a list containing the names of all the objects in a group, and you can do this using the makelist() method. For example

```
Do $clib.$classes.$makelist($ref.$name) Returns cLIST
; builds a list of all the classes in the current library and
; places the result in cLIST
Do $imenus.$makelist($ref.$name) Returns cLIST
; builds a list of all the currently installed menus
```

Do inherited

The *Do inherited* command runs an inherited method from a method in a subclass. For example, if you have overridden an inherited $construct() method, you can use the *Do inherited* command in the $construct() method of the subclass to execute the $construct() method in its superclass.

Do default

You can use the *Do default* command in a custom method with the same name as a standard built-in method to run the default processing for method. For example, you can use the *Do default* command at the end of a custom $print() method behind a report object to execute the default processing for the method after your code has executed.

Do redirect

You can use the *Do redirect* command in a custom method to redirect method execution to another custom method with the same name that is contained in another object in your library. You specify the notation for the instance or object you want execution to jump to.

Inheritance and custom methods are further discussed in the *Object Oriented Programming* chapter.

Calculate Command and Evaluating Expressions

This section describes how you use the *Calculate* command with an expression. It also discusses using square bracket notation for strings.

The *Calculate* command lets you assign a value to a variable calculated from an OMNIS expression. Expressions can consist of variables, field names, functions, notation strings, operators, and constants. For example

```
Calculate var1 as var2+var3
```

in this case, “var2+var3” is the expression.

```
Calculate var1 as con('Jon', 'McBride')
```

Here the expression uses the *con()* function which joins together, or concatenates, the two strings ‘Jon’ and ‘McBride’. You must enclose literal strings in quotes.

See the *OMNIS Studio* Help for a complete list of functions. In expressions, functions appear as the function name followed by parentheses enclosing the arguments to the function. The function returns its result, substituting the result into the expression in place of the function reference. Calling a function does not affect the flag.

The OMNIS operators are shown below, in precedence order, that is, the order in which they get evaluated by OMNIS. Operators in the same section of the table are of equal precedence, and are evaluated from left to right in an expression.

Parentheses	()
Unary minus	-
Multiplication	*
Division	/
Addition	+
Subtraction	-
Less than	<
Greater than	>
Equal to	=
Less than or equal to	<=
Greater than or equal to	>=
Not equal to	<>
Logical AND	&
Logical OR	|

When you combine expressions with operators, the order of expressions will often make a difference in the interpretation of the expression; this is a consequence of the mathematical properties of the operators such as subtraction and division. You can group expressions using parentheses to ensure the intended result. For example

```
Calculate lv_Num as 100 * (2 + 7)
```

evaluates the expression in parentheses first, giving a value of 900. If you leave off the parentheses, such as

```
Calculate lv_Num as 100 * 2 + 7
```

OMNIS evaluates the * operator first, so it multiplies 100*2, then adds 7 for a value of 207.

Square Bracket Notation

You can use a special notation in strings to force OMNIS to expand an expression into the string. You do this by enclosing the expression in square brackets; OMNIS evaluates the expression when the string value is required. You can use this in all sorts of ways, including the technique of adding a variable value to the text in the SQL or text buffer.

You can use square bracket notation wherever you can specify a single variable or field name, including

– command parameters, for example, *OK message*

```
OK message {Your current balance is [lv_curbalance]}
```

– window or report fields; you can include values in text objects, such as

```
Your current balance is [lv_curbalance]
```

– variable or field names within a *Calculate* command or text object

– function parameters

Square bracket notation lets you refer to a value indirectly letting you code general expressions that evaluate to different results based on the values of variables in the expression; this is called *indirection*. For example, you can include a variable name enclosed in square brackets in a text object to add the value to the text at runtime. However in general, there is a significant performance penalty in using indirection.

If you need to use [or] in a string but do not want the contents evaluated, then use [[and] to enclose the contents—double up the first or opening square bracket. This is useful when you use square bracket notation with external languages that also use square brackets, such as the VMS file system or DDE.

Type Conversion in Expressions

OMNIS tries its best to figure out what to do with values of differing data types in expressions. For example, adding a number and a string generally isn't possible, but if OMNIS can convert the string into a number, it will do so and perform the addition. Some other examples are

```
; Declare local variable lDate of type Date D m Y
Calculate lDate as 1200
; 1200 is no. of days since 31st Dec 1900
Calculate lDate as 'Jun 5 93'
; conv string to date in format D m Y
OK message {Answer is [jst(lDate,'D:D M CY')]}   ;; reformat date
Calculate lNum as lDate  ;; sets lNum to 1200, the no. of days
```

Boolean values have a special range of possibilities.

– YES, Y, or 1 indicate a true status

– NO, N, or 0 indicate a false status

FALSE and TRUE are not valid values; OMNIS converts them to empty.

```
; Declare local variable LBOOL of type Boolean
Calculate LBOOL as 1    ;; is the same as...
Calculate LBOOL as 'Y'  ;; or 'YES'
; the opposite is
Calculate LBOOL as 0  ;; or 'NO' or 'N'
OK message { The answer is [LBOOL] }
Calculate LBOOL as 'fui' ;; is the same as...
Calculate LBOOL as ''
```

You can convert any number to a string and any string that is a number in string form to a number.

```
; Declare local variable lChar of type Character
; Declare local variable lNum of type Number floating dp
Calculate lChar as 100
OK Message { [lChar], [2 * lChar], and [con(lChar,'XYZ')] }
; Gives message output  100 200 and 100XYZ
Calculate lNum as lChar
Calculate lChar as lNum
OK Message { [lChar], [lNum * lChar], and [con(lChar,'ABC')] }
; Gives message output  100 10000 and 100ABC
```

Constants

You will often find situations in OMNIS where you must assign a value that represents some discrete object or preset choice. OMNIS has a set of predefined constants you should use for this kind of data. For example, a class type can be one of the following: code, file, menu, report, schema, and so on. Each of these is represented by a constant: kCode, kFile, kMenu, kReport, kSchema, respectively. You can get a list of constants from the Catalog; press F9/Cmnd-9 to open the Catalog. You can use constants in your code, like this

```
Calculate obj1.$align as kRightJst    ;; or use Do
Do obj1.$align.$assign(kRightJst)
; aligns the object obj1 to the right
```

Although you can use the numeric value of a constant, you should use the predefined string value of a constant in your methods. In addition to ensuring you're using the right constant, your code will be much more readable. Moreover, there is no guarantee that the numeric value of a particular constant will not change in a future release of OMNIS.

Calling Methods

You can execute another method in the current class using *Do method*, or call a method in a code class using *Do code method*. These commands let you pass parameters to the called method and return a value in a return field. For example, the following method named Setup calls another method named Date and returns a value.

```
; Setup method
Do method Date (lNum,lDate+1) Returns lDate
OK Message {Date from return is [lDate]}

; Date method, the called method
; Declare Parameter var lpNum of type Number 0 dp
; Declare Parameter var lpDate of type Short Date 1980..2079
OK Message {Date from calling method is [lpDate], number is [lpNum]}
Quit method {lpDate + 12}
```

Note that when you call a code class method from within an instance the value of $cinst, the current instance, does not change. Therefore you can execute code in the code class method that refers to the current instance and it will work.

WARNING OMNIS does not stop a method calling itself. You must be careful how the method terminates: if it becomes an infinite loop, OMNIS will exhaust its method stack.

Quitting Methods

You can use the *Quit* command, and its variants, to quit methods at various levels.

– *Quit method*
quits the current method and can return a value

– *Quit event handler*
quits an event handling method

– *Quit all methods*
quits all the currently executing methods, but leaves OMNIS running

– *Quit all if canceled*
quits all methods if you press Cancel

– *Quit OMNIS*
exits your application and OMNIS

You can also clear the method stack with the *Clear method stack* command, which does the same thing as the debugger menu **Stack>>Clear Method Stack**; it removes all the methods

except for the current one. If you follow *Clear method stack* with *Quit method*, it has the same effect as *Quit all methods*.

Flow Control Commands

The *Constructs...* group contains many commands that let you control the execution and program flow of your methods. *If* statements let you test a condition and branch accordingly; loop commands iterate based on tests or sequences; the *Comment* command lets you comment your code; and reversible blocks let you manipulate objects and values and restore their initial values when the block terminates.

Several commands in this command group have starting and terminating commands (*If* and *End if*, for example). You must use the correct terminating command, or you will get unexpected results. If chromacoding is enabled, the beginning and terminating commands for most branching and looping constructs are highlighted. You can enable chromacoding using the View>>Show ChromaCoding menu option in the method editor.

Branching Commands

The *If* command lets you test the flag, a calculation, or a Cancel event. The Flag is an OMNIS variable with a True or False value which is altered by some commands to show an operation succeeded, or by user input. The *Else* command lets you take an alternative action when the *If* evaluates to false, *Else if* gives you a series of tests. You must use the *End If* command to terminate all *If* statements.

A simple test of the flag looks like this:

```
If flag true
   Do method Setup
End if
```

You can do a sequential checking of values using a calculation expression:

```
If CollCourse ='French'
   Do method Languages
Else If CollCourse = ‘Science’
   If CollSubCourse = ‘Biology’
      Do method ScienceC1
   Else
      Do method ScienceC2
   End If
Else
   OK message {Course is not available.}
End If
```

While Loops

The *While* loop tests an expression at the beginning of a loop. The *While* command will not run the code block at all if the expression is false immediately. You would use a *While* command when you want to loop while an expression is true.

```
; Declare Count with initial value 1
While Count <= 10
   OK message {Count is [Count]}
   Calculate Count as Count + 1
End While
```

This loop will output 10 messages. If the condition was ‘Count <= 1’, it would run only once.

Repeat Loops

A *Repeat* loop lets you iterate until an expression becomes true. Repeat loops always execute at least once, that is, the test specified in the *Until* command is carried out at the *end* of the loop, after the commands in the loop are executed, whereas While loops carry out the test at the beginning of the loop.

```
; Declare Count of Integer type with initial value 1
Repeat
   OK message {Count is [Count]}
   Calculate Count as Count + 1
Until Count >= 10
```

This loop will output 9 messages.

For Loops

The *For field value* command lets you loop for some specific number of iterations, using a specified variable as the counter. The following example builds a string of ASCII characters from their codes using the functions *con()* and *chr().*

```
; Declare Count
Calculate cvar1 as ''   ;; clear the string
For Count from 48 to 122 step 1   ;; set the counter range
   Calculate cvar1 as con(cvar1,chr(Count)) ;; add char to string
   Do $cwind.$redraw()
End for
```

The *For each line in list* command loops through all the lines in the current list.

```
Set current list LIST1
For each line in list from 1 to LIST1.$linecount step 1
   ; process each line
End for
```

Switch/Case Statements

The *Switch* statement lets you check an expression against a series of values, taking a different action in each case. You would use a *Switch* command when you have a series of possible values and a different action to take for each value.

The following method uses a local variable lChar and tests for three possible values, “A”, “B”, and “C”.

```
; Parameter pString(character 10)        ;; receives the string
Calculate lChar as mid(pString, 1, 1)    ;; takes the first char
Switch lChar
   Case ‘A’
      ; Process for A
   Case ‘B’
      ; Process for B
   Case ‘C’
      ; Process for C
   Default
      ; do default for all cases other than A, B, or C
End switch
```

It is a good idea to use the *Switch* command only for expressions in which you know all the possible values. You should always have one *Case* statement for each possible value and a *Default* statement that handles any other value(s).

Escaping from Loops

While a loop is executing you can break into it at any time using the *break key* combination for your operating system: under Windows this is Ctrl-Break, and under MacOS it is Cmnd-period. Effectively, this keypress ‘quits all methods’. When OMNIS performs any repetitive task such as building a list, printing a report, or executing a Repeat/While loop, it tests for this keypress periodically. For Repeat/While loops, OMNIS carries out the test at the end of each pass through the loop.

To create a more controlled exit for the finished library, you can turn off the *end of loop* test and provide the user with a working message with a **Cancel** button. When the **Cancel** button is visible on the screen, pressing the Escape key under Windows or Cmnd-period under MacOS is the equivalent to clicking **Cancel**. For example

```
Disable cancel test at loops    ;; disables default test for loops
Calculate Count as 1
Repeat
   Working message (Cancel box) {Repeat loop...}
   If canceled
      Yes/No message {Do you want to escape?}
      If flag true
         Quit all methods
      End If
   End If
   Calculate Count as Count+1
Until Count > 200
```

The *If canceled* command detects the Cancel event and quits the method. To turn on testing for a break, you can use the *Enable cancel test at loops* command.

The *Break to end of loop* command lets you jump out of a loop without having to quit the method, and the *Until break* provides an exit condition which you can fully control. For example

```
Repeat
   Working message (Cancel box) {Repeat loop...}
   If canceled
      Yes/No message {Are you sure you want to break out?}
      If flag true
         Break to end of loop
      End If
   End If
Until break
OK message {Loop has ended}
```

If you have not disabled the cancel test at loops, a Ctrl-Break/Cmnd-period terminates all methods and does not execute the *OK message*. Having turned off the automatic cancel test at loops, you can still cause a *Quit all methods* when canceled. For example

```
Disable cancel test at loops
Calculate Count1 as 1
Calculate Count2 as 1
Repeat
   Repeat
      Working message (Cancel box) {Inner repeat loop}
      Calculate Count2 as Count2 + 1
   Until Count2 > 12
   Calculate Count2 as 1
   Working message (Cancel box) {Outer repeat loop...}
   Quit all if canceled
   Calculate Count1 as Count1 + 1
Until Count1 > 20
```

If the user selects Cancel in the outer loop, the method quits, but from the inner loop there is no escape.

Optimizing Program Flow

Loops magnify a small problem into a large one dependent on the number of iterations at runtime, and other program flow commands can use a lot of unnecessary time to get the same result as a simpler command.

Here are some tips to help optimize your methods.

Use the *For* command instead of the equivalent *While* or *Repeat* commands. *For* has a fixed iteration, while the other commands test conditions. By eliminating the expression evaluation, you can save time in a long loop.

Use the *Switch* command instead of equivalent *If*/*Else* commands where possible. Arrange both the *Case* commands within a *Switch* and the several *If* and *Else* if commands so that the conditions that occur most frequently come first.

Use the *Quit method* command to break out of a method as early as possible after making a decision to do so. This can be a tradeoff with readability for long methods because you have multiple exits from the method; if falling through to the bottom of the method involves several more checks, or even just scanning through a large block of code, you can substantially improve performance by adding the *Quit method* higher up in the code.

Avoid using commands that don’t actually execute *within* a loop. For example, don’t put comment lines inside the loop. You can also use *Jump to start of loop* to bypass the rest of that iteration of the loop.

You can speed up a frequently called method by putting *Optimize method* at the start: refer to *OMNIS Studio* Help for details of this command.

Reversible Blocks

A reversible block is a set of commands enclosed by *Begin reversible block* and *End reversible block* commands; a reversible block can appear anywhere in a method. OMNIS reverses the commands in a reversible block automatically, when the method containing the reversible block ends, thus restoring the state of any variables and settings changed by the commands in the reversible block.

```
; commands...
Begin reversible block
   ; commands...
End reversible block
; more commands...
```

Reversible blocks can be very useful for calculating a value for a variable to be used in the method and then restoring the former value when the method has finished. Also you may want to change a report name, search name, or whatever, knowing that the settings will return automatically to their former values when the method ends.

The *OMNIS Studio* Help indicates which commands are reversible.

Consider the following reversible block.

```
Begin reversible block
   Disable menu line 5 {Menu1}
   Set current list cList1
   Define list {cvar5}
   Build window list
   Calculate lNum as 0
   Open window instance Window2
End reversible block
; more commands...
```

When this method terminates:

1. OMNIS closes window Window2
2. OMNIS restores lNum to its original value
3. The definition of cList1 returns to its former definition
4. OMNIS restores the former current list
5. OMNIS enables line 5 of Menu1

At the end of the method, OMNIS steps back through the block, reversing each command starting with the last. If there is more than one reversible block in a method, OMNIS reverses the commands in each block, starting from the last reversible block. If you nest reversible blocks, the commands in all the reversible blocks are treated as one block when

they are reversed, that is, OMNIS steps backward through each nested reversible block reversing each command line in turn. You cannot reverse any changes that the reversible block makes to OMNIS data files or server-based data unless you carefully structure the server transaction to roll back as well.

Error Handling

When you enter a command, OMNIS automatically checks its syntax. When a command is executed in a method, you can get a *runtime error*, a processing error rather than a syntax error. *Fatal errors* either display a message and stop method execution or open the debugger at the offending command.

You can cause a fatal error to occur with the *Signal error* command, which takes an error number and text as its argument. This lets you define your own errors, but still use the standard OMNIS error handler mechanism.

In addition, OMNIS maintains two global system variables #ERRCODE and #ERRTEXT that report error conditions and warnings to your methods. Fatal errors set #ERRCODE to a positive number greater than 100,000, whereas warnings set it to a positive number less than 100,000.

You can trap the errors and warnings by adding a method to test for the various values of #ERRCODE and control the way OMNIS deals with them; this is called an *error handler*. The command *Load error handler* takes the name of the method and an optional error code range as its parameters:

```
Load error handler Code1/1 {Errors}
; warnings and errors will be passed to handler in code class
```

Once you install it, OMNIS calls the error handler when an error occurs in the specified range. Please refer to the *OMNIS Studio* Help for a detailed description of the *Load error handler* command and examples of its use.

There are several commands prefixed with SEA, which stands for Set error action. Using these commands, you can tell OMNIS what to do after an error:

– *SEA continue execution*
  continues method execution at the command following the command that signaled the error; if the error handling routine has not altered them, #ERRCODE and #ERRTEXT are available to the command

– *SEA report fatal error*
  if the debugger is available, it displays the offending command in the method window and the error message in the debugger status line

– *SEA repeat command*
  repeats the command that caused the error.

Repeating a command should be done with care since it is easy to put OMNIS into an endless loop. If the error has a side effect, it may not be possible to repeat the command. If an ‘Out of memory’ condition occurs, it may be possible to clear some lists to free up enough memory to repeat the command successfully.

Redrawing Objects

There are a number of commands that let you redraw a particular object or group of objects. The Redraw command has the following variants.

– *Redraw* field or window
redraws the specified field or window, or list of fields or windows

– *Redraw lists*
redraws all list fields on the current window or redraws all lists in your library

– *Redraw menus*
redraws all the currently installed menus

– *Redraw toolbar*
redraws the specified custom toolbar

You can use the $redraw() method to redraw a field or fields, a window or all windows, as described earlier in this chapter.

Message Boxes

There are a number of message boxes you can use in your library to alert the user. The commands for these messages are in the *Message boxes...* group. They include

– *OK message*
displays a message in a box and waits for the user to click an OK button. For emphasis you can add an info icon and sound the system bell. You can use square bracket notation in the message text to display the current value of variables or fields. For example, *OK message {[sys(5)]}* will display your serial number

– *Yes/No message*, and *No/Yes message*
displays a message in a box and waits for Yes or a No answer from the user. Either the Yes or the No button is the default

– *Prompt for input*
displays a dialog prompting the user for input

– *Working message*
displays a message while the computer is processing data or executing a method; with a Cancel button the user can break into the processing with Ctrl-Break/Cmnd-period

Chapter 3—Debugging Methods

You can debug the methods in your library using the OMNIS *debugger*. The debugger is an integral part of the method editor. It helps you find errors by

- Running and stepping through methods
- Setting breakpoints
- Tracing execution of method lines and field values
- Viewing and altering fields and variables
- Inspecting the method stack
- Programming with debugger commands

The OMNIS debugger provides several tools to help you monitor the execution of a method, including the ability to create watch variables, interrogate and edit the contents of variables during execution, and place a variety of breakpoint conditions, which when met will interrupt execution.

The debugger operations are controlled from the Debug and Options menus on the method editor menubar. The debug options are also on the toolbar, which you can show using the View>>Toolbar menu option. The hierarchy of methods calling other methods is saved in the *method stack* and shown on the Stack menu.

You can also check your code using the *Method Checker*, available under the Tools menu and described in this chapter.

Executing a Method

You can open most class and field methods and run them from the debugger menu bar or toolbar. Note that event handling methods will not run from the *On* command without the event, but you can try out most types of methods while you're in design mode. You cannot execute methods that contain instance or task variables at design time since these variables are available when the objects are instantiated.

To run or execute a method

- Select Debug>>Go from the debugger menu bar

or

- Click on the Go button on the debugger toolbar

Execution will begin from the selected line. When you first open the method editor the first line of the first method is selected. You can halt execution by pressing the stop key combination Ctrl-Break/Cmnd-period. When you break into a method the debugger completes the current command and halts execution.

The basic debugging operations on the Debug menu are

– **Go** executes from the Go point

– **Step** executes from the Go point to the next method line, stepping into recipient methods

– **Step Over** runs from the Go point to the next method line, executing method calls, but not stepping into them

– **Trace** steps automatically through the method

– **Set Go Point** sets the current method line as the Go point

– **Go Point Not Set** indicates the method with the Go point when one is set

– **From Line** and **To Line** runs, steps or traces from the current line or to the current line

– **To Return** runs or traces to the return address in the calling method

– **Read Only Mode** prevents editing of methods

The Go Point

A method normally runs from the start, but you can start execution from any method line by setting it as the Go Point.

To set the Go point

- Double-click on the line

or

- Select the method line and choose the Debug>>Set Go Point menu option

or

- Select the method line and click the Set Go Point button on the toolbar

The debugger highlights this line and puts a yellow arrow in the left margin pointing to the method line where execution will begin. You can move around the program, changing the code, without changing the go point, which is independent of the current line. The name of the method containing the Go point is shown in the Debug menu and choosing this option from anywhere returns you to the Go point. You can clear the Go point using Stack>>Clear Method Stack.

Execution Errors

When an error occurs in a running method, OMNIS takes you into the debugger. The offending method is displayed with the go point at the method line that encountered the error, and an error message is shown in the status area. Error messages include the error number and text, for example “E108139: Set main file command with no valid file name.” You can use the various inspection tools to find out why the error occurred, fix it, and continue.

You can use the Debug>>From Line submenu to run the method from the currently selected line rather than the go point. The submenu items let you Go, Step, Step Over, or Trace from the current line instead of from the go point. The To Line submenu lets you Go or Trace from the go point to the current line, which becomes a one-time breakpoint.

Stepping through a Method

Normally when debugging you will want to step through the code rather than just run it. This gives much more control over when to start and stop methods and lets you examine fields, variables, and the method stack at specific points in the program. You use stepping in conjunction with breakpoints to control the debugging of your code.

To step through a method

- Choose Debug>>Step from the debugger menubar, or click on the Step In button

Every time you click on the Step In button, OMNIS executes the line at the go point and sets the go point to the next line. If a command at go point calls another method, the debugger loads the recipient method on the method stack and sets the go point to the first line in that method.

The Step In option steps into a recipient method. You can avoid this with Step Over where the debugger executes the recipient method without stepping into it. This speeds up debugging if you have a lot of method calls.

Tracing a Method

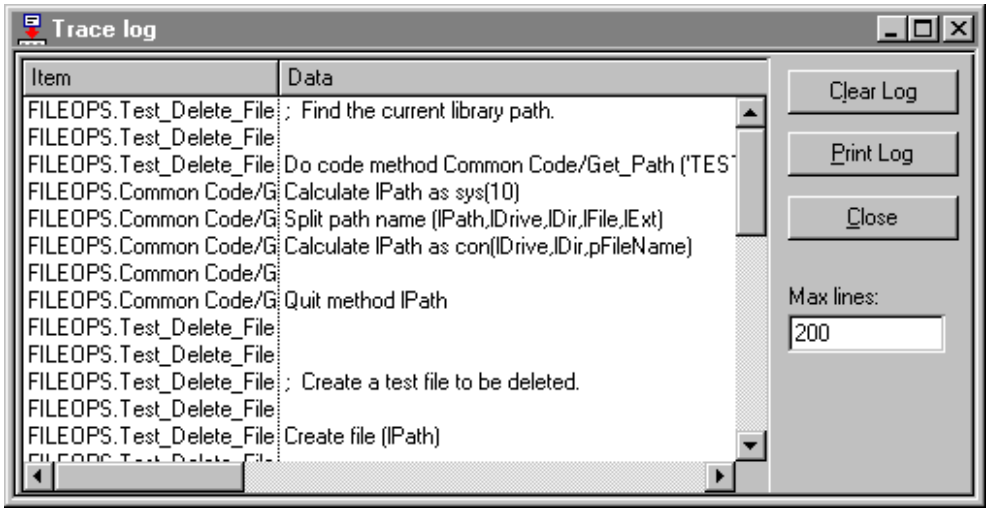
As well as stepping through your code, you can record or trace method execution.

To trace a method

- Choose Debug>>Trace from the debugger menubar, or click on the Trace button

The debugger steps through your code automatically, including stepping into recipient methods, and adds each method line and its parameters to a *trace log*.

You open the log from the Tools menu.



The first column in the trace log shows the name of the currently executing method, and the class it belongs to. The second column shows the method line and parameters of the currently executing command. When you double-click on a line in the trace log, the debugger goes to and highlights that method line.

You can open the trace log from within a method using the *Open trace log* command. For example, you can place the *Open trace log* command in the startup task of your library to trace method execution immediately after your library opens.

You can specify the maximum number of lines in the log in the Max lines entry field. When the log contains that many lines, it discards the earliest lines when new ones are added.

Private Methods

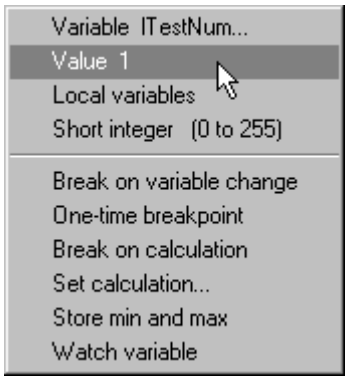
When you step or trace through the methods in your library the debugger will normally enter each method that is called, even if a method is in a private library. However if you set the library property $nodebug to true, the debugger will not display methods contained in private libraries. You need to set this property each time you open the library.

Inspecting Variable Values

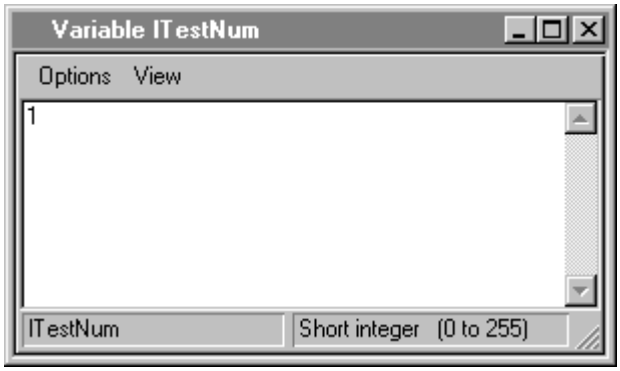
You can inspect the value of a variable or field using the variable context menu. This menu gives you full information about the variable or field and the class it belongs to, if any. You can display the variable context menu for any variable or field by right-clicking on its name in a method or the Catalog. Note that you cannot inspect the value of instance or task variables in design mode since the variables do not exist: in this case, the variables context menu is grayed out.

To display a variable context menu

- Right-click on the variable or field



This menu contains the variable name, value, parent group and data type, and a series of debugging options you can apply to the variable. If you click on a variable name in the variable pane of the method editor, the context menu has the Insert New and Delete Variable options as well. The other options at the bottom of the context menu are discussed under Breakpoints. The first option Variable opens the Variable Value window, except that for Item References with a value, it opens the Notation Inspector.



This window shows the variable name and type at the bottom and displays the value, which you can edit. OMNIS updates the value in the window whenever you bring the window containing the method to the top, but you cannot observe the value change dynamically through this window.

On the Variable Value window's View menu

– Redraw Values redraws the variable on any window

– Single Window Mode shows subsequent variable values in the same window

You can show and edit a list variable in a value window in just the same way. Note you cannot edit the binary variable.

The value window for a variable is valid only for as long as the variable is current.

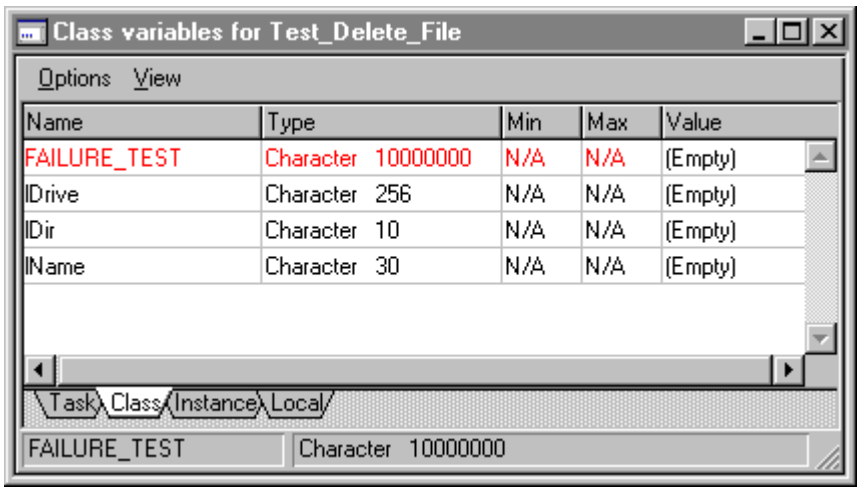
The Values List

In addition to the Value window for an individual variable you can show a Values List for whole groups of variables such as task variables, class variables, and local variables.

To show the Values List for class variables

- Right-click on a class variable name in a method or in the catalog
- Select the Class Variables option from the variable context menu

The Values List for class variables appears with the different variable types on tabbed panes.



On the View menu for the window

– Redraw Value redraws the variable wherever it occurs on a window

– Show Full Value opens a scrolling Value window below for the selected variable

The Variable popup menu for a file or schema class lets you modify the class, and for file classes only the Values List shows the current values for the file class.

Watching Variable Values

You can monitor or watch the value of a variable by making it a *watched variable*. You can add task, class, local, instance and parameter variables to the Watch variables pane in the method editor. When you run the debugger you can see the value of a watched variable change in the Watch variable pane.

To set a watch variable

- Right-click on the variable name and choose Watch Variable

or

- In the method editor, drag the variable from the variable pane into the watch pane

The Watch Variable item on the context menu is now checked. You can enlarge the watch pane by dragging its borders. The watch variable value is only updated when stepping, unless the method redraws it.

To remove a watch variable

- Right-click on the variable name and uncheck Watch Variable on the popup menu

Breakpoints

A *breakpoint* is a marker on a method line. When the debugger reaches that line, it stops execution and makes that line the go point.

To set a breakpoint

- Select the method line
- Choose the Breakpoint>>Breakpoint or click on the Breakpoint button

When you set a *breakpoint* for a line, a red dot appears in the left margin. A *one-time breakpoint* is a breakpoint that the debugger removes immediately after you break on it. It is marked by a blue dot in the margin.

When you close a library, you lose all breakpoints in the methods in that library. You can use the *Breakpoint* command in a method to set permanent breakpoints.

The Breakpoint menu lets you create and clear breakpoints.

– **Breakpoint** (Ctrl/Cmnd-Shift-B)
sets a full breakpoint at the current line

– **One-time Breakpoint** (Ctrl/Cmnd-Shift-O)
sets a one-time breakpoint at the current line

– **Clear Breakpoints** (Ctrl/Cmnd-Shift-C)
clears all the breakpoints

– **Clear Field Breakpoints** (Ctrl/Cmnd-Shift-F)
clears all the field change breakpoints, calculation breakpoints, and min and max settings (see below)

The rest of the **Breakpoint** menu is a list of all the current breakpoints. Choosing a breakpoint from this menu displays the line in the debugger.

You can also set breakpoints and from line to line execution, by right-clicking in the left margin of the method line or by using the appropriate tools on the toolbar.

Breaking on Variable Change

In the second half of the variable context menu, there is a group of breakpoint options that let you set breakpoints based on variable or field values.

The debugger only tests for variable or field breakpoints when methods are running, so a variable change during an enter data suspension of a method will be immediately reported if there is a control method and delayed otherwise. If there are several variable breaks at the same command, the debugger only displays one. Setting a variable value breakpoint slows down method execution considerably.

The menu option **Break on Variable Change** tells the debugger to stop the method when the variable value changes. The debugger puts a check mark against the line. Reselecting the same line toggles the break off. The status line displays the text ‘Break on variable change (field)’ when the break occurs.

The **One-Time Breakpoint** option puts a single-stop variable change breakpoint on the line.

Breaking on Calculation

You can also create a variable value breakpoint with a calculation. For example, to stop a method when a local variable lvLines becomes equal to the number of lines in list cvList, the calculation is entered as

```
lvLines = lst(cvList,cvList.$linecount)
```

The menu option **Break On Calculation** sets the breakpoint, and the following line **Set Calculation** prompts for the calculation. The debugger treats the calculation value as a boolean value where zero corresponds to No and anything else corresponds to Yes.

Execution breaks when the calculation evaluates to Yes, but with a qualification: the break happens only when the calculation changes from No to Yes. This means that if the calculation is always Yes, the break never happens; it also means that the break happens only when the change is from No to Yes, not every time the calculation evaluates to Yes.

For example, the calculation break

```
(lvNumber<10) | (lvNumber>20)
```

ensures that local variable lvNumber stays within the range 10-20.

Each variable or field can have one calculation breakpoint. There is no requirement that the calculation refers to the variable.

The **Store Min And Max** option adds the minimum and maximum variable values to the end of the menu as execution proceeds, along with the item **Clear Min and Max** that lets you turn off the feature. If you choose either menu item, OMNIS writes a line to the trace log. Turning on **Store Min And Max** slows down the debugger a good deal.

The Method Stack

A stack is a list of things that you can access in a last-in, first-out manner. When you call a method, OMNIS pushes the current method onto the method stack of executing methods. The debugger adds each new method to the **Stack** menu in the method editor. The top-most menu item is the latest method, the one below it called it, and so on. When a method returns, OMNIS removes the top item, also known as popping the stack, and goes to the calling method. You can examine any method on the stack by selecting it. You can also move up and down the stack with the **Stack** menu items **Move Up Stack** and **Move Down Stack**.

If you select a method in a different class while holding down the **Shift** key, the debugger opens a new method design window.

When you stop in a method with a breakpoint, an error, a step, or an explicit stop, OMNIS sets the go point to the next method line and saves the stack. It marks the commands in the methods on the stack that will execute when you return to that method with a gray arrow in the left margin pointing to the method line where execution will resume.

A method can appear more than once in the method stack with a completely different set of local variables.

Debug>>To Return runs or traces the method from the go point or current line until it returns control to the method that called it. If the only method on the stack is the current method, this option is grayed out.

There are times when you may want to throw away the current stack and start over. For example, if you follow a problem to its conclusion and everything freezes up, you can

restart by clearing the stack. You do this with **Stack>>Clear Method Stack**, which also grays out the **To Return** item and removes the Go point.

Debugger Options

The debugger **Options** menu appears with the other debugger menus.

- **Debug Next Event**
  stops at the first line of a method executed for an enter-data event with a control method (a field method, a window control method or a timer method). Note this option is not saved with other debugger options and defaults to off whenever OMNIS is started
- **Trace All Methods**
  sets trace mode permanently on.
- **Open Trace Log**
  opens the trace log window or brings it to the top
- **Disable Debugger at Errors**
  stops OMNIS from breaking into the debugger on program errors; this is what the end user of your application would see
- **Disable Debugger Method Commands**
  deactivates any debugger commands in the methods
- **Save Debugger Options** saves all the debugger options, and Revert To Saved Options reverts back to the last saved set of debugger options

Debugger Commands

You can control the debugger using the commands in the Debugger... group. These commands effectively disappear when you set **Options**>>**Disable Debugger Method Commands**. See the *OMNIS Studio* Help for a complete description of the following commands.

Breakpoint breaks the program when OMNIS executes it. If you specify a message, it appears on the status line when the break happens. *Trace on* switches trace mode on, optionally clearing the trace log, and *Trace off* switches trace mode off.

Send to trace log adds a new line to the trace log containing the specified text. The text can contain square bracket notation. You can then use the log as a notepad for your comments, variable or field values, and bookmarks in the code. When the methods are run, double-clicking on trace log lines opens the design windows at the appropriate points in the methods. See the Send value to trace log option for the *Variable menu command* below.

The *Variable menu command* applies a variable context menu option to a list of variables. The list has the same format as *Define list*, and for fields can include file names and so on. This command has several options.

- *Set break on field change*
  sets a field change breakpoint for each field in the list
- *Clear break on field change*
  clears any field change breakpoints for each field in the list or all breakpoints if you don't specify a field list
- *Set break on calculation*
  sets a calculated breakpoint for each field in the list; set the calculation for each field with *Set break calculation*
- *Clear break on calculation*
  clears any calculated breakpoints for each field in the list or all calculated breakpoints if you don't specify a list
- *Store min and max*
  stores minimum and maximum values for each field in the list
- *Do not store min and max*
  clears store min and max mode for each field on the list or all modes set if you don't specify a list
- *Add to* watch variables *list*
  adds each specified field to the watch variables pane
- *Remove from* watch variables *list*
  removes each specified variable from the watch variables pane or all variables if you don't specify a list. Variables with breakpoints or with store min and max mode set always appear on the watch variables list
- *Send value to trace log* adds a line to the trace log for each field on the list; if you don't specify a list, adds a line for all fields
- *Send minimum to trace log* adds a line to the trace log with the minimum for each field on the list for which the debugger is storing minimums; if you don't specify a list, adds a line for all such fields
- *Send maximum to trace log* adds a line to the trace log with the maximum for each field on the list for which the debugger is storing maximums; if you don't specify a list, it adds a line for all such fields
- *Send all to trace log* adds a value line to the trace log for each field on the list; also adds minimum and maximum lines to the trace log for each field on the list for which *Store min and max* is set; with no list, adds a line for all appropriate fields

– *Open value window* opens a value window for each field on the list; with no list, opens a window for all fields with whatever limit the operating system puts on the number of window instances

– *Open values list* opens a values list containing the value for each field on the list; with no list, opens a values list for all fields, subject to the operating system limit on the number of window instances. There is one values list for each file class so if more than one field name from a particular file class appears in the list, OMNIS displays only one values list for that file class

– *Set break calculation* sets up the calculation for the field breakpoint

Checking Methods

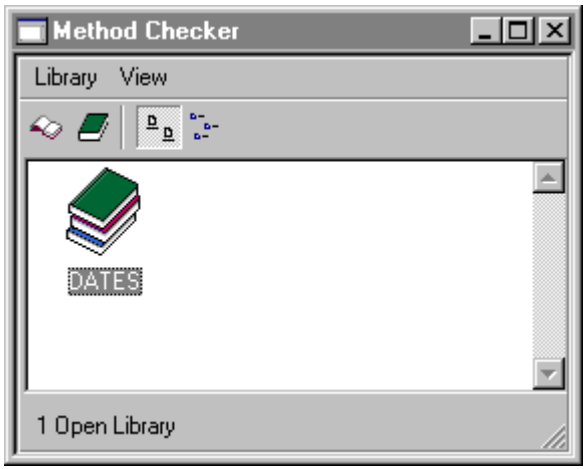
You can check the methods in your library using the *method checker*. The method checker is available under the Tools menu on the main OMNIS menu bar. It checks your code for syntax errors, unused variables, methods without code, and so on. It provides various levels of checking and reports errors in individual classes or all classes in the current library. It is particularly useful for checking libraries converted from an earlier version of OMNIS.

Note that the method checker does not correct the code in your libraries automatically, it simply reports any errors and potential problems in your code.

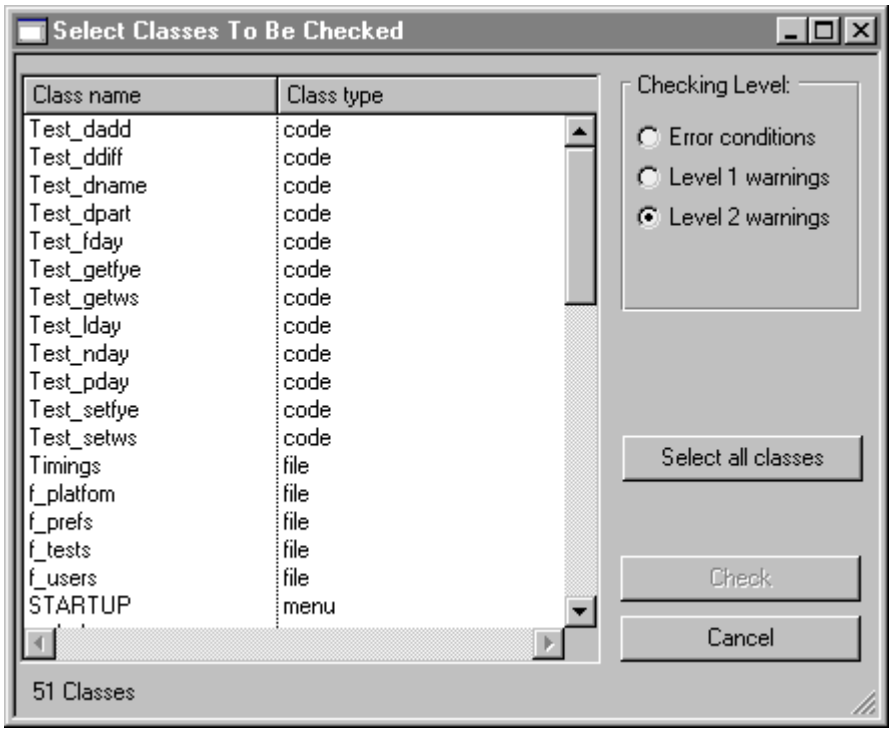
When you open the method checker it loads all libraries that are currently open. Alternatively you can open a particular library from within the method checker.

To check the methods in your library

- Select the Tools>>Method Checker menu item from the main OMNIS menu bar



- If you need to load a library, click on the Open Library button on the method checker menu bar
- Double-click on the library you want to check



- Shift-click or Ctrl/Cmnd-click to select the classes you want to check, or click on the Select all classes button to select them all

The following checking levels are available

– **Error conditions**
this level of checking finds problems that can cause runtime errors or undesired behavior; you *must* fix these errors

– **Include Level 1 warnings**
finds problems that you should investigate because they might result in subtle bugs and strange behavior; you *ought to* fix these problems

– **Include Level 2 warnings**
finds problems that you should be aware of, including empty methods and/or inefficient code, potential compatibility problems, and platform-dependent code

The different levels of checking are *inclusive*, that is, if you select Level 2 Warnings (the default) this includes Level 1 and the Errors categories.

- Select a checking level, and click on the Check button

The method checker works through the classes you selected displaying their statistics in the Method Checker Error Log. You can cancel checking at any time by pressing Ctrl-Break/Cmnd-period.

When checking is complete, you can sort the log by clicking on one of the headers in the list. You can print the log or save it to a text file.

You can show a summary of the errors for each class by clicking on the Show Summary button.

Interpreting Errors and Warnings

The following sections detail the different levels of errors and the possible action you should take.

Fatal Errors

These are the type of errors that you *must* fix.

Encountered a construct End without a construct Begin
An ending construct was found without a matching beginning:

– *End if, End switch, End while, End for, Until, End reversible block*

Method contains a construct Begin without a construct End
A beginning construct was encountered without the proper ending:

– *If, Switch, While, For, Repeat, Begin reversible block*

Construct End does not match construct Begin
An ending construct was encountered that did not match the beginning construct, e.g. Begin reversible block followed by an End if.

Encountered a construct element in an invalid context
One of the following was found outside of a proper construct: *Else, Case, Default*

Encountered a command in an invalid context
One of the following commands was found outside of a proper construct: *Break to end of switch* outside of a Switch construct, or *Break to end of loop* or *Jump to start of loop* outside of a *For, While,* or *Repeat* loop.

Incomplete command
A command with no parameters set, for example, *Set reference* with no reference, *Set current list* with no list name, *Call method* with no method name.

Invalid field reference
An invalid reference to a field or variable (i.e. #???) was encountered: usually a reference to a field or variable that has been deleted.

Invalid method reference
Encountered a command containing a reference to a non-existent method, an unnamed method, or a method in a library that is not currently open. For example, *Call method* with name of non-existent method, *Enable menu line* with reference to non-existent menu or menu line.

Missing extended command or function
A missing extended command or function was encountered, either not loaded or installed: these show in your code beginning with the letter "X".

Bad library name
The library name contains one or more periods.

Level 1 warnings

These are the type of problems that you *ought to* fix.

Class variable with the same name as a library variable
Could cause precedence problems at the class level.

Optimize method command not in first line of method
The *Optimize method* command should be the first line of a method.

Code in an unnamed method
Named method with no code
Check to see if this code/method is required.

Debugging code?
You should remove all breakpoints before deploying your application. One of the following was encountered: *Breakpoint, Trace on/off, Field menu command, Set break calculation, Send to trace log.*

Debugging message?
Either *OK message* or *Sound bell* was encountered: remove messages inserted for debugging purposes.

Obsolete command
You should not use obsolete commands: remove them from your code. For example you can replace *Call method* with *Do method* or *Do code method*.

Command removed by converter
In converted libraries certain commands are commented out: you should use another command or use the equivalent notation.

Level 2 warnings

These are the type of problems that you should investigate that might require fixing.

Unused variable
Variable defined but unused, or referenced and not defined.

Unfriendly code: Code which could affect other libraries if running in a multi-library environment
For example, *Clear method stack, Quit all methods, Close all windows, Remove all menus.*

Unfriendly code: Code which would cause the current library to be closed
The following commands will close the current library if the "Do not close other libraries" is not set: *Open library, Create library, Prompt for library.*

Class name specified in an internal method call
Inefficient code.

Code that modifies a library or class
One of the Classes... group of commands, such as *New class, Rename class, Delete class.*

Platform-dependent code
Functions which return different values depending on which platform they are executed, including sys(6), sys(10) to sys(22), sys(103) to sys(114).

Comment containing a question mark
Usually indicates code that needs to be tested, completed, or fixed.

Reference to hash variable
Avoid using hash variables: replace with variable of appropriate scope.

Chapter 4—Object Oriented Programming

The *Introduction* in the *Using OMNIS Studio* manual describes the basic concepts of object-orientation used in OMNIS, and the *OMNIS Tools* chapter describes the tools you use to access these features. This chapter deals specifically with the more advanced object-oriented features including inheritance, custom properties and methods, and creating and using object classes and external objects.

Inheritance

When you create a new class you can derive it from an existing class in your library. The new class is said to be a subclass of the existing class, which is in turn a superclass of the new class. You can make a subclass from all types of class except code, schema, file, and search classes. The general rule is that if you can open or instantiate a class, you can make a subclass of that type of class. OMNIS does not support mixed inheritance, that is, you cannot make a subclass of one type from another type of class.

When you make a subclass, by default, it inherits all the variables, methods, and properties of its superclass. Window subclasses inherit all the fields and objects on the window superclass, and menu and toolbar subclasses inherit all menu lines and tools from their respective superclass.

Inheritance saves you time and effort when you develop your application, since you can reuse the objects, variables, and methods from a superclass. When you make a change in a superclass, all its subclasses inherit the change automatically. From a design point of view, inheritance forces uniformity in your GUI by imposing common properties, and you get a common set of methods to standardize the behavior of the objects in your library.

Making a Subclass

You can make subclasses from the following types of class.

– **Window**
inherits variables, methods, and properties from its superclass, as well as all fields on the window superclass

– **Menu**
inherits variables, methods, and properties from its superclass, as well as the menu lines in the menu superclass

– **Toolbar**
inherits variables, methods, and properties from its superclass, as well as the toolbar controls in the toolbar superclass

– **Report**
inherits variables, methods, and properties from its superclass: note that a report class *does not* inherit the fields, sections, and graphics from its superclass

– **Task**
inherits variables and methods from its superclass, but none of its properties

– **Table**
inherits variables and methods from its superclass, and only some of its properties

– **Object**
inherits variables and methods from its superclass, but none of its properties

To make a subclass in the Browser

- Open your library in the Browser and show its classes
- Select the class and choose Class>>Make Subclass from the Browser menu bar

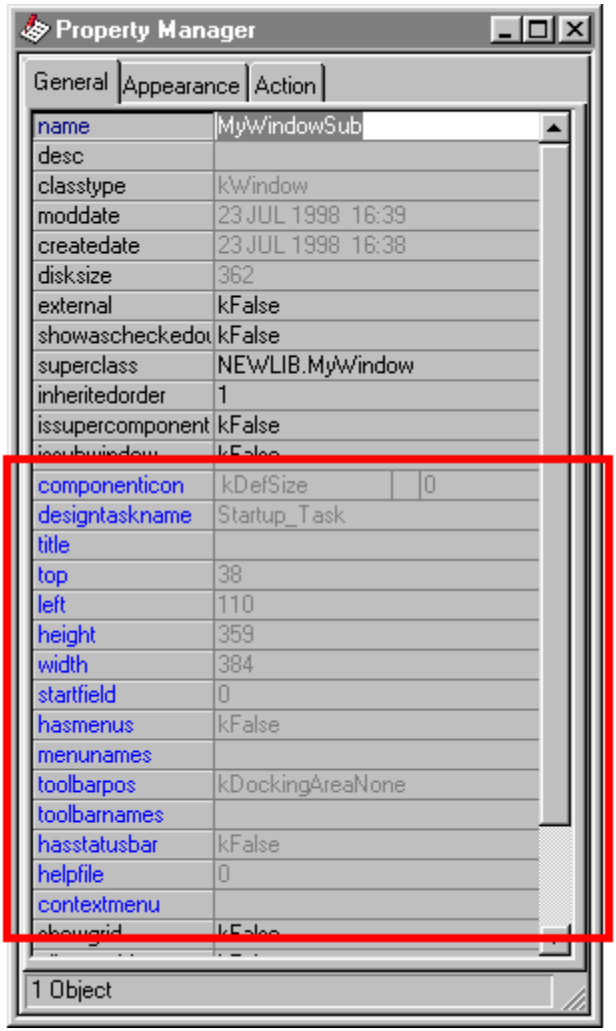
or, you can

- Right-click on a class and choose Make Subclass from its context menu

When you make a subclass OMNIS creates a new class derived from the selected class. The new class inherits all the objects, variables, and methods from its superclass. OMNIS supports up to 10 superclass levels, that is, a single class can inherit objects from up to ten other superclasses that are directly in line with it in the inheritance tree. If you create further levels of subclass they do not inherit the objects from the superclasses at the top of the tree.

You can view and edit a subclass, as you would any other class, by double-clicking on it in the Browser. When you edit the methods for a subclass in the method editor, you will see its inherited variables and methods shown in a color. When you view the properties of a subclass its inherited properties are shown in a color in the Property Manager. You can set the color of inherited objects using the **inheritedcolor** OMNIS preference.

The following screen shot shows the inherited and non-inherited properties for a window class: note that all the Appearance and Action properties for a window are inherited too.



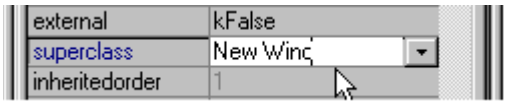
The properties at the top of the Property Manager are standard properties for the class and are non-inherited, so too are the grid properties. The properties to do with the general appearance of the window, such as **title** and **hasmenus**, are inherited and shown in a color, which defaults to bright blue. You cannot change inherited properties unless you overload them: their values are grayed out in the Property Manager. If you change the properties of the superclass, the changes are reflected in the subclass when you next open it in design mode.

There are some general properties of a class that relate to inheritance. These are

superclass	the name of the superclass for the current class, prefixed with the library name: the superclass can be in another library
inheritedorder	for window classes only, determines where the inherited fields appear in the tabbing order: by default inherited fields appear before non-inherited fields on your window
issupercomponent	if true the class is shown in the Component Store as a superclass; when you drag such a class from the Component Store you create a subclass of the class
componenticon	icon for the superclass when it is loaded in the Component Store

Making a Subclass Manually

You can set the **superclass** property for a class manually, to make it a subclass of the specified class, either using the notation or in the Property Manager.



However when you make a subclass in this way it *does not inherit* any of the properties of the superclass. Only objects are inherited from the superclass. You have to open the Property Manager and inherit the properties manually using a context menu.

To inherit a property manually

- View the properties of the subclass in the Property Manager
- Right-click on the property and select Inherit Property from its context menu

If the property cannot be inherited the context menu will display Cannot Inherit and will be grayed out. If the class does not have a superclass the inheritance context menu does not appear or is grayed out.

To inherit a method manually

- Open the method editor for the subclass
- Right-click on the method and select Inherit Method from its context menu

When you inherit a method in this way, OMNIS will delete the current method.

Overloading Properties, Methods, and Variables

Having created a class from another class using inheritance you can override or overload any of its inherited properties, methods, and variables in the Property Manager or the method editor as appropriate. All inherited objects are shown in a color. To overload an object, you can Right-click on the object and select Overload from the object's context menu. Note that for windows, menus, and toolbars you cannot overload or delete inherited fields, menu lines, or toolbar controls. If you don't want certain objects to be displayed in a subclass you can hide them temporarily at runtime using the notation.

To overload a property

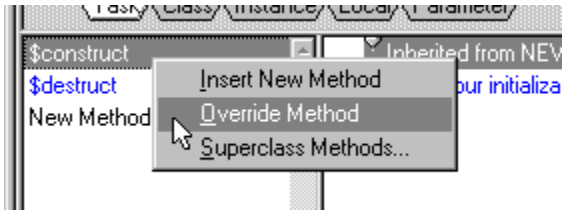
- View the properties of the subclass in the Property Manager
- Right-click on the inherited property and select Overload Property from its context menu

When you overload a property its inherited value is copied to the class and becomes editable in the Property Manager. You can overload any number of inherited properties in the class and enter values that are local to the class.

To reverse overloading, you Right-click on a property and select Inherit Property: the local value will be overwritten and the value inherited from the superclass will appear in the Property Manager.

To override a method

- View the methods for the subclass in the method editor
- Right-click on the inherited method and select Override Method from its context menu

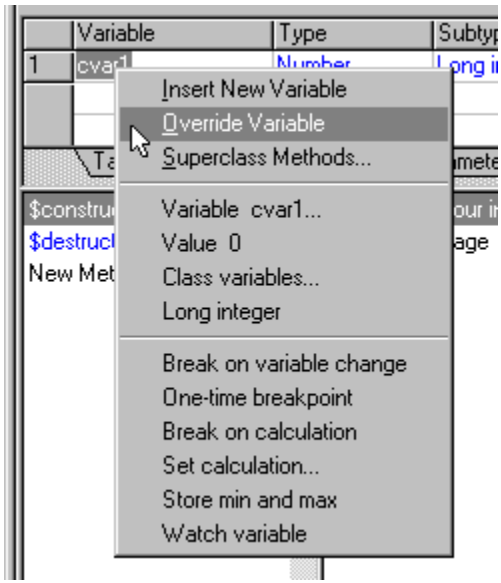


When you override a method it becomes like a non-inherited method in the class, that is, you can select it and add code to it.

To reverse this process and inherit the method with the same name from the superclass, you can right-click on the method and select Inherit Method: the code in the non-inherited method will be deleted and the inherited method will now be in place. Alternatively, if you override a method and then delete the local one, the inherited method will reappear when you close and reopen the class.

To override a variable

- View the methods for the subclass in the method editor and click on the appropriate tab in the variable pane to find your variable
- Right-click on the inherited variable and select Override Variable from its context menu



When you override a variable it becomes like a non-inherited variable in the class and is only visible in that class.

To reverse this process and inherit the variable with the same name from the superclass, you can Right-click on the variable and select Inherit Variable.

Inheritance Tree

The Inheritance Tree shows the superclass/subclass structure of the classes in your library. All classes below a particular class in the hierarchy are subclasses of that class. When you select a class in the Browser and open the Inheritance Tree it opens with that class selected, showing its superclasses and subclasses above and below it in the tree.

To open the Inheritance Tree for a class

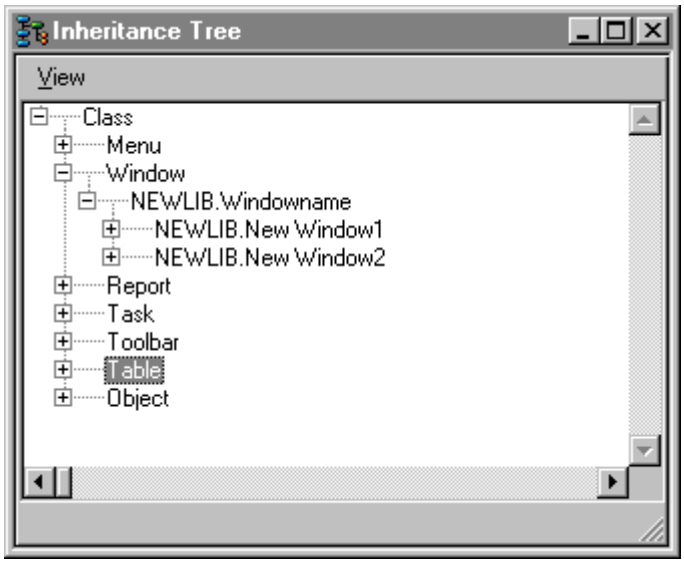
- Select the class in the Browser
- Choose Class>>Inheritance Tree from the Browser menu bar

or

- Choose View>>Inheritance Tree from the main menu bar, in which case you will need to navigate to the class, since it will either come to the front as it was if already open, or open at the default top level

or, you can

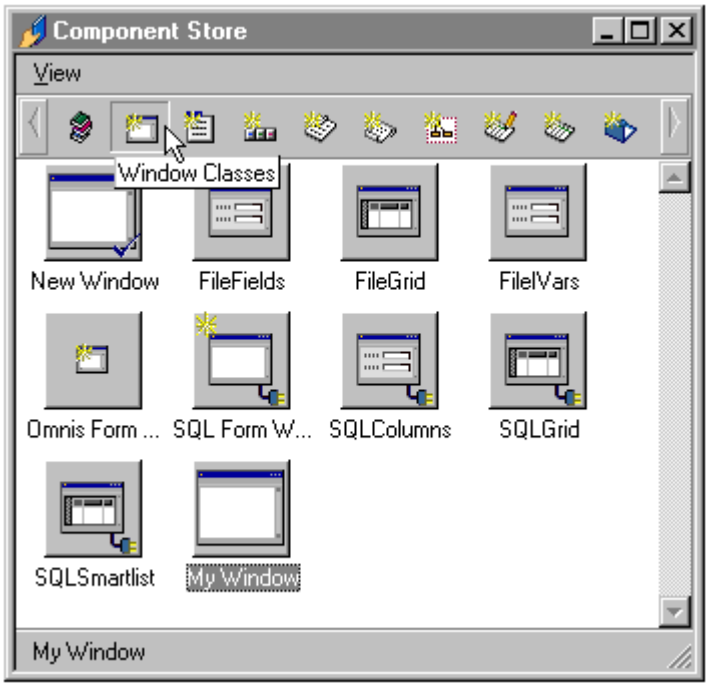
- Right-click on the class and select Inheritance Tree from its context menu



Showing Superclasses in the Component Store

You can show any class that supports inheritance in the Component Store by setting its **issupercomponent** property. You can specify the icon for a superclass by setting its **componenticon** property. If you create a class from a superclass displayed in the Component Store, by dragging its icon on to your library, the new class will be a subclass of the class in the Component Store automatically.

For example if you create a window called ‘My Window’ and set its **issupercomponent** property to true, it will appear in the Component Store under the Window Classes button.



When you drag the ‘My Window’ component on to your library, OMNIS creates a new window that is a subclass of ‘My Window’. The window superclass will only appear in the Component Store if the library containing it is open, since the class actually remains in your library and is not physically copied to the component library.

Note that classes that appear in the Component Store in this way cannot be made the default object for that class.

Inheritance Notation

You can use the $makesubclass() method to make a subclass from another class. For example

```
Do $windows.Window1.$makesubclass('Window2') Returns ItemRef
; creates Window2 which is a subclass of Window1 in the
; current library, ItemRef contains a reference to the new class
Do $windows.Window1.$makesubclass('Window2','MyLibrary') Returns
   ItemRef
; creates Window2 which is a subclass of Window1 in
; the library called MyLibrary
```

You can test if the current class can be subclassed by testing the $makesubclass() method with the $cando() method, as follows

```
If $cclass.$makesubclass().$cando()
   Do $cclass.$makesubclass(sClass,tLibName) Returns ItemRef
   ; creates a subclass of the current class in the
   ; library held in tLibName
```

You can test if a particular class is a superclass of another class using the CLASS.$isa(SUPERCLASS) method as follows

```
Do $windows.window2.$isa($windows.window1) Returns lresult
; returns true if window1 is a superclass of window2
```

You can change the superclass of a class by reassigning $superclass

```
Do $cclass.$superclass.$assign('DiffClassName') Returns lresult
```

You can test if a property can be inherited using

```
Do $cclass.$PropertyName.$isinherited.$canassign() Returns lresult
```

A superclass can be in a different library to a subclass. If you open an instance of a subclass when the superclass is not available, perhaps because the library the superclass belongs to is not open or has been renamed, a kerrSuperclass error is generated.

Calling Properties and methods

When a property or method name is referenced in a subclass, OMNIS looks for it first in the subclass and progressively up the chain of superclasses in the inheritance tree for the current library. Therefore if you haven't overridden the property or method in the subclass, or at any other level, the property or method at the top of the inheritance tree will be called. For example, the following command in a subclass

```
Do $cinst.$MethodName()
; will call $MethodName() in its superclass
```

However, if you have overridden a property or method in the subclass the method in the subclass is called. You can still access the inherited property or method using the $inherited property. For example, assuming $MethodName() has been overridden in the subclass

```
Do $cinst.$inherited.$MethodName()
; will call the subclass method
Do $windows.MySubWin.$MethodName().$inherited
; will call $MethodName() in the superclass
```

Referencing Variables

When a variable is referenced in a subclass, OMNIS looks for its value first in the subclass and progressively up the chain of superclasses in the inheritance tree for the current library. Therefore if you haven't overridden the variable in the subclass, or at any other level, the value at the top of the inheritance tree is used.

However, if you have overridden a variable in the subclass the value in the subclass is used. You can access the inherited variable using $inherited.VarName, in which case, the value at the top of the inheritance tree is used.

A superclass cannot use the instance and class variables of its subclasses, although the subclass can pass them as parameters to superclass methods. References to class and instance variables defined in a superclass are tokenized by name and the Remove Unused Variables check does not detect if a variable is used by a subclass. If an inherited variable does not exist, its name is displayed as $cinst.VarName in design mode and will create a runtime error.

Inherited Fields and Objects

All inherited window fields on a window subclass are included in the $objs for an instance. Since some field names may not be unique, when $objs.name is used OMNIS looks first in the class containing the executing method, then in the superclasses and lastly in the subclasses. The $objs group for report fields, menu lines, and toolbar controls behave in the same way as window fields.

You should refer to fields at runtime by name, since at runtime OMNIS assigns artificial $idents to inherited fields. The $order property of a field may also change at runtime to accommodate inherited fields into the tabbing order.

Do inherited Command

You can use the *Do inherited* command to run an inherited method from a method in a subclass. For example, if you have overridden an inherited $construct() method, you can use the *Do inherited* command in the $construct() method of the subclass to execute the $construct() method in its superclass. You could use this command at the end of the $construct() method in the subclass to, in effect, run the code in the subclass $construct() and then the code in the superclass $construct() method.

Custom Properties and Methods

You can add methods to the objects in your library and call them what you like; you execute these methods from within the class or instance using the *Do method* command. However you can create your own properties and methods and execute them using the notation, as you would the standard properties and methods. These are called *custom properties* and *custom methods*, or collectively they are referred to as *custom attributes*. The name of a custom attribute is case-insensitive and can be anything you like, but it must begin with the dollar "$" sign, such as "$xyz".

Any class that can be instantiated can contain custom attributes, including window, report, table, and object classes. In practice you can use custom attributes to override the behavior of the standard attributes, or to add your own properties and methods to an object. Custom attributes can only be executed at runtime, in an instance of the class.

If the name of a custom property or method is the same as a standard one, such as "$printrecord()", it will override the standard one. However, if you create a custom property or method with the same name as a common attribute, such as $name, the common attribute takes precedence over your custom attribute.

You create custom properties and methods for an object in the method editor. You enter custom properties for a field in the Field Methods for the field, and for a class in the Class Methods for a class. The code for a custom property or method can be structured like any other method; it can modify the current instance, or calculate and return a result.

An instance of a class contains whatever custom attributes you define in the class, together with the properties and methods for that type of instance. The object group $attributes contains all the built-in and custom properties and methods for an instance. You can use $first() and $next() against $attributes, but $add() and $remove() are not available.

You can reference a custom attribute using the notation NOTATION.$xyz, where NOTATION is some notation for an instance of a class and "$xyz" is the name of your custom attribute. If you specify parameters, such as NOTATION.$xyz(p1,p2,p3), they are passed as parameters to the custom attribute, and a value may be returned.

You can use the *Do default* command within the code for a custom attribute to execute the default behavior for a property or method with the same name as the custom attribute. You can use the *Do redirect* command to redirect execution from a custom attribute in one instance to another instance containing a custom attribute with the same name.

To create a custom method

- Open the Class or Field methods for your class
- Right-click on the method names list, and select Insert New Method from the context menu
- Enter a name for your custom method, including a dollar sign at the beginning of its name
- Enter the code for the custom method as you would any other method

Using Custom Methods

The following example uses a task class containing a custom method called $printprinter(). You can call this method from anywhere inside the task instance using

```
Do $ctask.$printprinter()
```

The $printprinter() method sets the print destination and calls another class method depending on whether the user is accessing SQL or OMNIS data; it contains the following code

```
; $printprinter() custom method
Begin reversible block
   Send to printer
   Set report name REPORT1
End reversible block
If iIsSQL
   Do method printSQLData
Else
   Begin reversible block
      Set search name QUERY1
   End reversible block
   Do method printOMNISData
End If
```

The next example uses a window that contains a pane field and a toolbar with three buttons. When the user clicks on a button, the appropriate pane is selected and various other changes are made. Each button has a $event() method that calls a custom method called $setpage() contained in the window class. Note that you can send parameters with the custom method call, as follows

```
; $event() method for second toolbar button
Do $cwind.$setpage(2)
```

The $setpage() custom method contains a parameter variable called pPage, and has the following code

```
; $setpage() custom method
Switch pPage
   Case 1
      Do $cwind.$objs.MainPane.$currentpage.$assign(1)
      Do $cwind.$title.$assign('Queries')
   Case 2
      Do $cwind.$objs.MainPane.$currentpage.$assign(2)
      Do $cwind.$toolbars.$add('tbModify1')
      ; installs another toolbar
      Do $cwind.$title.$assign('Modifying')
   Case 3
      Do $cwind.$objs.MainPane.$currentpage.$assign(3)
      Do $cwind.$menus.$add('MReports')
      ; installs a menu in the window menu bar
      Do $cwind.$title.$assign('Reports')
   Default
      Quit method kFalse
End Switch
```

The final example uses a window containing a subwindow, which in turn contains a tree list. The subwindow contains a custom method called $buildtree() that builds and expands the tree list. You can call the $buildtree() method from the parent window and send it parameters, using the notation

```
Do $cwind.$objs.SubWin.$buildtree(lv_ClassList)
```

The $buildtree() method contains a parameter variable called pv_SourceList of List type that receives the list passed to it, and a reference variable called TreeRef set to the tree list field, and contains the following code

```
; $buildtree() custom method
Do TreeRef.$setnodelist(kRelationalList,0,pv_SourceList)
Do TreeRef.$expand()
```

Object Classes

Object classes let you define your own structured data objects containing variables and methods. You can create an object variable based on an object class which contains all the variables and custom methods defined in the class. When you reference an object variable an instance of the object class is created containing its own set of values. You can store an object variable instance and its values on a server or OMNIS database. The structure and data handling capabilities of the object instance is defined by the types of variables you add to the object class; similarly, the behavior of an object variable is defined by the methods you add to the object class.

Object classes have the general properties of a class and no other special properties. They can contain class and instance variables, and your own custom methods. You can make a subclass from an object class, which inherits the methods and variables from the superclass.

To create an object class

- Open your library in the Browser
- Display the classes in your library using the View>>Down One Level menu option on the Browser menu bar
- Drag the template called “New Object” from the Component Store onto the Browser

or

- Select Class>>New>>Object from the Browser menu bar
- Name the new object class
- Double-click on the object class to modify it

When you modify an object class, OMNIS opens the method editor for the class. This lets you add variables, and your own custom properties and methods to the class.

When you have set up your object class you can create any other type of OMNIS variable based on your object class: an object variable has type Object and its subtype is the name of your object class. For example, you can create an instance variable of Object type that contains the class and instance variables defined in the object class. When you reference a variable based on an object class you create an instance of that object class. You can call the methods in the object class with the notation ObjVarName.$MethodName(), where $MethodName() is any custom method you define in the object class.

You can store object variable instances in an OMNIS data file, or in a server database that stores binary values. When you store an instance of an object variable in a database, the value of all its contained instance variables are also stored. When the data is read back into

memory the instance is rebuilt with the same instance variable values. In this respect you can store a complete record or row of data in an object variable.

You can store object instances in a list. Each line of the list will have its own instance of the object class. Object instances stored in a task, instance, local, or parameter variable belong to the same task as the instance containing that variable. Similarly object instances stored in a list or row belong to the same task as the instance containing the list or row variable. All other object instances have global scope and are instantiated by the default task and belong to the default task.

You cannot make an object instance a private instance. If you delete the object class an object variable uses, the object instance will be empty.

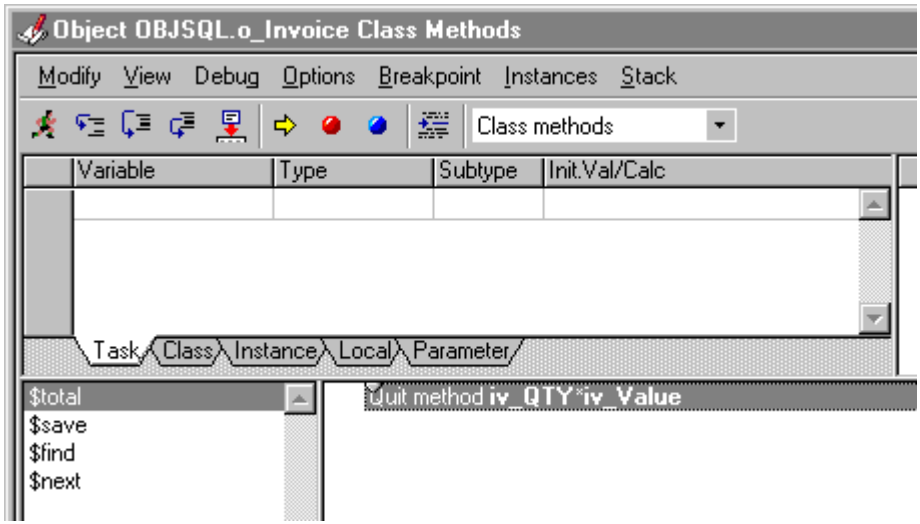
To add variables and methods to an object class

- Open your object class in design mode
- Right-click in the variable pane of the method editor and select the Add New Variable option from the context menu
- Name the variable, give it a type and subtype as appropriate
- Right-click in the Method Names pane of the method editor and select the Add New Method option from the context menu
- Name the method, including the dollar prefix

If you right click on an object variable, and use the Variable <name>... entry in the context menu, the variables list window opens, initially showing instance variable values.

Using Object Classes

This section describes an invoices example and uses an object class and simple invoices window; it is intended to show what you can do with object classes, not how to implement a fully functional invoices application. You can create the data structure required for an invoice by defining the appropriate variables and custom methods in an object class. The following screenshot shows the object class called o_Invoice, which contains the variables you might need in an invoice, such as the invoice ID, quantity, value, and a description of the invoice item.



The o_Invoice object class also contains any custom methods you need to manipulate the invoice data, such as inserting or fetching invoices from your database. The methods in the object class contain the following code

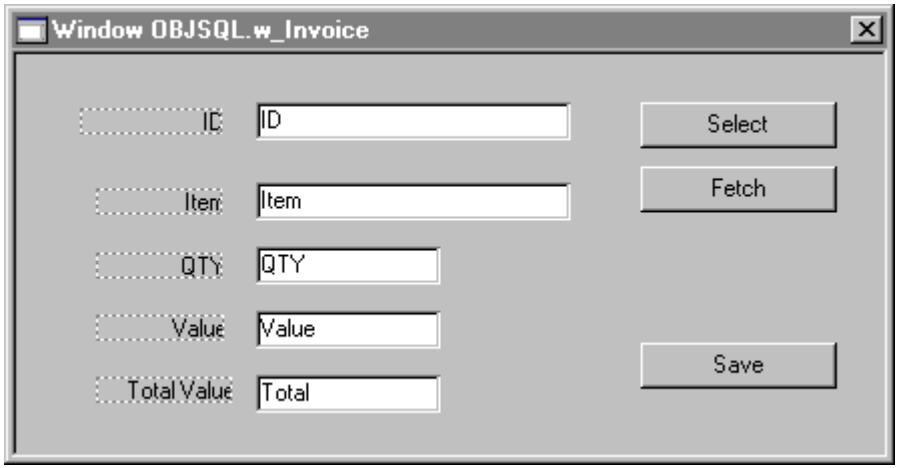
```
; $SaveInvoice() method contains
; local var lv_InvoiceRow of type Row and
; local var lv_Bin of type Binary and
; parameter var pv_Object of type Field reference
Do lv_InvoiceRow.$definefromtable(t_Invoices)
Calculate lv_bin as pv_Object
Calculate lv_InvoiceRow.InvoiceObject as lv_bin
Do lv_InvoiceRow.$insert() Returns #F
```

```
; $SelectInvoice() method contains
; local var lv_Row of type Row
Do lv_Row.$definefromtable(t_Invoices)
Do lv_Row.$select()
Do lv_Row.$fetch() Returns #S1
Quit method lv_Row.InvoiceObject
```

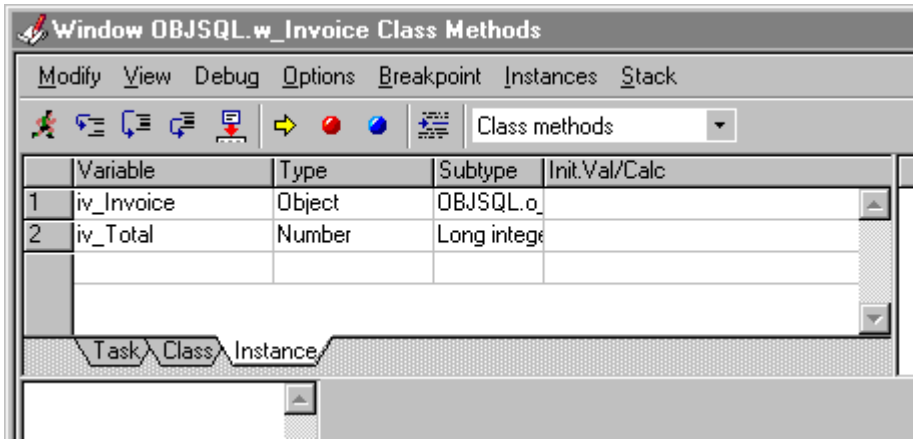
```
; $FetchInvoice() method contains
; local var lv_Row of type Row
Do lv_Row.$definefromtable(t_Invoices)
Do lv_Row.$fetch()
Quit method lv_Row.Inv_Object
```

```
; $total() method
Quit method iv_QTY * iv_Value
```

The invoice window can contain any fields or components you want, but would contain certain fields that reference the instance variables in your object class, and buttons that call the methods also in your object class. The window might look something like this



The invoice window contains no class methods of its own. All its functionality is defined in the object class. The window contains a single instance variable called iv_Invoice that is based on the o_Invoice object class.



Note that the instance variable has type Object and its subtype is the name of your object class prefixed with the library name, OBJSQL.o_Invoice in this case.

When you open the invoice window an instance of the object class is created and held in iv_Invoice. Therefore you can access the instance variables and custom methods defined in the object class via the instance variable in the window; for example, iv_Invoice.iv_QTY accesses the quantity value, and iv_Invoice.$SaveInvoice() calls the $SaveInvoice() method. Each field on the invoice window references a variable in the object class; for

example, the dataname of the quantity field is iv_Invoice.iv_QTY, the dataname of the item or description field is iv_Invoice.iv_Item, and so on.

The buttons on the invoice window can call the methods defined in the object class, as follows.

```
; $event() method for the Select button
On evClick
   Do iv_Invoice.$SelectInvoice(iv_Invoice.iv_ID) Returns iv_Invoice
   Do $cwind.$redraw()
; $event() method for the Fetch button
On evClick
   Do iv_Invoice.$FetchInvoice Returns iv_Invoice
   Do $cwind.$redraw()
; $event() method for the Save button
On evClick
   Do iv_Invoice.$SaveInvoice(iv_Invoice)
```

When you enter an invoice and click on the Save button, the $SaveInvoice() method in the object class is called and the current values in iv_Invoice are passed as a parameter. The $SaveInvoice() method receives the object instance variable in the parameter pv_Object and executes the following code

```
Do lv_InvoiceRow.$definefromtable(t_Invoices)
Calculate lv_bin as pv_Object
Calculate lv_InvoiceRow.InvoiceObject as lv_bin
Do lv_InvoiceRow.$insert() Returns #F
```

The row variable lv_InvoiceRow is defined from the table class t_Invoices which is linked to the schema class called s_Invoices which contains the single column called InvoiceObject. The binary variable lv_bin, which contains the values from your object instance variable, is assigned to the row variable. The standard $insert() method is executed which inserts the object variable into your database. The advantage of using an object variable is that all the values for your invoice are stored in one variable and they can be inserted into a binary column in your database via a single row variable. If you want to store object variables in an OMNIS database you can create a file class that contains a single field, called InvoiceObject for example, that has Object type, rather than Binary, and use the appropriate methods to insert into an OMNIS data file.

Dynamic Object Instances

The object class has a $new() method that lets you create an object instance dynamically and store it in an object variable, for example

```
Do $clib.$objects.objectclass.$new(parm1,parm2,...)
   Returns objectvar
```

where parameters parm1 and parm2 are the $construct() parameters for the object instance. When the instance is assigned, any existing instance in the object variable is destroyed. It would be normal practice to put no class in the variable pane for object variables which are to be set up dynamically using $new(), but there is no class checking for instance variables so no error occurs if the class shown in the variable pane is different from the class of the new instance.

You can do a similar thing with an external function library if it contains instantiable objects, such as Fileops. For example

```
Do Fileops.$objects.fileops.$new() Returns objectvar
Do objectvar.$openfile(pFilename)
```

The following example uses an Object variable ivSessionObj which has no subtype defined in the method editor. When this code executes ivSessionObj is instantiated based on the object class SessionObj which was created using the Session Wizard in OMNIS. Once the object instance exists the $logon() method is called.

```
Do $clib.$objects.SessionObj.$new() Returns ivSessionObj
; runs $construct() in the SessionObj object class
Do ivSessionObj.$logon()
; runs the $logon() method in SessionObj
```

Self-contained Object Instances

Object classes have the $selfcontained property. If set to kTrue the class definition is stored in all instances of the object class. An instance of such an object class is disconnected from its class and instead relies on its own class data for method and instance variable definitions. When the object instance is stored on disk the class definition is stored with the instance data and is used to set up a temporary class whenever the instance is read back into memory. Any changes to the original object class have no effect on existing instances, whether on disk or in memory.

Once an instance is self-contained it is always self-contained, but you can change its class definition by assigning to $class, for example

```
Do file1.objvar1.$class.$assign($clib.$objects.objectclass1)
```

causes the class definition stored inside objvar1 to be replaced by the current method and variable definitions for objectclass1. The instance variable values are maintained provided the new instance variable definitions are compatible with the old ones (OMNIS can handle

small changes in the type of variables but won't carry out substantial conversions). Note that the old instance variables are matched to the new ones by $ident and not by name, so to avoid problems the new class should be a descendant of the original class with none of the original variables having been deleted.

Only the main class is stored with the object instance, inheritance is allowed but any superclasses must exist in open libraries whenever the instance is present in memory. Assigning to $class does not change the superclass structure of self-contained instances.

External Objects

External objects are a type of external component that contain *methods* that you can use by instantiating an object variable based on the external object. External objects can also contain *static functions* that you can call without the need to instantiate the object. These functions are listed in the Catalog under the Functions pane.

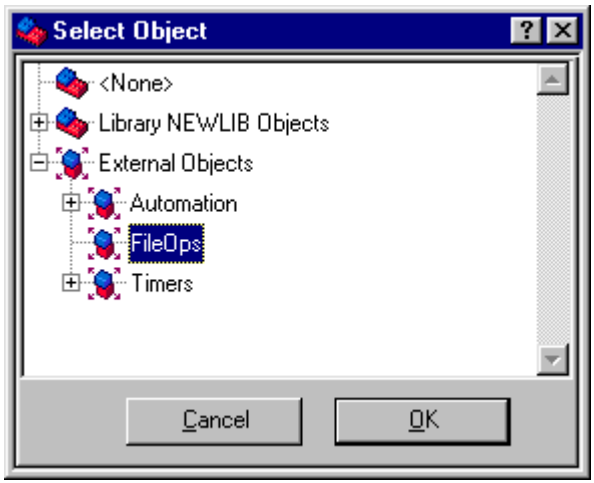
Some external objects are supplied with OMNIS Studio; these include equivalents to the FileOps and FontOps externals, and a Timer object. Writing your own external objects is very similar to writing external components, which is described on the OMNIS website. The FileOps and FontOps functions are documented in the OMNIS Help.

External objects are created and stored in component libraries in a similar manner to external components, and in future releases are intended to replace external functions and commands, although to maintain backward compatibility, the old external interface is still supported at present.

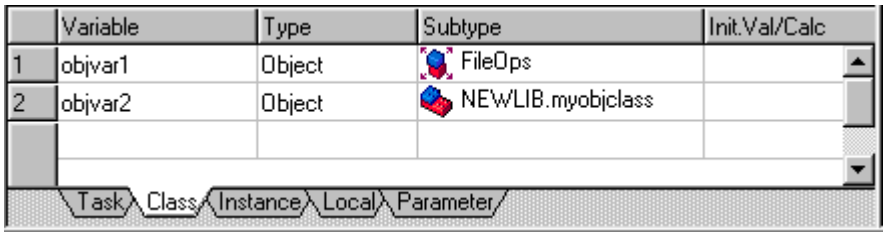
External object libraries are placed in the XCOMP folder, along with the visual external components. They must be loaded in the same way as external components using the External Components dialog, available from the Library>>External Components menu item in the Browser. See the *Window Classes* chapter of *Using OMNIS Studio* for more details about loading external components.

Using External Objects

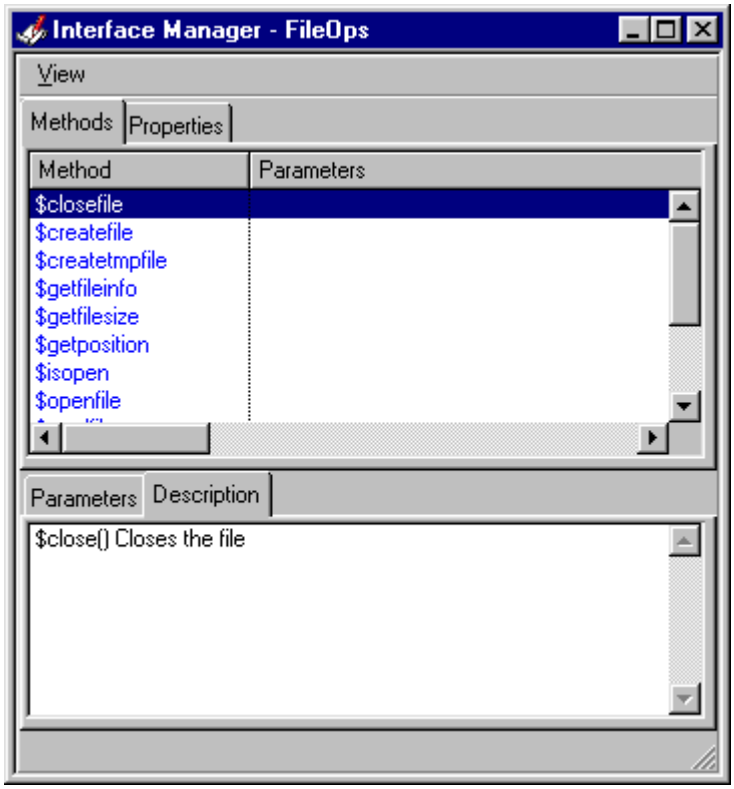
You can add a new object in the method editor by inserting a variable of type Object and using the subtype column to select the appropriate external object. You can click on the subtype droplist and select an external object from the Select Object dialog. This dialog also appears when you create an object elsewhere in OMNIS, such as the file class editor.



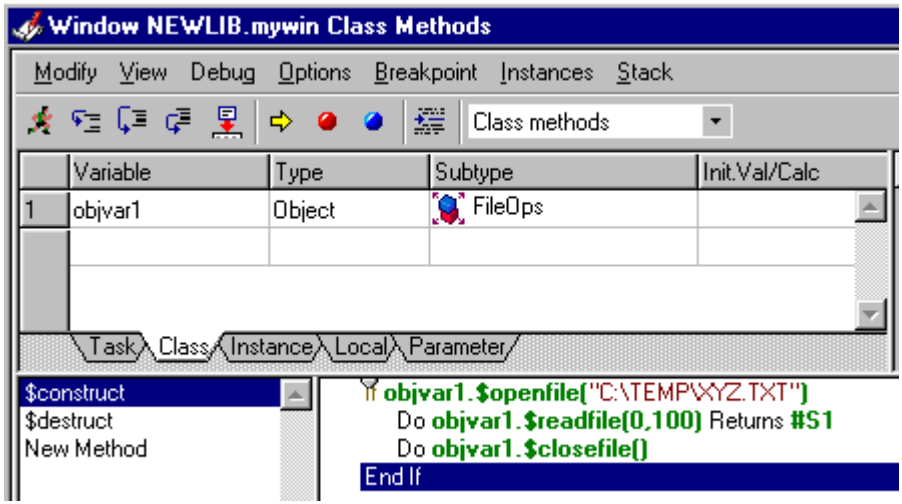
An icon in the variable subtype cell shows whether the variable is based on an object class or an external object. For example, objvar1 is a variable based on an external object, objvar2 is based on an object class in the current library.



When an instance of the external object has been constructed, you can inspect its properties and methods using the Interface manager.



To use the object's methods in your code, you can drag the method you require from the Interface Manager into the command parameters box. Your code could look like this:



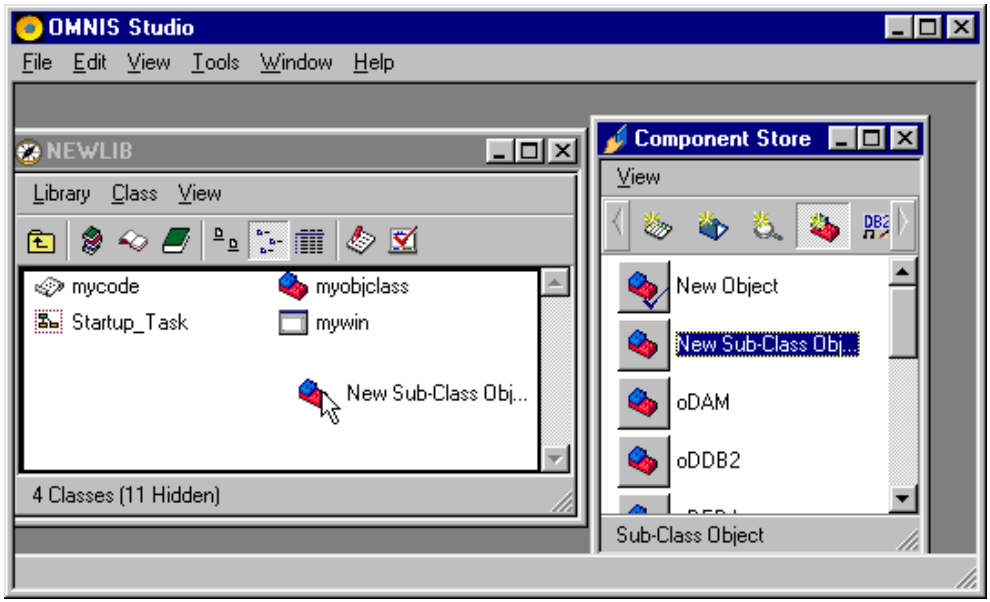
For some objects it is important to note that for the Interface manager to interrogate an object it will need to be constructed. For example if the Interface Manager was used on an Automation object, the Automation server needs to be started.

External objects are contained in the notation group $extobjects, which you can omit from notation.

External Object Events

External objects do not support events in the GUI sense. They can however define notification methods which they call when certain events occur.

You can subclass an external object and then override the notification method so your code is informed of the event. The Timer object supplied in OMNIS is an example of this. To subclass an object, you can either set the superclass property in the Property Manager, or use the New Subclass Object wizard. You can drag the wizard from the Object classes pane in the Component Store on to the Browser.



Interface Manager

The Interface Manager displays the public methods and properties for objects in OMNIS Studio, that is, any class that can contain methods and can be instantiated, including window, menu, toolbar, report, task, table, and object classes (not available for code classes). Furthermore, for window, report, toolbar, and menu classes the Interface Manager displays the methods for the objects in the class. For each class or object, the Interface Manager displays all built-in methods, including those available in the instance of the class, as well as any custom methods you have added to the object.

Private methods, namely methods with a name that does not begin with a dollar sign, are not included in the Interface Manager since these methods are confined to the class or instance.

For each method in the class or object, the Interface Manager displays the method name, its parameters, return types, and a description, if any are present.

You can view the Interface Manager from several places in OMNIS, including the Browser, method editor, and from various context menus.

To view the Interface Manager

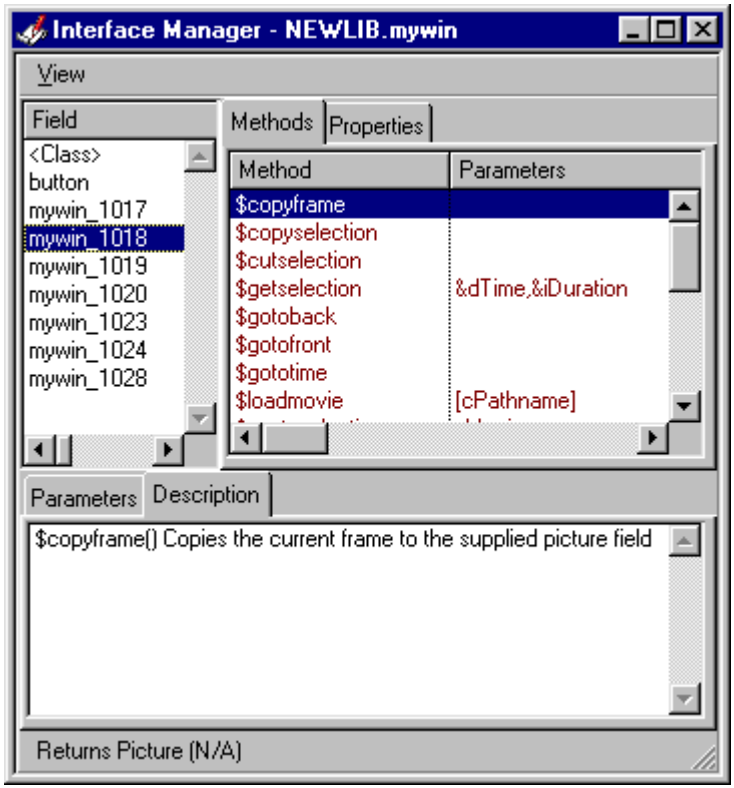
- Click on the class in the Browser
- Select Class>>Interface Manager from the Browser menubar

or you can

- Right-click on the class in the Browser and select Interface Manager from the context menu

or from the method editor

- Open the method editor for the class
- Select View>>Interface Manager from the method editor menubar



The Interface Manager contains a list of objects in the class, that is, for windows and reports a list of window or report fields, for toolbars a list of toolbar controls, and for menus a list of menu lines. For other types of class or instance that do not contain objects, such as object classes, the Interface Manager contains the class methods only. You can click on each object or field in the left-hand list to view its methods. Built-in methods are shown in the color specified in the $nosetpropertycolor preference. Inherited methods are shown in the color specified in the $inheritedcolor preference. The properties tab similarly shows the object's properties.

The Details pane shows the parameters for the currently selected method. It also lets you add a description for your own custom methods. The status bar shows the return type for built-in methods, but not for your own methods, since these can return any type.

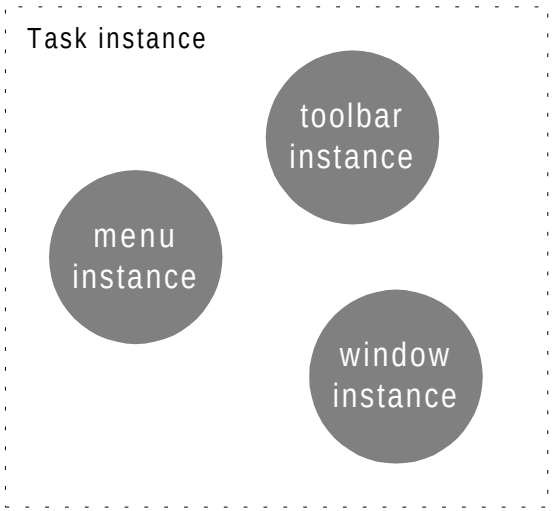
The View menu on the Interface Manager menubar lets you open the method editor for the class, in addition to hiding or showing the built-in methods and details pane.

Dragging methods from the Interface Manager

You can drag a method or property from the method list and drop it on to an edit field in the method editor, or you can use copy and paste to do the same thing. The method name is prefixed by a variable name, such as “var_name.$methodname()” if you opened the Interface Manager by right-clicking on a variable of type Object. Otherwise the method name is prefixed by a dot, such as “.$methodname()”, suitable to concatenate onto a variable name or some notation in the method editor. In all cases the parameters for the method are copied too, so they can be used as a template for the actual values you pass to the method.

Chapter 5—Using Tasks

OMNIS contains two environments, a *design mode* and a *runtime mode*. In design mode, you can create and store classes in your library. In runtime mode, various objects or instances are created as you run your application. You can group and control the runtime objects in your application by opening or instantiating them in a *task*. You can manipulate whole groups of instances by manipulating their task, rather than having to deal with each separate instance. You define a task in your library as a *task class*.



OMNIS contains and opens a default task for your application to run in, but you can add your own tasks. OMNIS provides the tools to create, manage and destroy tasks.

Task classes can contain variables and methods, and you can define custom properties and methods for a task class as well. When you open a task you create an instance of that task class. The task instance is unique in that it can contain other instances including window, report, toolbar, and menu instances. Task instances cannot contain other tasks. When you open an instance from within a task it *belongs to* or is *owned by* that task.

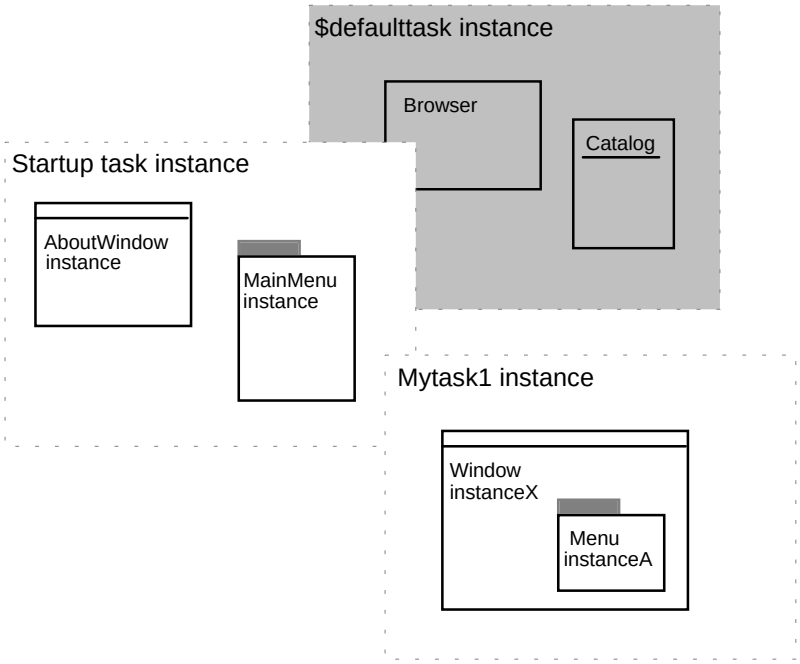
By opening and closing different tasks, or by switching from one task instance to another, you can control whole groups of objects. OMNIS provides certain default actions which happen as the task context switches. You define exactly what happens in the task by creating methods in the task class. For example, in the task you can specify which windows are opened and which menus are installed using commands or the notation.

Each library contains a group of task classes called $tasks, and OMNIS has a group containing all open task instances called $root.$itasks in the order that they were opened.

Default and Startup Tasks

When OMNIS opens, it creates a task instance for the IDE to run in. This task is called the *default task*, and is represented in the notation as $root.$defaulttask. This task instance contains all the IDE objects such as the Browser, Catalog, Property Manager, and so on.

When you create a new library, it contains a task class called *Startup_Task* by default. When you open a library, an instance of the startup task is created automatically. From thereon all instances opened in the library are owned by the startup task. You can delete the startup task, or you can create other tasks for your application components to run in.



It is not essential to add tasks to your library, your library will safely execute in the startup task, or the default task along with the IDE objects.

The startup task instance has the same name as your library. For a simple application, the startup task will probably be all you need, with all the other class instances belonging to it. The startup task remains open for as long as the library is open, but you can close it at any time using a command or the notation. You can change the name of the task to be opened on startup by setting the library preference $startuptaskname; for all new libraries this is set to Startup_Task by default.

If you have an application that spans multiple libraries, often only the library used to start the application will have a startup task. If a library is opened using the *Open library* command with the option *Do not open startup task*, the startup task is not instantiated. In design mode, you can stop a library's startup task from running if you hold down the Alt/Option key as you open your library.

Creating Task Classes

This section describes how you create a task class from the Component Store or from the Browser.

To create a task class

- Open your library in the Browser
- Display the classes in your library using the View>>Down One Level menu option on the Browser menu bar
- Drag the template called "New Task" from the Component Store onto the Browser

or

- Select Class>>New>>Task from the Browser menu bar
- Name the new task
- Double-click on the task class to modify it

You modify a task class in the method editor. You can place in the $construct() method any code that you want to run when the task is opened. For the Startup_Task, the $construct() method is executed when you open your library. You can add any other custom properties and methods to the task, as well as any type of variable.

Opening Tasks

Apart from the startup task instance, which is opened automatically when you open your library, you can open a task using the *Open task instance* command or the $open() method. Any parameters you supply with the command are sent to the task's $construct() method.

```
Open task instance MyTask/TaskInstance2 (p1,p2,...)
; opens the task, assigns an instance name, and sends parameters
```

Alternatively you can open a task instance using the $open() method.

```
Do MyTask.$open('TaskInstance2',p1,p2,...) Returns iTaskRef
; does the same as above & returns a reference to the task instance
```

Current and Active Tasks

OMNIS keeps references to two different tasks, the *active task* and the *current task*, to keep track of the tasks that own the topmost instance or GUI object and the currently executing method. The active task is the task that owns the topmost open window, installed menu, or toolbar currently in use. The current task is the task that owns the currently executing method.

A *task context switch* occurs when OMNIS changes the current or active tasks. As OMNIS runs your library, the current and active tasks may point to different task instances depending on the user's actions.

The Active Task

The active task is affected by the user, and is typically the task containing the topmost open window. When an instance belonging to another task is selected, OMNIS performs a task context switch. As part of the context switch, messages are sent to both tasks. The active task gets sent a $deactivate() message, and the new active task is sent an $activate() message.

When the active task changes, you can use the $activate() and $deactivate() messages to perform other relevant actions such as hiding other windows, installing menus, and any other requirements your application has.

In order for OMNIS to perform an automatic task context switch when the user selects an instance belonging to another task, the task's $autoactivate property must be set to kTrue.

OMNIS can install and remove menus and toolbars automatically during task context switches. Menu and toolbar instances each have a $local property that you can set. When set to true, the menu or toolbar instance is made local to the task that owns it. When a task context switch occurs, local menus for the previously active task will be removed from the menu bar, and any local menus instances owned by the new active task will be installed. Toolbars behave similarly. If the tasks use different docking areas, OMNIS will not hide the docking areas, only the toolbars.

You can change the active task using the notation, rather than waiting for the user to initiate a task context switch. To do this, you can set the property $root.$activetask to a different task instance name to switch tasks.

The Current Task

The current task is under the control of OMNIS itself, and is the task instance which contains the currently executing method. When a custom attribute or event is sent to an instance, the current task is switched to the task which owns the instance, and when control returns from that attribute or event, the previous task is restored.

When the current task changes, messages are sent to both tasks. The current task is sent a $suspend() message, and the new current task gets a $resume() message. If the new current task is being instantiated for the first time, it gets a $construct() message rather than the $resume().

In order to avoid endless recursion a task does not get suspend or resume messages during the execution of a suspend or resume method.

Since $suspend() and $resume() are likely to be called frequently, it is important that the code for them should be kept as short and efficient as possible and should not:

– alter the user interface

– open or close an instance

– switch tasks

You can find out the name of the current task using the notation $ctask().$name, and the task that owns the instance by using InstanceName.$task().$name.

Closing Tasks

You can close a task instance using the *Close task* command or the $close() method. When you close a library all its task instances are closed, and when you quit OMNIS the default task is closed and all instances belonging to the default task are closed.

When you close a task, all instances belonging to that task are closed or destructed providing they can be closed. When instances are closed, a message is sent to the instance asking it to confirm whether or not it can be closed. If the instance returns a false message, OMNIS will not close that instance. For tasks, each instance belonging to the task is sent the message, and then the task itself is sent the message. If any of the instances belonging to the task cannot be closed, none of the instances nor the task instance are closed.

Task Variables

Task classes can contain both class and instance variables of any standard OMNIS data type. Tasks can also contain *task variables*, which are accessible to any instance owned by the task. As with other variables, you create task variables in the variable pane of the method editor.

When two or more types of variable use the same variable name, a reference to that variable may be ambiguous. In this situation, OMNIS uses the variable with the smallest scope automatically. All other variable scopes have precedence over task variables.

When a method in a code class is called from another class or instance using the *Do code method* command, the current task continues to point to the calling method. This allows methods in a code class to have access to the task variables from the calling method.

The Design Task

In order for task variables to be available to you for use in design mode, you must establish a connection between a class and the task whose variables you want to access. You do this by setting the *design task* for the class. The design task determines which task variables are available to the class: if no design task has been set, the method editor does not let you declare or see any task variables.

Setting the design task for a class doesn't guarantee that the task will be available in runtime when you open your class, nor will OMNIS automatically create an instance of the task. The design task is simply a way to give you access to a set of task variables while you create the classes in your library.

You can also access task variables without setting a design task by referring to the variable as $ctask.variablename. This assumes that the variable will always belong to a task and can therefore default to the current task.

If you attempt to access a task variable in an instance, and that variable is not available in the task, a runtime error of 'Unrecognized task variable' will be generated, and the variable will have a NULL value.

If you rename a task variable, any references to it are not renamed. Also if one with that name ceases to exist, references to it which were entered as VariableName are shown in design mode as $ctask.VariableName. Similarly, if some code containing a task variable is pasted into a different class, any task variables used by that code are not copied into the destination class.

Private Instances

Normally an instance is visible to other tasks and you can reference it using the notation from anywhere in your library. However you can override this default behavior by making an instance *private* to the task that owns it. You can do this by setting the instance's $isprivate property to kTrue.

When you make an instance private, you cannot destruct it, make references to it, or even see it unless you are within the task that owns it. A task can even be private to itself, so it can be closed only when it is the current task. If access to a private instance is required from outside of the task, an item reference can be set to the instance, and the item reference can be passed outside of the task. Once this has occurred, the item reference can be used to manipulate the instance.

The $root object has a series of object groups, one for each instance type, that are represented in the notation as $iwindows, $imenus, $itoolbars, $ireports, $itasks. Each of these object groups displays all public instances, as well as instances which are private to the current task. As the current task changes, the contents of these groups may change to reflect the private instances present in your library.

Private Libraries

Libraries can be private to a task, and both the library and its classes are visible only to that task.

The group of open libraries, $libs, contains a private library only when the task which owns that library is the active task. The Browser does not display classes from a private library. Standard entry points to the debugger such as shift-click on a menu line do not enter the debugger if the menu belongs to a private library.

As with private instances, if an item reference to any object within a private library is passed to an object outside the library, it is able to access the library using notation.

You can make a library private by setting its $isprivate property to true. This is normally done immediately after opening the library, but can be done at anytime as long as the task which owns the library is the active task. Libraries also have the $alwaysprivate property, which, if set, means they are always and immediately private to their startup task.

Private libraries have an additional property, $nodebug, which keeps the debugger from being entered for any reason when code from that library is executing, including errors, breakpoints, and the stop key. Code from a private library with $nodebug set does not appear in the stack menu or the trace log.

When a task is closed, it closes all its private libraries unless they cannot be closed. This can occur if, for example, the library has instances belonging to other tasks. If a private library cannot be closed, it will become non-private.

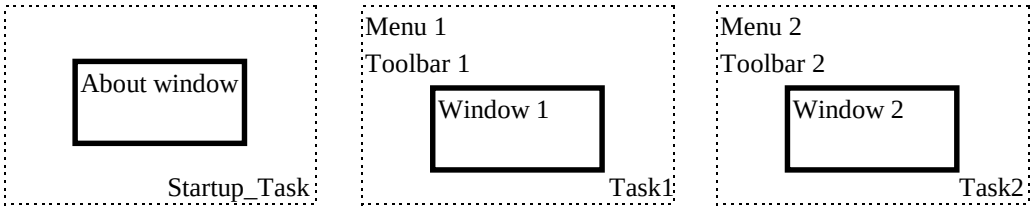
Multiple Tasks

When designing an application, you might want to partition your library by creating modules containing all of the windows, reports and methods of like functionality. Each module can have its own menus and toolbars. An example containing such modules might be an accounting package, with General Ledger, Accounts Payable and Accounts Receivable modules.

In a totally modal application, where the user switches between modules, it is easy to ensure that the user sees the correct menus and tools for the current module. In a modeless, multi-window environment, controlling this can sometimes be difficult. Tasks automate the process of creating modular applications by providing all the management of menus and tools for you.

Consider the following example in which a single library is running three tasks: the *Startup_Task* and two user tasks *Task1* and *Task2*. The startup task, which opens automatically when the library opens, contains an About window. The other two tasks each contain a window, a menu, and a toolbar. When the user selects a window from either Task1

or Task2, you may want OMNIS to display the correct tools and menus for that window automatically.



When the library opens, the startup task opens and displays the About window and then opens the other tasks, each of which opens its window and installs its menu and toolbar. The startup task can close itself once the About window is closed if it's no longer needed.

To open the two tasks, you should execute the following in the $construct() method of the startup task

```
Open window instance AboutWindow
Open task instance MyTaskClass1/Task1
Open task instance MyTaskClass2/Task2
Close task instance LibraryName  ;; close Startup_Task instance
```

Every task has a property $autoactivate, that allows the task to take control whenever the user tries to bring a window it owns to the front. If the property is set to false, the window won't come to the front. To activate each task automatically, you need to execute the following in the $construct() of each task

```
Do $ctask.$autoactivate.$assign(kTrue)
```

To ensure that your menus and toolbars show and hide appropriately as the tasks change, you need to set the $local property for each class. By making each menu and toolbar local to the task that owns it, OMNIS hides and shows them automatically as the task context changes.

In the $construct() for a task, you can install your menu and toolbar, and set their $local property. For example

```
; $construct() for task1...
Do $menus.MyMenuClass1.$open('Menu1') Returns iMenuRef
Do iMenuRef.$local.$assign(kTrue)
Do $toolbars.MyToolbarClass1.$open('Toolbar1') Returns iToolRef
Do iToolRef.$local.$assign(kTrue)
Do $windows.MyWindowClass1.$open('Window1') Returns iWinRef
```

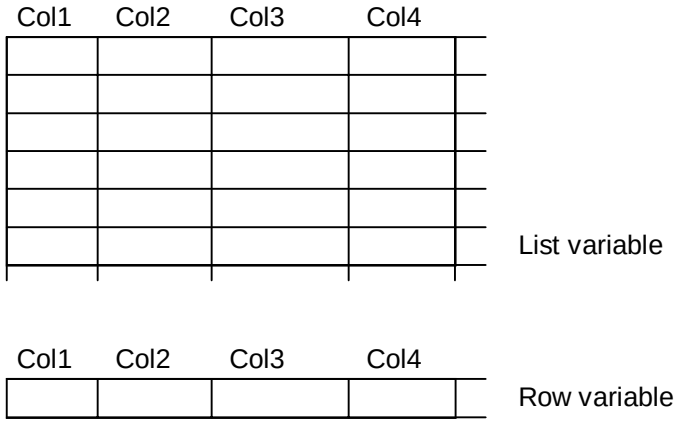
You can do the same for the other task.

```
; $construct() for task2...
Do $menus.MyMenuClass2.$open('Menu1') Returns iMenuRef
Do iMenuRef.$local.$assign(kTrue)
Do $toolbars.MyToolbarClass2.$open('Toolbar1') Returns iToolRef
Do iToolRef.$local.$assign(kTrue)
Do $windows.MyWindowClass2.$open('Window1') Returns iWinRef
```

This functionality will change menus and toolbars as you switch from one window to the other.

Chapter 6—List Programming

OMNIS has two structured data types; the list and the row. A *list* can hold multiple columns and rows of data each row having the same column structure, while a *row* is effectively a single-row list. You can create lists of strings, lists of records, or lists of lists. You can define a list from individual variables, or base a list on one of the OMNIS data classes, such as a schema, query, table, or file class. In the latter case, the list gets its column definitions from the columns defined in the data class. Each list can hold an unlimited number of lines with up to 400 columns.



OMNIS makes use of lists in many different kinds of programming tasks such as generating reports, handling sets of data from the server, and importing and exporting data. The list is the single most important data type in OMNIS programming.

In this chapter, rows are generally treated the same as lists, that is, you can use a row name in any command that takes a list name as a parameter. In addition, references to *SQL lists* in this chapter refer to lists based on either schema, query, or table classes, which are referred to collectively as *SQL classes*. Designing simple list and grid fields is described in the *Lists and Grids* in *Using OMNIS Studio*.

Declaring List or Row Variables

You can create various scopes of list and row variables, including task, class, instance, and local variables. You declare a list or row variable in the variable pane of the method editor. The following table summarizes the variable types and their visibility.

List or row type	When created?	Where visible?	When removed?
Task variable	on opening task	within the task and all its classes and instances that belong to the task	on closing task
Class variable	on opening the library	within the class and all its instances	on clearing class variables or closing library
Instance variable	on opening instance	within the instance only	on closing instance
Local variable	on running method	within the method only	when method terminates
Field in file class with list or row type	when defined in file class	within the whole library	when you update or delete the record, or Next or Previous command

To declare a list or row variable

- Right-click in the variables pane of the method editor
- Select **Insert New Variable** from the context menu
- Enter the variable name
- Click in the **Type** box and choose **List** or **Row** from the droplist

Defining List or Row Variables

To define a list or row variable you need to specify its columns. You can do this using OMNIS commands or the notation. You can define a list or row variable

– from variables

– from a schema, query, or table class

– from a file class

Lists from Variables

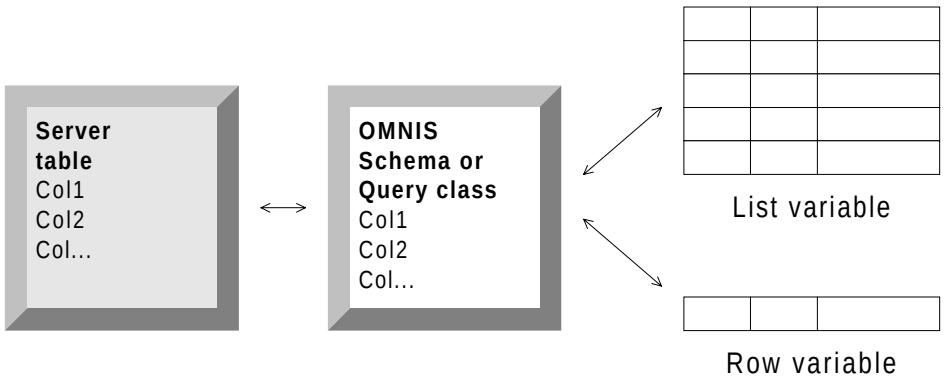
To define a list from a number of variables you use the *Define list* command. For example

```
; Declare class variable cvList1 of List type
; Declare class variable cvCol1 of Short integer type
; Declare class variable cvCol2 of Character type
; Declare class variable cvCol3 of type Date Time (Short date
  1980..2079)
Set current list {cvList1}
Define list {cvCol1,cvCol2,cvCol3}
```

This method will define the list cvList1 with the columns cvCol1, cvCol2, cvCol3. The data type of each field or variable defined in the list determines the data type of the corresponding column in the list.

Lists from Schema or Query Classes

You can define a list based on one of the SQL classes, that is, a schema, query, or table class, using the *Define list from SQL class* command or $definefromsqlclass() method. This binds the list or row variable to the schema or query class and consequently maps the list's columns to the server table.



To define a list or row variable from a schema, query, or table class you can use the *Define list from SQL class* command. For example

```
; Declare class variable cvList1 of List type
Set current list {cvList1}
Define list from SQL class {MySchema}   ;; or use a query or table
```

will define a list with all the columns in your schema class. You cannot specify certain columns only when using a schema. However, you can select certain columns by creating a query class containing specific columns and basing your list on the query class.

Specifically, when you define a list or row variable from a table class, it must have its **sqlclassname** property set to the associated schema or query class. You can do this either in the Property Manager or using the notation.

```
Do $clib.$tables.MyTable.$sqlclassname.$assign('MySchema') ;; or
Do $clib.$tables.MyTable.$sqlclassname.$assign('MyQuery')
```

When you create a list or row variable based on one of the SQL classes a table instance is created, so the list or row variable contains the standard properties and methods of a table instance. Specifically, if you create a variable based on a table class it contains any custom methods you have added to the table class; these can override the standard table instance methods. The following standard methods are available for lists based on a SQL class.

- **$select()**
  issues a select statement to the server
- **$fetch(n[,append]**)
  empties the list and fetches the next n rows from the server; for row variables, n is set to one and the fetched row always replaces any existing data; the append switch is for list variables and defaults to kFalse which means the list is cleared by default, otherwise if you pass the append switch as kTrue the fetched rows are added to the end of any existing data in the list variable
- **$insert**()
  inserts a row into the server database (row variables only)
- **$update(old_row**)
  updates a row in the server database (row variables only)
- **$delete**()
  deletes a row from the server database (row variables only)
- **$sqlerror()**
  reports the type, code and text for an error in processing one of the above methods

These methods offer a powerful mechanism for processing or inserting data on your server via your SQL list or row variable. For example, to fetch 30 rows into your list

```
; declare cvList1 of list type
Do cvList1.$definefromsqlclass(MySchema)
Do cvList1.$select() Returns myFlag     ;; sends a select
If myFlag = 0                           ;; checks for errors
   OK message  {SQL error [sys(131)]: [sys(132)]}
End If
Do MyList.$fetch(30) Returns myFlag         ;; fetches 30 rows
; to fetch another 10 rows and add them to your list
Do MyList.$fetch(10,kTrue) Returns myFlag
```

Lists from a File Class

To define a list from fields in a file class use

```
Set current list {cvList1}
Define list {field1,field2,field3,...}
```

or to include all the fields from a file class

```
Set current list {cvList1}
Define list {fileclassname}
```

The *Define list* command clears the list or row of any existing data. You can change the column names without affecting the data by using *Redefine list*: this is useful if, for instance, you want more friendly default column names for a graph. Note that you should use Filename.LISTVAR when you reference a file class list using the notation.

Building List Variables

You can build lists

– from SQL data

– from OMNIS data

Building a List from SQL Data

The SQL SELECT statement defines a select table, which is a set of rows on the server that you can read into OMNIS in three ways:

– *Fetch next row*
brings a row into CRB fields defined by a file class

– $fetch(n[,append]) table instance method
brings n rows into a list defined from a SQL class

– *Build list from select table*
transfers the select table to the current list as one block of data

To transfer rows:

```
Do mylist.$definefromsqlclass(SchemaRef,Lname,Town) ;; define list
Do MyList.$select() Returns myFlag  ;; make select table
Do MyList.$fetch(10) Returns myFlag ;; fetch 10 rows into list or
Do MyRow.$fetch() Returns myFlag    ;; fetch a row into a row var
```

To transfer the whole select table use the *Build list from select table* command:

```
; Declare class variable cvList of type List
Set current list cvList
Define list {fItems,LVAR1,LVAR5}
; defines the list with all fields from fItems, plus LVAR1 & LVAR5
Build list from select table  ;; builds list from SQL query
```

When loading large select tables into the list, you should avoid making the user wait for the whole set of rows to arrive before refreshing the screen to display the first fifty or so rows. You can retrieve the rows in batches using the $linemax property which limits the size of the list, pausing after each batch to redraw the list field.

Building a List from OMNIS Data

You can build a list from OMNIS data using the *Build list from file* command. It puts all the rows from an OMNIS data file into the current list. For example

```
Set current list LIST2
Define list (fCustomers)
Build list from file
```

You can also apply a search and sort with this command; see the *OMNIS Data Files* chapter.

For a list defined from file class or other variables you can add values either as variables or literals using *Add line to list*. This adds a line to the end of the list, or inserts a line after the specified line. For example

```
Add line to list {('Jones','Ipswich',fCountry)}   ;; adds at the end
Add line to list {10('Jones','Ipswich',fCountry)} ;; adds at line 10
```

Viewing the contents of a list variable

You can view the current contents of a list variable by Right-clicking on the variable name and selecting the first option in the context menu. You can do this wherever the variable name appears in OMNIS, including the method editor and Catalog.

List and Row functions

OMNIS provides functions for converting independent variables into a row, and for converting a series of row variables into a list.

The *list()* Function

The list() function accepts a set of row variables as parameters, and creates a list variable from them. The definition for the first row variable is used to define the list. If subsequent row variables have different definitions, OMNIS will convert the data types to match the first row.

```
Calculate myList as list(myRow1, myRow2, myRow3)
```

The *row()* Function

The row() function accepts a set of variables as parameters, and creates a row variable from them. The variable types are used to define the columns of the row.

```
Calculate myRow as row(myVar1, myVar2, myVar3)
```

Accessing List Columns and Rows

You can access data in a list by loading an entire row of data, or an individual cell into other variables. An entire row of information is loaded with the *Load from list* command. You can access individual cells using the *Load from list* command or the *lst()* function, or by referencing the list row and column as part of a calculation. Don't confuse *lst()* with the *list()* function discussed in the previous section.

The *Load from list* command takes an optional set of variables to load data into. In the case of a list defined from a File class or other variables, the load command will automatically place each column's data into the fields of the same name. To load a list defined from a SQL class, you include a list of variables as part of the *Load from list* command.

```
Load from list              ;; loads the row into a file class
Load from list (Var1, Var2) ;; loads the row into specified vars
```

You can use the *lst()* function as part of a calculation to extract a particular cell of information from a list.

```
Calculate MyVar as lst(MyList, rowNumber, ColumnName)
```

You can address cells directly by referring to them as ListName.ColumnName for the current row or ListName.RowNumber.ColumnName for a specified row. You can use RowName.ColumnName for a row variable. OMNIS also recognizes the syntax ListName('ColumnName',RowNumber). The column name must be in quotes, for example

Since ListName.ColumnName and ListName.RowNumber could be ambiguous, OMNIS assumes character values are column names. In the case of the row number being contained by a character variable, this should be indicated by adding '+0'.

```
Calculate MyNum as MyList.Amount ;; the current row
Calculate MyNum as MyList.5.Amount ;; row 5
Calculate MyNum as MyList('Amount',5) ;; Amount column, row 5
```

The two types of statement above are also used to assign a value to a list element.

```
Calculate MyList.5.Amount as 100 ;; sets Amount column, row 5 to 100
```

List Variable Notation

List variables have certain standard properties and methods that provide information about the list, such as how many rows or columns it has, or the number of the current line. List columns, rows, and cells have properties and methods of their own which are listed in the *OMNIS Help*.

List Properties and Methods

All types of list have the following properties. A list created from a SQL class has the standard properties and methods of a table instance, together with these list properties.

- **$linecount**
  returns the number of lines in the list; you can change this property or use *Set final line number* to truncate the list
- **$linemax**
  holds the maximum number of lines in the list; this is set to 10,000,000 by default but you can change it to restrict the list size
- **$line**
  holds the current line in the list; this changes when the user clicks on a list line, or when using a method such as $search()
- **$colcount**
  returns the number of columns in the list
- **$isfixed**
  true if the list has fixed length columns; changing $isfixed clears the data and the class for the list, but keeps the column definitions (note that a list defined using $define() has columns of any length). Fixed length columns improve performance in some cases, but cannot contain all data types
- **$class**
  returns the schema, query, or table class for the list, or is empty if it is not based on a SQL class

– **$cols**
group containing the columns in the list; you can use $add() to add a column, but $addbefore() and $addafter() do not work for $cols

For a row variable, $linecount, $linemax and $line are all set to 1 and cannot be changed.

Lists also have the following methods.

– **$define**()
without parameters this clears the list definition, otherwise $define(var1[, var2, var3]...) defines a list using variables or file class fields

– **$definefromsqlclass**()
$definefromsqlclass(notation for SQL class[,parm1,parm2]...) defines a list or row variable from a schema, query, or table class; the parameters are sent to the $construct() of the table instance

– **$copydefinition**()
$copydefinition(list or row variable[,parm1,parm2]...) clears the list and copies the definition but not the data from another list or row variable; if the list being copied from is derived from a SQL class, the parameters are passed to $construct() of the table instance

– **$clear**()
clears the data for the list, but keeps the list definition

– **$first**()
$first(selected only[, backwards]) sets the current row of the list to the first row or first selected row and returns a reference to that row

– **$next**()
$next(list row or row number[, selected only, backwards]) sets the current row of the list to the next row or next selected row and returns a reference to that row

– **$add**()
$add(column1 value[, column2 value]...) inserts a row at the end of the list

– **$addbefore**()
$addbefore(list row or row number,col1 value[, col2 value]...) inserts a row before the specified row

– **$addafter**()
$addafter(list row or row number,col1 value[, col2 value]...) adds a row after the specified row

– **$remove()**
$remove(list row or row number) deletes the specified row

- **$search**()
  $search(search calculation[, from start, only selected, select matches, deselect non matches]) searches the list; behaves the same as for the *Search list* command
- **$sort**()
  $sort(first sort variable or calculation, sort order[, second sort variable or calculation, sort order]...) sorts the list; you can specify up to 9 sort fields, including the sort order flag. The sort fields or calculations can use $ref.colname or list_name.colname to refer to a list column. The sort order flag defaults to kFalse (that is, the sort is normally ascending). For calculated sorts, the calculation is evaluated for line 1 of the list to determine the comparison type (Character, Number or Date).
- **$removeduplicates**()
  $removeduplicates(listname.column) removes all list lines with duplicate values in the column; you must sort the list before using this method
- **$merge**()
  $merge(list or row[, by name, only selected]) merges the two lists

Properties and Methods of a List Column

The columns of a list are contained in the $cols group. The $cols group contains the following properties:

- **$name**
  returns the simple name of the column
- **$dataname**
  returns the dataname of the list column; empty for a list defined from a SQL class
- **$objtype**
  returns the data type of the column; changing this clears the list data
- **$objsubtype**
  returns the data subtype of the column; changing this clears the list data
- **$objsublen**
  returns the length of character and national columns; changing this clears the list

List columns have the following methods:

- **$clear**()
  clears the data for the whole column; the column definition is left unchanged
- **$total**()
  calculates the total of all rows for the specified column
- **$average**()
  calculates the average value of all non-NULL rows for the specified column

- **$minimum**()
  calculates the minimum value of all rows for the specified column
- **$maximum**()
  calculates the maximum value of all rows for the specified column
- **$count**()
  number of rows for the specified column whose value is not NULL

Properties and Methods of a List Row

A list row has the following properties:

- **$group**
  returns the list containing the row
- **$selected**
  returns true if the row is selected

A list row has the following methods:

- **clear**()
  clears the value of all the columns in the row
- **$loadcols**()
  $loadcols(variable1[, variable2]...) loads the column values for the row into the specified variables
- **$assigncols**()
  $assigncols(column1 value[, column2 value]...) replaces the column values for the row with the specified values
- **$assignrow**()
  $assignrow(row, by name) assigns the column values from the specified row into the list row on a column by column basis

Properties of a List Cell

If a list cell is itself a list or row variable it has all properties of a list or row. List cells have the following properties.

- **$group**
  returns the list row containing the list cell
- **$ident**
  returns the column number for the list cell
- **$name**
  returns the column name for the list cell

– **$line**
returns the row number for the list cell; not necessarily the current line in the list

Manipulating Lists

You can change both the structure and data of a list variable using both commands and notation.

Dynamic List Redefinition

You can add, insert, remove, or move columns in list or row variables without losing the contents of the list or row. This functionality applies to all types of list and row variables including smart lists. In particular, the following notation no longer causes the contents of the list to be lost. In addition, $addbefore() and $addafter() have been implemented for list and row variables.

– List.$cols.$add(variable name)
adds a column to the right-hand end of the list using the specified variable as its definition

– List.$cols.$add(colname, type, subtype, length)
adds a column to the right-hand end of the list using the specified definition

– List.$cols.$remove(column name or number)
removes the specified column and moves any remaining columns to the left

– List.$cols.$addbefore(column name or number, variable name)
inserts a column to the left of the specified column using the specified variable as its definition, and moves any columns to the right as necessary

– List.$cols.$addbefore(column name or number, colname, type, subtype, length)
inserts a column to the left of the specified column using the specified definition, and moves any columns to the right as necessary

– List.$cols.$addafter(column name or number, variable name)
inserts a column to the right of the specified column using the specified variable as its definition, and moves any columns to the right as necessary

– List.$cols.$addafter(column name or number, colname, type, subtype, length)
inserts a column to the right of the specified column using the specified definition, and moves any columns to the right as necessary

– List.$cols.column name or number.$ident.$assign(new column number)
moves the column to a new position and moves other columns to the right or left as appropriate; in this case the $ident of a list column is its column number, therefore changing the ident moves the column to a different position

You cannot insert, remove, or move columns in a list defined from a SQL class, since you cannot redefine schema-, query-, or table-based lists. However you can use List.$cols.$add() to add extra columns to a SQL list.

Clearing List Data

You can use the command *Clear list* or ListName.$clear() to clear the data from a list. You can clear individual columns of a list with the ListName.ColumnName.$clear(), and individual rows with ListName.rowNumber.$clear().

Searching Lists

You can search a list using the *Search list* command or $search() method. With *Search list*, the search criteria are set up either as a search calculation or a search class, and a successful search sets the flag. The following method selects all lines matching the search.

```
Set current list MaiList
Set search as calculation {Country = 'USA'}
Search list (From start, Select matches (OR))
```

Using the $search() method this example would be

```
Do MaiList.$search(Country = 'USA') Returns myFlag
```

Selecting List Lines

When you display the data in a list variable in a list field on a window, by default you can select a single line only. However, you can allow multiple selected lines by setting the list or grid field's $multipleselect property. When the user highlights list lines with the mouse, the $selected property for those lines is set. If the field does not have $multipleselect set, the current, selected line is the highlighted one; if the $multipleselect property is set, all highlighted lines are selected, and the current line is the one with the focus.

Some of the commands that operate on a list variable use $selected to indicate their result. For example, *Search list (Select matches)* will set $selected for each line that matches the search criteria.

Each list variable has two select states, the *saved* and *current* selections. The current selection is the set of lines currently selected, whereas the saved selection is the previous set of lines that was selected before the current selection changed.

There are a number of commands that you can use to manipulate selected lines, save the current selection, and swap between the selected and saved states. These commands are described in the *OMNIS Studio Help.*

Merging Lists

You can copy lines from one list to another using the *Merge list* command or the $merge() method. Merging copies one whole list to another, or certain lines only if you include the *Use search* option. The following example copies the selected lines from LIST1 to LIST2 by checking each line's $selected property.

```
Set current list LIST2
Set search as calculation {$clist.$selected=1}
Merge list LIST1 (Use search)
```

$merge() provides slightly different capabilities in that it can align the results by column name as well as by position, whereas *Merge list* works by position only. The syntax is $merge(listName, byColumnName, useSearch). The example above could be written as:

```
Do List1.$search($selected=kTrue)
Do List1.$merge(List2, kFalse, kTrue)
```

Sorting Lists

You can specify up to nine levels of sorting using the *Sort list* command or $sort() method. To use *Sort list* you need to set up the sort fields first, and clear any existing sort levels since these are cumulative. $sort() clears existing sort fields automatically. For example

```
Set current list {MyList}
Clear sort fields
Set sort field Country
Set sort field Town
Set sort field Name
Sort list
Redraw lists
```

The $sort() method takes the sort variables or column names in order, each followed by a boolean indicating the sort direction. Using notation, the equivalent of the above example would be

```
; Country, Town, Name are columns in MyList
Do MyList.$sort(Country,kFalse,Town,kFalse,Name,kFalse)
Redraw lists
```

Removing Duplicate Values

List columns have the $removeduplicates() method which removes lines with duplicate values in the column. You must sort the list on the column before using this method.

```
Do MaiList.$sort(CustNum,kFalse)   ;; sorts list on CustNum column
Do MaiList.$cols.CustNum.$removeduplicates() return NumRemoved
```

Smart Lists

You can track changes made to a list by enabling its **$smartlist** property. A smart list saves any changes, such as deleting or inserting rows, in a parallel list called the history list. Smart lists can be filtered, a process which allows data not meeting a particular criteria to be made invisible to the user while being maintained in the history list.

A smart list variable therefore contains two lists:

– the *normal* list containing the list data, and

– the *history* list containing the change tracking and filtering information

If you store a smart list in an OMNIS data file or as a binary object is a SQL database, all the smart list information is stored automatically.

Enabling Smart List Behavior

To enable the smart list capability of any list variable you have to set its $smartlist property to kTrue.

```
Do ListName.$smartlist.$assign(kTrue)   ;; to enable it
```

Setting $smartlist to kTrue creates and initializes the history list. If it is already kTrue, then setting it again has no effect.

Setting $smartlist to kFalse discards the history list completely. The current normal list remains unchanged, so the current contents of the normal list are preserved, but all history and filtering information is lost.

If you define or redefine a list using any mechanism, or add columns to a list, its $smartlist property is set to kFalse automatically.

The History List

The history list has one row for each row in the normal list, together with a row for each row that has been deleted or filtered. The history list has the columns contained in the normal list as well as the following additional columns:

– **$status**
contains the row status, which is one of the constants kRowUnchanged, kRowDeleted, kRowUpdated, or kRowInserted, reflecting what has happened to the row. Only one status value applies, so a row that has been changed and then deleted will only show kDeleted. Note that kRowUpdated is true if the row has changed in anyway, even if the current values do not differ from the original column values.

– **$rowpresent**
true if the row is still present in the normal list, otherwise, the row is treated as if it has been deleted

– **$oldcontents**
a read only row variable containing the old contents of the row

– **$currentcontents**
a read only row variable containing the current contents of the row

– **$errorcode**
an integer value that lets you store information about the row; the standard table instance methods use this to store an error code

– **$errortext**
a text string that lets you store information about the row; the standard table instance methods use this to store an error text string

Properties of the History List

You can access the history list via the $history property, that is, LIST.$history where LIST is a smart list. $history has the properties:

– **$linecount**
read-only property that returns the number of rows in the history list

$history also supports the standard group methods $first() and $next() as well as $makelist(), but you cannot change the history list.

Properties of Rows in the History List

LIST.$history.N refers to the Nth row in the history list. You can use this notation to access the columns using the following properties:

– **$status**
the status of the row: not assignable

– **$rowpresent**
results in the row being removed from, or added to, the normal list: this is assignable, but there are several circumstances which cause OMNIS itself to change $rowpresent and override your changes (deleting a row, applying or rolling back a filter, etc.)

– **$rownumber**
the row number of the row in the normal list, or zero if $rowpresent is false; not assignable

– **$filterlevel**
the number of filters applied to the history list, up to 15: not assignable (see filtering below)

– **$oldcontents**
the old contents of the row in the normal list: not assignable

– **$currentcontents**
the current contents of the row in the normal list: not assignable

– **$errorcode**
the error code for the row; assignable and initially zero

– **$errortext**
the error text for the row; assignable and initially empty

The above row properties are also properties of the list rows in the normal list, and provide a means of going directly to the history data for a line. In this case, $rowpresent is always kTrue, but can be set to kFalse.

Tracking the Changes

Change tracking occurs automatically as soon as you enable the $smartlist property for a list. From this time, OMNIS automatically updates the status of each row in the history list whenever it inserts, deletes, or makes the first update to the row. Note that change tracking only remembers a single change since the history list was created. Hence:

– Updating a row of status kRowUnchanged changes it to kRowUpdated; updating a row with any other status leaves the status unchanged

– Inserting a row always sets the status to kRowInserted and makes the row present in the normal list

– Deleting a row always sets the status to kRowdeleted and makes the row not present in the normal list; the row is still present in the history list (and can be made present in the normal list) until a $savelistdeletes operation is performed

Change Tracking Methods

The history list has several standard methods that let you undo or accept changes to the list data. After using any of these methods, the list is still a smart list.

You can use the following methods for accepting changes:

– **$savelistdeletes**()
removes rows with status kRowDeleted from the history list, and also from the normal list if $rowpresent is kTrue

– **$savelistinserts**()
changes the status of all rows with kRowInserted to kRowUnchanged, and sets the old contents of those rows to the current contents. It does not change $rowpresent

– $**savelistupdates**()
changes the status of all rows with kRowUpdated to kRowUnchanged and, for all rows, sets the old contents to the current contents; this does not change $rowpresent

– $**savelistwork**()
quick and easy way to execute $savelistdeletes(), $savelistinserts() and $savelistupdates()

And these are for undoing changes made to the list data:

– $**revertlistdeletes**()
changes the status of all kRowDeleted rows to kRowUnchanged or kRowUpdated (depending on whether the contents have been changed); for these rows $rowpresent is set to true

– $**revertlistinserts**()
removes any inserted rows from both the normal list and the history list

– $**revertlistupdates**()
changes the status of all kRowUpdated rows to kRowUnchanged and, for all rows, the current contents are set to the old contents; this does not change $rowpresent

– **$revertlistwork**()
quick way to execute $revertlistdeletes(), $revertlistinserts() and $revertlistupdates()

The history list also has a default method that lets you set the row present property based on the value of the status.

– **$includelines(**status**)**
includes rows of a given status, represented by the sum of the status values of the rows to be included. Thus 0 means no rows, kRowUnchanged + kRowDeleted means unchanged and deleted rows, and kRowAll means all rows, irrespective of status. This is a one-off action and does not, for example, mean that rows deleted later will remain flagged as present

Filtering

Filtering works only for smart lists. You apply a filter by using the $filter() method, for example

```
Do ListName.$filter(COL1 = ‘10’) Returns Count
```

$filter() takes one argument, which is a search calculation similar to one used for $search(). It returns the number of rows rejected from the list by the filter.

Filtering uses the row present indicator of the history list to filter out rows. In other words, after applying a filter, OMNIS has updated $rowpresent to kTrue for each row matching the search criterion and kFalse for the others. Filtering applies only to the rows in the normal list, that is, rows where $rowpresent is kTrue, with the result that repeated filtering can be used to further restrict the lines in the list.

Filter Level

Each history row contains a filter level, initially zero. When you apply the first filter, OMNIS sets the filter level of all rows excluded by the filter to one; that is, for each row in the normal list, for which $rowpresent becomes kFalse, $filterlevel becomes one. Similarly for the nth filter applied, OMNIS sets $filterlevel for the newly excluded rows to n. You can apply up to 15 filter levels.

Whenever a row is made present, for whatever reason, the filter level is set back to zero, and whenever the row is made not present, for any reason other than applying a filter, the filter level is also set back to zero.

Undoing a Filter

You can restore filtered rows to the normal list using the $unfilter() method, for example:

```
Do ListName.$unfilter() Returns Count
```

When called with no parameters, $unfilter() removes the latest filter applied. Otherwise, $unfilter removes filters back to the level indicated by the parameter. Thus $unfilter(0) removes all filters, $unfilter(1) removes all but the first, and so on.

Reapplying a Filter

You can reapply all the filters which have already been applied, in the same order, to all lines present in the normal list using the $refilter() method. For example

```
Do ListName.$refilter() Returns Count
```

The Filters Group

A list has a read-only group called $filters which lets you navigate through a list of the filters that have been applied. For example

```
ListName.$filters.N
```

identifies the Nth filter currently applied to the list, that is, the filter which filtered out rows at filter level N. Each member of the $filters group has a single property, $search calculation, which is the text for the search calculation passed to $filter() when applying the filter.

Committing Changes to the Server

The current state of the normal list can be committed to the corresponding server table, assuming the list was defined from a SQL class, using the following smart list methods

– **$doinserts**()
inserts any rows in the list with the row status kRowInserted

– **$dodeletes()**
deletes any rows in the list with the row status kRowDeleted

– **$doupdates()**
  updates any rows in the list with the row status kRowUpdated

– **$dowork()**
  executes the above methods one after the other, in the order delete, update, insert

List Commands and Smart Lists

Any command or notation which defines a list sets $smartlist to false, so that any history information is lost. You can use the following list commands and notation with smart lists but with particular effects.

– *Search list* and equivalent notation selects only lines in the normal list.

– *Sort list* and equivalent notation, includes all rows, even those with $rowpresent set to false, so that if those lines become present in the normal list they will be included in the correct position.

– When using *Merge list* or equivalent notation, if the source list is a smart list only its normal list is merged, not the history information. If the destination list is a smart list the merged lines are treated as insertions and have the status kRowInserted.

– When using *Set final line number*, if lines are added they are treated as insertions and have the status kRowInserted, and if lines are removed they are treated as deletions and are kRowDeleted.

– Using a *Build list...* command gives all lines the status kRowInserted. This performance overhead can be avoided by not setting $smartlist until after the list is built.

Chapter 7—Window Programming

This chapter describes some of the more advanced properties of window classes, and the different OMNIS window components you can use, including

- *Container field* types
  including tab panes, tab strips, page panes, subwindows, and complex grids
- Advanced *list* and *grid field* types
  including string and data grids, headed list boxes, icon arrays, and tree lists
- *Modify Report* and *Screen Report* fields
  to let users modify report classes and print reports to a window
- *Field styles*
  to implement styles throughout your application and across platforms
- *Format strings* and *Input masks*
  to format data input and display
- *Drag and Drop*
  for dynamically exchanging data and objects in your application

The *Window Classes* chapter, in the *Using OMNIS Studio* manual, tells you how to create window classes and describes the simpler fields and objects you can use to design windows. It also describes how you can create SQL and OMNIS data entry forms using wizards.

Container Fields

Some of the complex field types are described as *container* fields. A container field is simply a window field that contains other fields. These include tab and page panes, complex grids, group boxes, scroll boxes, and subwindows. All container fields except subwindows have the $objs and $bobjs object groups containing the fields and background objects within the container field. Therefore in the notation you access the objects within a container field via these object groups. For example the notation for a field called MyField inside a paged pane is

```
$windows.WindowName.$objs.PagePaneField.$objs.MyField
```

Every field within a container field has the $container property which returns the name of the container field the object belongs to.

You can nest container fields four levels deep. Beyond this level, the most deeply nested field is not set up when the window is opened and becomes a display field showing an error message.

Tab Panes, Page Panes, and Tab Strips

Tab panes and Page panes are types of window field that contain a number of panes on which you can place other fields. When the user clicks on a tab or other field its associated pane can be brought to the front. This type of container field is useful for Options or Preference-style dialogs in which you need to group a series of fields or options into logical or functional areas. Standard field properties control the overall size, position, and border style of a pane field, whereas each tab or pane has particular properties.

You can access the contained fields and background objects in a tab pane or page pane field via the $objs and $bobjs groups for the container field. To set the properties of individual tabs or panes, you must first set the **currenttab** or **currentpage** property as appropriate.

Tab Panes

Tab pane fields provide separate panes in design mode, on each of which you can place any number of fields and background objects. You can switch panes in design and runtime modes by clicking on the tab belonging to the pane. You set the position, number and style of the tabs, and whether the tabs have icons in the properties for the tab pane.

To create a Tab Pane field

- Drag a Tab Pane field from the Component Store onto your window
- Set **tabcount** and the other appearance properties
- Click on each tab and add the fields and background objects as required

Tab pane fields have the following properties

- **taborient** and **tabstyle**
  the position and style of the tabs; either at the top or bottom, with square, rounded, or triangular shaped panes
- **tabcount** and **currenttab**
  number of tabs or panes, and the currently selected one
- **imagenoroom**
  when insufficient room shows just picture and not text for each tab
- **showimages**
  shows icons for tabs in the field; specify icons under pane properties
- **showfocus**
  shows the focus for the selected tab
- **multirow**
  if true forces the tabs to stack rather than scroll when the field has many tabs
- **forecolor, backcolor,** and **backpattern**
  sets the color and pattern of the area behind the tabs, not the tab panes
- **selectedtabcolor**
  the color of the selected tab; defaults to kColor3DFace
- **tabcolor**
  the color of non-selected tabs; defaults to kColor3Dface

An individual pane has the following properties

- **tabcaption**
  text or label for the tab
- **iconid**
  id of the icon from an icon data file; you cannot use icons larger than 48x48 pixels for tabs (enable **showimages** to show icons)
- **tabtooltip**
  tooltip for the tab; you must enable the OMNIS preference **showwindowtips** to show object tooltips

Programming Tab Panes

When the user clicks on a tab at runtime, the field receives the event evTabSelected, with the parameter pTabNumber holding the number of the tab clicked. $currenttab changes to the current tab (and pane). Discarding the event will prevent the current tab from changing.

Tab pane fields can have a $control() method to control events for each of the contained fields.

To access individual tabs or panes using the notation, you must first set $currenttab. For example, to change the text on the second tab use

```
Do $cinst.$objs.TabPane.$currenttab.$assign(2)
Do $cinst.$objs.TabPane.$tabcaption.$assign('New tab text)
```

Page Panes

Page panes are similar to tab panes except that they have no tabs: you can switch the current pane or page using a tab strip, a set of radio buttons, a pushbutton, or some other field. In design and runtime mode you can set the number of panes in **pagecount**, and change the current page using **currentpage.**

To create a Page Pane field

- Drag a Page Pane field from the Component Store onto your window
- Set **pagecount** to the number of pages
- Add the fields and background objects to each page, changing the current page by setting **currentpage**

Programming Page Panes

You can change panes in a method that sets **currentpage** as required. Using a tab strip you can set the page to the selected tab.

```
; $event() method for the tab strip field
On evTabSelected
   Do $cwind.$objs.PagePaneId.$currentpage.$assign([pTabNumber])
```

Alternatively, you can design your own Next and Back buttons that cycle through the pages, similar to a wizard. For example

```
; $construct() method for the window
; declare variable cCurPage initial value 1
; declare variable PageRef of type Item reference
Set reference PageRef to $cwind.$objs.PagePaneId

; $event() method for Next button
On evClick
   Calculate cCurPage as PageRef.$currentpage + 1
   If cCurPage > PageRef.$pagecount
      Calculate cCurPage as 1  ;; if last pane, go to first
   End If
   Do PageRef.$currentpage.$assign(cCurPage)
   Quit event handler (Discard event)
```

Tab Strips

Tab strips contain a set of tabs only, they do not have pages or panes. The tab strip offers similar functionality to a set of radio buttons in that only one tab can be selected at a time. You could use a tab strip in conjunction with the page pane field to hide and show a series of fields on your window.

To create a Tab Strip field

- Drag a Tab Strip field from the Component Store onto your window
- Set the **tabs** property to set the text and number of tabs

You enter the text and number of tabs for the field in the **tabs** property. Enter a text label for each tab separated by commas. For example, the text Tim,Sue,Bill will enable three tabs with the specified text.

Tab strip fields have the additional appearance properties

– **backcolor**
sets the color of the area behind the tabs; turn off **ditherbackground** for a solid color

– **tabcolor** and **selectedtabcolor**
the color of the tabs, and the selected tab

– **tabtextcolor** and **selectedtabtextcolor**
the color of the text on the tabs, and the selected tab text color

– **showedge**
if true shows the edge of the tab strip

– **ditherbackground**
if true shows dithered background for the tab strip

– **overlap**
the overlap for the tabs in pixels

– **tableftmargin**
the indent for the left tab in pixels

Programming Tab Strips

Tab strips receive evTabSelected which you can handle in the same way as tab panes.

To add a new tab using the notation you have to assign to the $tabs property. For example, to add a third tab called Bill use

```
Do $cinst.$objs.TabStrip.$tabs.$assign('Fred,Sarah,Bill')
```

String and Data Grids

String and Data grids are both types of window field that display data from a list variable in an enterable table format. String grids display character-based data, whereas Data grids can display any type of data. You can scroll these grid types horizontally and vertically, and you can make the first column and/or first row non-scrolling headers if required. If you tab out of the last column in the last row in a data or string grid, a new row can be added to the grid.

String Grids

You can use string grids to display character-based data from a list. The row height and column width are set at design time, but columns can also be sized at runtime if the first row is fixed. String grids have the following properties

– **dataname**
the source list variable

– **defaultheight** and **defaultwidth**
the default row height and column width in pixels

– **designcols** and **designrows**
number of columns and rows displayed in design mode

– **fixedrow** and **fixedcol**
sets top row or first column as a fixed header or column

– **extendable**
if true a new row is added when you tab out of the last column of the last row

A string grid instance has the read-only properties

– **gridrows** and **gridcols**
the number of rows and columns

– **gridhcell** and **gridvcell**
the current cell column and row number

To create a string grid

- Drag a String Grid field from the Component Store onto your window
- Set the **dataname** property to the source list variable
- Set the number of list rows and columns in **designrows** and **designcols**
- Add the list-building method behind the grid field

You can make the first column and row fixed and non-scrolling by setting **fixedrow** and **fixedcol**. This sets the first row and column of your list data as the column and row headers. In addition, setting **fixedrow** lets you size the columns both at design and runtime by dragging in the column header. Dragging individual columns overrides any values set in **defaultwidth**. If you want to have variable column widths at design time, but not have a fixed row, set **fixedrow** back to false after sizing the columns. If you change **defaultwidth** after manually sizing the columns, a message asks whether you wish to keep the non-default widths.

Data Grids

Data grids are very similar to string grids with regards to their appearance, but with some extra features.

– data can be of many types, even pictures

– the row height adjusts to fit the data

– column header names are added as a property

To create a data grid

- Drag a Data Grid field from the Component Store onto your window
- Set **dataname** to the source list variable
- Set **autosize** if you want the row height to adjust to fit the data
- Set the number of list rows and columns in **designrows** and **designcols**
- Add the list-building method behind the grid field

You use **fixedrow** to adjust the column widths at design time. You can enter the column headings in the **columnnames** property as a comma-separated string.

With the **autosize** property on, character-based columns will size to a maximum of 5 lines deep; for larger amounts of data cells will scroll. Columns containing pictures will size to fit the picture. Different data types are displayed in different ways in a data grid: Boolean data types become droplists with true/false options, and lists are shown as droplists. Character, number, date and all other types map to edit fields.

Programming Data and String Grids

String grid cells are normally enterable except those in a fixed row or column. The grid receives specific events when the user clicks in the grid; it does not receive evClick and only receives evBefore and evAfter when entering or leaving the field. When the user clicks in a data cell, two events are sent, as follows

– evCellChanging
  returns pHorzCell, pVertCell, and pCellData event parameters

– evCellChanged
  returns pHorzCell, and pVertCell event parameters

Note these events are not available for complex grids. The parameters pHorzCell, pVertCell are the column and row numbers and pCellData is a character variable holding the contents of the updated cell. You can use the evCellChanging event to validate cell data entered by the user. If you discard the event the data is not changed.

```
; $event() method for the grid field
On evCellChanging
   If pCellData = ''
      OK message {You must enter a value}
      Quit event handler (Discard event)
   End If
```

If the user tabs out of the last column of the last row and the **extendable** is set, the evExtend event is sent with the parameter pRow. You can use this to set up the new row with default data or stop the grid extending, as follows

```
; $event() method for the grid field
On evExtend
   If $cobj.$gridrows > 20
      OK message {This grid cannot have any more lines}
      Quit event handler (Discard event)
   Else
      Calculate pRow.Column1 as DefaultVal
   End If
```

Setting column widths

You can set the column widths for Headed lists, String and Data grids using the runtime only property $columnwidths. This property returns a comma separated list of column widths in pixels. When you use the $assign() method to assign to this property, you must put the comma separated list in quotes. For example

```
; Item reference HeadedListRef set to headed list instance
Do HeadedListRef.$columnwidths returns HeadedCols
; returns something like '30,40,55'
Do HeadedListRef.$columnwidths.$assign('20,30,40')
; assigns the column widths to the headed list instance
```

Scrolling Tips for String and Data Grids

The following properties control string and data grid scrolling.

– $vscrolltips
  if true, enables vertical scrolltips showing the current row number when scrolling; the scrolltip contains the value of column 1 of the current row while scrolling

– $hscrolltips
  if true, enables horizontal scrolltips showing the current column number when scrolling; the scrolltip contains the value of row 1 of the current column while scrolling

– $cellbordercolor
  the grid cell borders

– $gridendcolor
  the color of empty grid where no data appears at the end of the grid

– $gridhcell and $gridvcell
  return the current cell using (row,column) coordinates; the first row of the grid is row 1, the first column is column 1

You can replace the default scrolltips for string and data grids by intercepting the evScrollTip event, and providing your own scrolltip string, evSrcollTip has three event parameters:

– pIsVertScroll
  if true, the current scrolltip is on the vertical scroll bar, otherwise it is on the horizontal scroll bar

– pScrollPos
  the list row number for vertical scrolling, otherwise the column number for horizontal scrolling

– pScrollTip
  the scrolltip text, if you do not assign a value the default scrolltip is used

You can also use the *Quit event handler* command with the discard event option if you do not want OMNIS to display a scrolltip.

Headed List Boxes

A *headed list box* is a type of window field that displays data from a list variable in a table format. You can add button style headers to each column of the list on which the user can click to sort the data. You can also make the columns sizeable, and individual cells can have different colors, patterns, and text styles.

In addition to the general list box properties, such as **multipleselect**, the headed list box has the following properties

- **dataname**
  the name of the list variable
- **calculation**
  the calculation to format the columns for the list
- **maxeditchars**
  the maximum size of the edit field for editing a column, or 0 if columns cannot be edited
- **enableheader**
  if true the column headings act like buttons
- **canresizeheader**
  if true the columns can be sized at runtime
- **boldheader**
  if true the headings are bold
- **showcolumnlines**
  if true the list draws lines between the columns at runtime
- **designcols**
  the number of columns, maximum of 30 columns
- **columnnames**
  a comma-separated list of heading text for the columns
- **align**
  allows you to determine column alignment
- **columnalignmode**
  provides runtime alignment control
- **headerfillcolor**
  the color of the header; defaults to kColor3DFace
- **headertextcolor**
  the color of the text in the header; defaults to kColorDefault, that is, the text color of the list

– **colcount**
  a runtime only property, which is the number of columns in the headed list.

– **columnwidths**
  a runtime only property, allowing the column width to be set. See the previous section on data and string grids..

Text Alignment

The $align text property lets you set the alignment of all the columns in the list. At runtime, you can override the alignment for individual columns using the method

– $setcolumnalign(columnNumber, alignment)
  sets the alignment to kLeftJst, kRightJst, or kCenterJst, and returns kTrue for success

and you can return the current alignment for a column using

– $getcolumnalign(columnNumber)
  returns the alignment of the specified column

Note that headed list boxes do not support the style() function with type parameters kEscLTab, kEscCTab and kEscRTab.

The property $columnalignmode provides additional runtime control over $setcolumnalign(), and can have the following values: kAlignModeHeading, kAlignModeBody, kAlignModeAll and kAlignModeNone. These determine whether the heading, body, both or neither are affected by calls to $setcolumnalign(). Note that the call to $setcolumnalign() always stores the new alignment value in the list; $columnalignmode determines if the stored value is used. When the stored value is not used, $align determines the alignment.

To create a Headed List Box field

- Drag a Headed List Box field from the Component Store onto your window
- Enter the **dataname, designcols** and **columnnames** properties
- Set the column widths by dragging; shift-drag resizes the column to the right of the mouse pointer so you can use this for the last column
- Set **maxeditchars** if the columns are to be editable
- If necessary, enter the formatting expression to **calculation**

You do not need a calculation if the columns in the headed list are an exact mapping of the columns in the data list. If not, use the calculation to format the columns in the headed list box. They must contain column names from your list and special column delimiters. You can use the *con()* function to format the calculation, and insert column delimiters using *chr(9)*, the tab character. For example, to format three columns the calculation could be

```
con(Col1,chr(9),Col2,chr(9),Col3)
```

You can also use the *style()* function to change the style and color of specific columns. For example, to give Col1 a blue spot icon, make Col2 red and right-justified, and Col3 italic you would enter the following calculation

```
con(style(kEscBmp,1756),Col1,chr(9),
     style(kEscColor,kRed),Col2,chr(9),
        style(kEscStyle,kItalic),Col3)
```

Programming Headed List Boxes

The $event() method for headed list boxes receives specific event messages. In addition to the general entry field events evClick, evDoubleClick, evAfter, evBefore, and the drag and drop events, there are specific events to report clicks on the column headers and when the list data is edited.

Column Headers

When the header is enabled by setting **enableheader**, user clicks on the header buttons generate the evHeaderClick event with the column number held in pColumnNumber. You can use this event to sort the column clicked on. The following method sorts the column, reversing the existing order

```
; $event() method for headed list box
; declare class variable SortOrder of Boolean type init value 0
On evHeaderClick
   Calculate lColumnName as cList.$cols.[pColumnNumber].$name
   Do cList.$sort(cList.[lColumnName],SortOrder)
   Calculate SortOrder as not(SortOrder)  ;; reverses the order
   Redraw {HeadedListBox}
```

Editing the List

When text editing is enabled by **maxeditchars**, a headed list box receives three events in a specific order, together with parameters containing the list line, column number and new text entered.

– evHeadedListEditStarting
with parameters pLineNumber, pColumnNumber, is sent on the first click in the selected cell which puts the cell into edit mode; discarding the event prevents editing

– evHeadedListEditFinishing
with parameters pLineNumber, pColumnNumber, pNewText, is sent if the user enters a new value by hitting return or clicking away from the edit field; discarding the event leaves the field in edit mode, for example if pNewText is invalid. Note that you must store the new valid text in the list at this point: OMNIS cannot do this since the data is a calculated expression

– evHeadedListEditFinished
with parameters pLineNumber, pColumnNumber, is sent when the edit is completed

You could use the following event handlers for these events

```
; $event() method for the headed list box
On evHeadedListEditStarting
   If pColumnNumber=2      ;; bar editing in this column
      OK message  (Icon,Sound bell) {Cannot edit this column}
      Quit event handler (Discard event)
   End If
On evHeadedListEditFinishing
   If pNewText=''
      Quit event handler (Discard event)
   Else
      Calculate cList.pLineNumber.pColumnNumber as pNewText
   End If
On evHeadedListEditFinished
   ; do anything necessary here
```

Default Methods

Headed list boxes contain the following default methods

– $edittext(column number)
puts the field into text edit mode if: there is a currently selected line, the field is the current field, editing is enabled by **maxeditchars**

– $getedittext(line number,column number)
called by the headed list to get the data to edit for a column, before displaying the edit field; this gets the column data and strips text escapes inserted by *style()*. You can override this attribute if the default processing is not what you want.

Complex Grids

A *complex grid* is a type of window field that can display multiple rows and columns of data taken from a list variable. To create a complex grid you place other fields, including standard entry fields, droplists, and checkboxes, in the row and header sections of the grid field. Complex grid fields are container fields having their own $objs and $bobjs groups containing the foreground and background objects inside the grid field.

Every object in a complex grid has the **gridsection** property which tells you the section the object is in, and **gridcolumn** which tells you its column. The fixed left-most column is column zero: the other columns are numbered from one and are separated by the dividers. Every field in the header is in column zero. The **top** and **left** properties of an object is

relative to the top left-hand corner of its grid section. The **dividers** group contains the dividers for the grid field.

The complex grid supports the standard properties $firstsel, $lastsel, $firstvis and $lastvis. $firstsel and $lastsel only apply when the grid is not enterable.

Every field or object contained in a complex grid has the $container property which returns, in this case, the name of the grid field the object belongs to.

To insert a field in a complex grid from a method you use the notation

```
Do MyGrid.$objs.$add(section,column,type,top,left,height,width)
   Returns NewFieldRef
```

Events for Complex Grids

Each contained field receives its normal event messages such as evClick, evBefore, evAfter, and the field event handler can pass these events to the $control() method contained in the complex grid.

Complex grids receive the events evRowChanged and evExtend which you can handle in the $event() method for the grid field. The evRowChanged event is sent whenever the user clicks in a different row and when the window is opened. The evExtend event is sent whenever a row is added to a grid with the **extendable** property set. These events return the pRow event parameter which holds a reference to the row changed or the new row. Thus pRow.$line gives you the row number and pRow.ListColName returns the value of the cell. Note the events evCellChanging and evCellChanged are available for string and data grids only, not complex grids.

Grid Field Exceptions

Generally, the properties of a complex grid apply to the whole grid or to a single row or column. However, you can set the properties of a single cell in the window instance by setting an *exception* for the grid cell. To do this you use the notation

```
Do MyGrid.$objs.fieldname.row.property.$assign(value)
```

For example, if you wanted to show cells in Grid1 column cBal in red if the value is negative, you could use the following code which runs when the user tabs out of the cell

```
On evAfter
   Calculate row as pRow.$line
   If cList.[row].cBal < 0
      Do $cinst.$objs.Grid.$objs.fBal.[row].$backcolor.$assign(kRed)
      Redraw {Grid}
```

You could use this event handler for the cBal field in the grid, but the interior fields could pass the events up to the $control() method in the complex grid field.

```
; $control() method for the grid field
On evAfter
   Calculate row as pRow.$line
   If $cobj.$name = 'fBal' & cList.[row].cBal < 0
      Do $cobj.[row].$backcolor.$assign(kRed)
      Redraw {Grid}
```

You can clear exceptions using the $clearexceptions() method which acts on a cell, row, column or the whole grid, as follows

```
; clear all exceptions
Do $cinst.$objs.GridName.$clearexceptions()

; clear exceptions for a row
Do $cinst.$objs.GridName.$clearexceptions(RowNum)

; clear exceptions for a column
Do $cinst.$objs.GridName.$objs.FieldName.$clearexceptions()

; clear exceptions for a cell
Do $cinst.$objs.GridName.$objs.FieldName.RowNum.$clearexceptions()
```

Subwindows

A *subwindow field* is a type of window field that contains another window class. You can put any window class into a subwindow field; in this context, the window class inside a subwindow field is referred to as the subwindow class, and the window containing the subwindow field is called the parent window. The subwindow class can contain any number of fields or window objects, such as a group of radio buttons, a set of standard pushbuttons, or it might contain a single field only, such as a complex grid field. The window class can contain its own methods which in effect become the methods for the subwindow field. Subwindow fields let you design sets of window objects and their associated methods, store them as separate window classes, and reuse them on different windows as subwindow fields with all their variables and methods encapsulated.

Creating a Subwindow

In design mode the subwindow field appears as a single object, so you cannot access the fields contained in the subwindow class. The title bar and size borders of the subwindow class are ignored. Fields inside the subwindow can have their edgefloat properties set so that they resize with the parent window. In runtime the fields contained in the subwindow field appear on the open window as standard fields and are part of the normal tabbing order. A subwindow field is a container field, but it does not contain the $objs and $bobjs groups like other container fields; its objects are treated as part of the parent window.

To create a subwindow field

- Drag a Subwindow field from the Component Store onto your window

or

- Click on the Subwindow icon in the Component Store and draw the field in your window
- Open the Property Manager or press F6/Cmnd-6 to bring it to the top
- Select the **classname** property and enter the name of a window class or select one from the droplist; normally you should leave the **dataname** property empty

When you place the subwindow field it will resize to accommodate the subwindow class. You can edit the subwindow class at any time by right-clicking on the subwindow field and selecting Subwindow Class from the context menu.

If you enable the **nobackground** property, under the Appearance tab, the background of the subwindow field becomes the same color and pattern as the parent window. Normally, the text style of individual fields inside your subwindow class is retained. However, you can force these fields to use the text style of the subwindow field if you enable their **subwindowstyle** property in the original window class.

If the window class inside the subwindow field has only one field you can override its dataname using the **dataname** property for the subwindow field. For example, your subwindow class may contain a single complex grid field that takes its data from a particular list variable. However you can change the list assigned to the complex grid by setting the subwindow field's dataname property to the name of another list. You could do this in the $construct() method of the subwindow field.

Opening the Parent Window

Opening a window containing a subwindow field or any number of subwindows creates an instance of each window, which belong to the same task as the parent window instance and contains all the variables of its class. OMNIS calls the $construct() methods of all the subwindow classes first in tabbing order, then the $construct() method of the parent window instance. The reverse happens on closing the parent window, with the subwindows being destructed after the parent window instance. It is important not to include an *Enter data* command in a subwindow $construct() method as this affects the opening of the parent window.

You can send parameters to the subwindow's $construct() method by including a list of parameters in the **parameters** property when you create or modify the subwindow field.

Programming Subwindows

A subwindow instance inherits the properties and methods of its class and superclasses, as well as having the normal properties of a window field. They are available only within the instance and not to the parent window since the subwindow is private to itself. Within a subwindow instance, $cobj refers to the current internal subwindow field rather than the container field. From an internal field method, you can access subwindow field properties using $cobj.$abc, whereas subwindow class methods such as $control() must be accessed using $cinst.$control().

Subwindow Events

To the parent window, the subwindow is a single field and never has the focus, but does receive some events. Field events in the subwindow are sent only to the subwindow $event() method and not to the parent window. If the **nobackground** property is not set, click and scroll events on the subwindow are sent to:

1. the subwindow $event() method
2. the subwindow field $event() in the parent window
3. the parent window $event() method
4. the task $control() method

Mouse enter and mouse leave events are sent only to the subwindow $event(), while mouse up, mouse down on the subwindow are passed to the parent window.

Drag and drop

It is possible to drag and drop data to and from the fields inside a subwindow as though they were in the parent window, and also to and from the subwindow field itself provided **nobackground** is off. When dropping data from the subwindow pDragValue will usually be set up, or alternatively you could use a custom $contents to hold the drag value (since a subwindow has no default $contents property).

You cannot use the ‘drag field’ mode to move an internal field out of a subwindow. You can use the ‘drag field’ and ‘drag duplicate’ modes to move or duplicate the complete subwindow. When you duplicate a subwindow field, a new instance of the subwindow class is constructed.

The drag and drop modes for the subwindow field belong to the field rather than to the window inside the field. They are therefore not known when the subwindow is designed, so the subwindow’s methods need to either switch off unsupported modes in the $construct() method or be capable of supporting all modes.

Nesting Subwindows

You can nest container fields, such as subwindows, four levels deep. Beyond this level, the most deeply nested field is not set up when the window is opened and becomes a display field showing an error message.

A subwindow can contain a grid field and vice versa. When a grid contains a subwindow, there is only one instance of the subwindow in the grid field, and not one per line. When a row of the grid is redrawn the current field values are set up and the subwindow is redrawn. Therefore a subwindow within a grid which displays instance variables will not work correctly, since all rows of the grid will share one set of instance variables.

A grid field cannot contain a subwindow which itself contains a grid. If this occurs the nested grid is not set up, and it becomes a display field showing an error message.

Using Subwindows

The following examples use subwindows containing tab strips and pushbuttons. In a window that contains a long list sorted alphabetically, you might want to allow the user to scroll the list with a single mouse click to show items starting with a given letter. This can be done using a subwindow containing either a tab strip or a set of Rolodex-type buttons.

To create the tab strip subwindow

- Create a window and put a *Tab strip* field on it
- On the tab strip field, set the **tabs** property to *A,B,C,..,Y,Z*
- You may want to change the **selectedtabtextcolor** property to highlight the tab selected

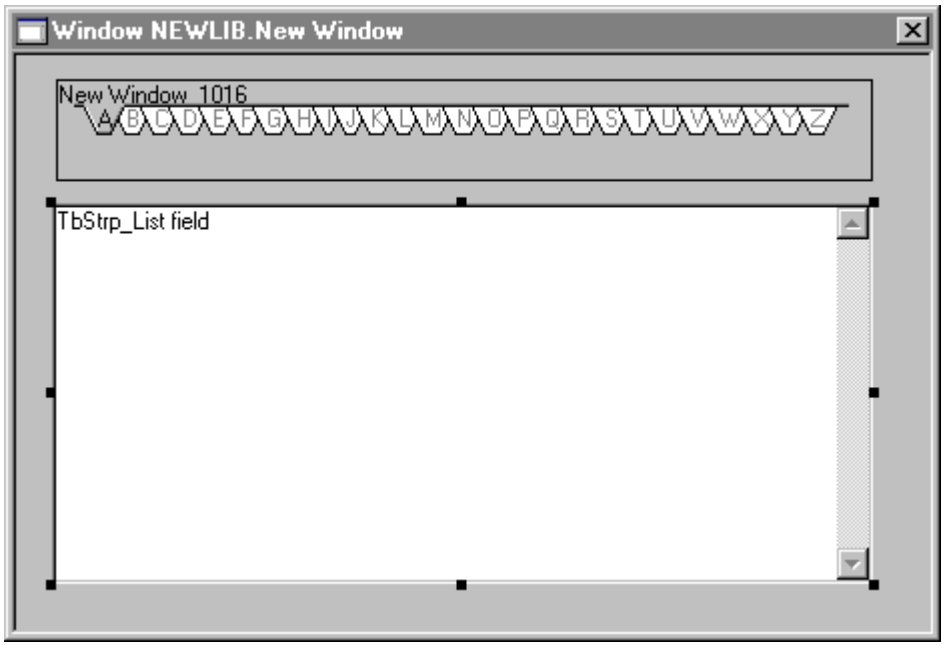
- Add a $control() method to the window and enter the single command

```
Do redirect $cwind
```

Subwindow field events are not passed beyond the subwindow field, but you can use the *Do redirect* command to redirect events to the $control() method in the parent window. The subwindow is completely generic and you could use it on any window.

- Create a new window and place a *Subwindow* field at the top
- Set the subwindow field **classname** property to the name of your tab strip window
- You may want to set the subwindow field **nobackground** property to false

- Place a List box field below the subwindow field and set its **dataname** to the name of your list variable and enter a **calculation** if necessary



You need to add the following methods to the parent window. The $construct() method builds and sorts the list using list commands, but you could equally use the $define() and $sort() list methods.

```
; $construct() method in parent window
Set current list cList
Define list {cCol1}
; build your list of data
Clear sort fields
Set sort field cCol1
Sort list
```

The $control() method in the parent window detects the tab strip event.

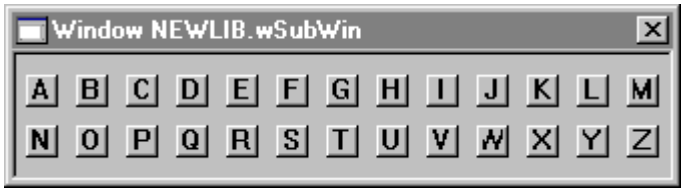
```
; $control() method in parent window
On evTabSelected
   Set search as calculation
        {upp(mid(cCol1,1,1)) = chr(64 + pTabNumber)}
   Search list (From start,Do Not Load Line)
   If flag true
      Queue scroll (Down,Page) {ListField}
      Redraw lists
   End If
```

The search calculation in the $control() method uses the *chr()* function to derive 'A' to 'Z' from pTabNumber of 1-26, and compares it to the value of the first column in the list using *mid()*. When a matching line is found, it will appear at the bottom of the list box and *Queue scroll* pages down to bring it into view.

To create the Rolodex buttons subwindow

The following example describes a subwindow containing a set of pushbuttons with the letters of the alphabet. Rather than creating a window with 26 buttons manually, you can do it automatically using the notation. You can paste this code into any method, but set up the variables first, and run it to create a window *wSubWin*.

```
; Declare variables cWRef and cRef of type Item reference
; Declare variables cLeft and num of type Number
Do $clib.$classes.$add(kWindow,'wSubWin') Returns cWRef
; returns a reference to the new window class
Do cWRef.$height.$assign(60)   ;; edit this to change the height
Do cWRef.$width.$assign(330)   ;; edit this to change the width
Calculate cLeft as 5
For num from 1 to 13 step 1
   Do cWRef.$objs.$add(kPushbutton,10,cLeft,15,15) Returns cRef
   ; returns a reference to the new object
   Do cRef.$text.$assign(chr(num+64))
   Do cWRef.$objs.$add(kPushbutton,35,cLeft,15,15) Returns cRef
   Do cRef.$text.$assign(chr(num+64+13))
   Calculate cLeft as cLeft+25
End For
```



Unlike the tab strip window described above, the parent window needs to receive the button text, which is not supplied as an event parameter, so *Do redirect* will only pass on the evClick. However, you can do this by calling a custom method, called $alphabutton() perhaps, in the parent class methods, and pass a parameter.

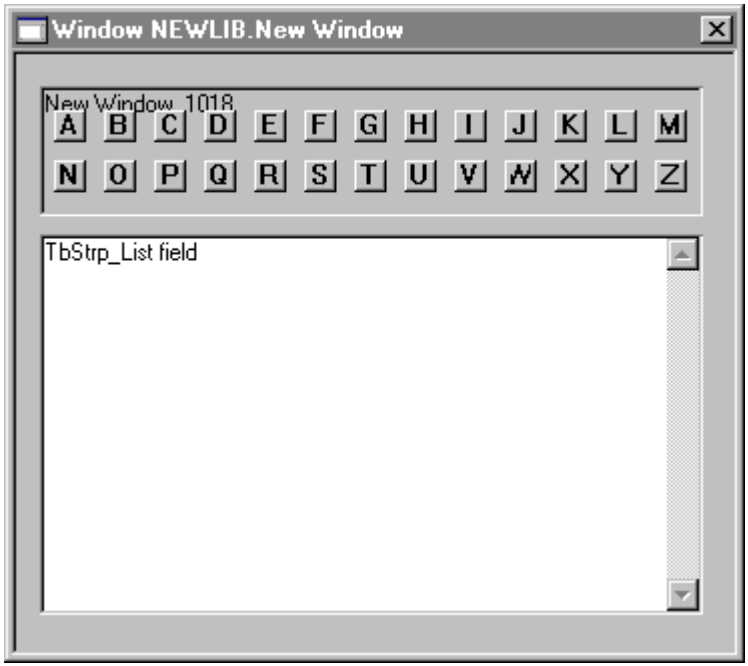
- Add a $control() method to you subwindow class containing the buttons, with the following code

```
On evClick
   Do method $cwind.$alphabutton($cobj.$text)
```

- Create a new parent window and place a *Subwindow* field at the top
- Set the subwindow field **classname** property to *wSubWin* and **nobackground** to kTrue
- Place a *List box* field below and set its **dataname, calculation,** and **multipleselect** properties

For the class methods in the parent window, the $construct() method is the same as the example above. The $alphabutton() custom method is similar to the $control() method above, but it has no event handling code and the search calculation is different

```
; declare parameter pChar of type Character
Set search as calculation {mid(cCol,1,1) = pChar}
```



To create a radio button subwindow

- Create a window with a set of radio buttons and declare a numeric variable, say iNum, for the $dataname property for each radio button
- Add the window as a subwindow field to your parent window

In the same way as for the previous examples, you can either pass up the event using *Do redirect* and get the ident of the button clicked from $cobj, or declare a custom method, say $buttonval, in the parent window, called by *Do method $cwind.$buttonval(iNum)*.

Icon Arrays

An *icon array* is a type of window field that you can use to display a list of items identified by icons. These choices are displayed as large or small icons which the user can click on or drag to select. Each icon also has a short text description which the user can edit, and you can add a button background. The data for an icon array is supplied from a list which contains the icon id and text label for each icon. The OMNIS Browser and Component Store use the icon array field, but you can build your own.

In addition to the general list field properties such as **multipleselect**, the icon array has the following properties

- **dataname**
  the list variable with at least two columns
- **maxeditchars**
  the maximum size of the edit field, or 0 if the text cannot be edited
- **smallicons**
  true for 16x16 icons, false for 48x48
- **showtext**
  displays text labels
- **buttonbackground**
  if true shows the icons on buttons
- **smalltextwidth**
  the width in pixels of the text in small icon mode; must be at least 20
- **hiliteline**
  if true lines highlight in single selection lists during drag and drop
- **autoarrange**
  adjusts the number of icon columns when the field size changes
- **enabledeletekey**
  allows the Delete key to delete the currently selected icons

To create an Icon Array

- Drag an Icon Array field from the Component Store onto your window
- Set up the general properties and those above

Programming Icon Arrays

You must set up a list variable containing the data for your icon array. You can write event handling methods to respond to user clicks, and drag and drop in the field.

Setting up the List

You must define the list variable for an icon array with at least two columns, the first column for the icon id and the second column for the text label. You can use icons from the OMNISPIC.df1 or USERPIC.df1 data files, or #ICONS in your library. You can see the id numbers in the Icon Editor, which you can also use to add your own icons to USERPIC.df1 or #ICONS. You can define and build the list in the window $construct() method.

```
; declare variable IconId (Number 0dp)
; declare variable IconName (Character)
; declare variable IconLIst of List type
Set current list IconList
Define list {IconId, IconName}
Add line to list (605,'Trash can')
Add line to list (603,'Back arrow')
Add line to list (601,'Pin')
; etc...
```

When the window instance is opened, the icon array will appear as follows: in this case, the **smallicons** property is set to true and the **smalltextwidth** property is set to 80.

Editing in the Array

In addition to the general entry field evClick, evDoubleClick, evAfter, evBefore, drag and drop events, there are specific events for editing the list and for deleting selected lines if the delete key is enabled. When text editing is enabled by **maxeditchars**, the field receives

three events in order, together with parameters holding the list line, column number, and the new text entered.

If the Delete key is enabled, two events are sent to the field:

– evIconDeleteStarting
  is sent to the field and Delete is pressed. Discarding the event prevents the delete occurring.

– evIconDeleteFinished
  is sent if the delete goes ahead, after all selected lines in the list have been deleted.

When text editing is enabled by **maxeditchars**, the field receives three events in order, together with parameters holding the list line and the new text entered.

– evIconEditStarting
  with the parameter pLineNumber, is sent on the first click in the selected cell which puts the cell into edit mode; discarding the event prevents editing

– evIconEditFinishing
  with parameters pLineNumber and pNewText, is sent if the user enters a new value by hitting return or clicking away from the edit field; discarding the event leaves the field in edit mode, for example if pNewText is invalid

– evIconEditFinished
  with the parameter pLineNumber, is sent when the edit is completed

Handlers for these events might be as follows

```
On evIconEditStarting
   If pLineNumber<10
      OK message  (Icon,Sound bell) {Cannot edit these lines}
      Quit event handler (Discard event)
   End If
On evIconEditFinishing
   If pNewText=''
      Quit event handler (Discard event)
   End If
On evIconEditFinished
   ; do anything necessary here
```

Default Methods

Icon arrays have the following method

– $edittext()
  puts the field into text edit mode if: there is a currently selected line, the field is the current field, editing is enabled by **maxeditchars**, **showtext** is true

Tree Lists

A *tree list* is a type of window field that provides a graphical way of displaying a list of items arranged in a hierarchy. The user can show or hide successive levels by expanding or collapsing the nodes. The Windows Explorer and MacOS Finder use a tree list to display the file hierarchy in your system. In OMNIS, the Notation Inspector uses a tree list to display the object tree.

The sort of information most conveniently displayed in a tree is a list sorted to several different levels. For example, a customer list might be sorted by

– Country

– Town

– Name

In a tree list, the first entry at each sort level is the *node*, with each entry at the highest level, Country, being a *root node*. Each of these nodes can be expanded to show its *child nodes*, by clicking on the *expand/collapse box*. Each node can have an icon. In practice, items may be continually added to or removed from different nodes and the tree must reflect this changing state.

Creating a Tree List

To create a Tree List

- Drag a Tree List from the Component Store onto your window

The appearance of tree lists is governed by a number of properties for the tree and for individual nodes, which you can set up either in design mode or using the notation. You can select a specific icon for each node, and for the expand/collapse box, and specify the color for the text name. You can also show lines connecting the nodes and change the horizontal and vertical spacing to accommodate large icons. The position and state of the node icons can also be set.

The tree Appearance properties are

– **treelinehtextra**
sets the distance between lines in the tree if using large icons; normally line height is controlled by the font

– **treeleftmargin**
the distance from the left before the tree starts drawing; used if root nodes are given large icons

– **treeindentlevel**
the distance between levels of nodes in the tree; used if nodes are given large icons

– **defaultnodeicon**
node icon id used by all nodes that do not already have **iconid** set

– **expandcollapseicon**
an icon id that will be used for the expand or collapse toggle button; defaults to the +- icon

– **showhorzlines** and **showvertlines**
if true show connecting horizontal and vertical lines

– **shownodeicons**
if true shows node icons

– **nodeiconpos**
controls position of the expand/collapse box if a node has children: kIconOnNode next to node icon or node name; kIconOnLeft on the left of the tree; kIconSystemSet according to the operating system (under MacOS left side of tree, under Windows and OS/2 next to the nodes icon or name)

– **treenodeiconmode**
controls the state that the nodes icon is displayed in: kNodeIconFixed normal, or checked if **checked** is set; kNodeIconLinkExpand checked if expanded, normal if collapsed; kNodeIconLinkLine checked only if node is current line

Plus the following methods

– **$expand()** and **$collapse()**
expands or collapses all nodes in the tree list

Populating a Tree List

Tree lists can either display data from a list variable or default list lines. To enter default lines the **dataname** property for the tree list must be empty. You can also populate it at runtime when a node is expanded using the $add() method or from a list variable using $setnodelist(), as described below.

To enter default lines in a Tree list

- Check **dataname** is blank
- Click in the **treedefaultlines** property

The default lines dialog lets you build a tree by adding root nodes and child nodes. You can edit the node names, change the icons, and add child levels. Note that clicking on a node shows or hides its child nodes. Choosing **defaultnodeicon** or **expandcollapseicon** shows the available icons at different resolutions.

- Right-click on a node to modify it

- **Always Show Expand Box**
  shows an expand/collapse box even when there are no child nodes: this toggles the node property $showexpandcollapsealways
- **Enterable**
  lets the user edit the node name
- **Node Color**
  presents a palette to choose node color; restored by **Default Color**
- **Node Icon**
  displays the available icons; restored by **Clear Icon**
- **Node Ident**
  lets you enter an ident number, cleared by **Clear Ident**; you should assign node idents since the names in your tree list may often be duplicated

- Click **Accept Lines** to enter the finished structure

When you open your window, the tree list displays all the properties you have set up.

Node Properties

A node has the properties

- **showexpandcollapsealways**
  if true the node will always draw an expand/collapse box; useful for populating a node on evTreeExpand event
- **iconid**
  id for the node icon: if 0, uses the default icon for the current OS
- **ident**
  a number you can assign to the node and refer to when node messages are received
- **textcolor**
  the color the node name is drawn in: if kColorDefault, the tree control's **textcolor** is used
- **name**
  node name: the name is the visible string part of the node in the tree
- **enterable**
  if kTrue, the node name can be edited in the tree
- **seedid**
  if true shows the icon id, a unique number assigned to each node by the tree
- **nodeparent**
  returns an item reference to the node's parent node

– **isexpanded**
  true for nodes which are in the expanded state

– **checked**
  if true node icon is drawn in the checked state: drawing of the node icon in check mode also depends on **treenodeiconmode**, see above

– **first**
  used on the node, returns an item reference to the first child node of the calling node

– **level**
  returns a number indicating the indent level of the node: level 1 indicates root nodes

Plus the following methods

– **$clearallnodes()**
  used on a node to clear all child nodes recursively

– **$count()**
  counts child nodes of the calling node

– **$expand()** and **$collapse()**
  used on the node to expand or collapse the node's child nodes

Programming Tree Lists

The $event() method for tree lists receives specific event messages in response to user actions in the tree. When a node is clicked on, the pNodeItem event parameter is sent holding an item reference to the node clicked on, so you can take action for that node. You can manipulate nodes with the following methods

– $clearallnodes()
  clears the tree of all nodes

– $count()
  returns the number of root nodes in the tree

– $add(name, ident)
  adds a new child node to the calling node: ident is optional and defaults to 0; returns an item reference to the new node

– $remove(itemref)
  looks for a child node itemref and removes it

– $getvisiblenode()
  returns a node item reference for a visible line

Expanding and Collapsing Nodes

The evTreeExpand event indicates a node is about to be expanded and provides the reference in pNodeItem. You would use this to populate a node using $add(), for example:

```
On evTreeExpand
   Set Reference NewNode to pNodeItem.$add('NewNode', 100)
   Calculate NewNode.$textcolor as kRed
```

If you have set up your nodes as default lines as described above and given them idents, such as:

Windows	100
Reports	200

you can expand the node from the appropriate list.

```
On evTreeExpand
   Set Reference TreeRef to $cwind.$objs.TreeList
   Switch pNodeItem.$ident
      Case 100
         Do TreeRef.$setnodelist(kRelationalList,pNodeItem,tWinList)
```

The evTreeCollapse event indicates a node is about to be collapsed and provides the reference in pNodeItem. You would use this to clear child nodes, for example:

```
On evTreeCollapse
   Do pNodeItem.$clearallnodes()
```

The evTreeExpandCollapseFinished event is sent to confirm that the evTreeExpand or evTreeCollapse message is finished. You can use this event to update other controls or states.

The evTreeNodeIconClicked event message is sent when the user clicks on a node icon. The second event parameter provides the name of the node clicked on.

Changing a Node Name

The evTreeNodeNameChanging event is sent to a tree list before the node is updated with some new value entered by the user. It provides the parameters pNodeItem for the node, and a character variable pNewText containing the new text entered. pNodeItem.$name still holds the original text. This message is normally used to validate node name changing.

When the user changes the name or ident of an enterable node, it is important, particularly for idents, to check that the value entered does not already exist. These two methods search for a match.

– $findnodename(itemref, name, recursive)
  returns an item reference to a found node using the node $name property for comparison, or NULL if nothing is found: itemref is the starting node; a NULL value searches the full tree; name is the name to search for; if recursive is kTrue, nodes with children are also searched

– $findnodeident(itemref, ident, recursive)
returns an item reference to a found node using the node $ident property for comparison, or NULL if nothing is found: itemref is the starting node; a NULL value searches the full tree; ident is the $ident value to search for; if recursive is kTrue, nodes with children are also searched. For example

```
On evTreeNodeNameChanging
   If pNewText = ''
      OK Message ( 'Name must contain a value')
      Quit event handler ( Discard Event )
   Else
      Do TreeRef.$findnodename(pNodeItem,pNewText,1) returns Found
      If Found != NULL
         OK Message ('Name must be unique')
         Quit event handler ( Discard Event )
      End if
   End if
```

The evTreeNodeNameChanged event is sent to a tree list after the node name has been changed. It provides the parameters pNodeItem for the node and a character variable pNewText containing the new text entered. For example

```
On evTreeNodeNameChanged
   OK Message('Tree node has been updated')
      .. update the status
```

Traversing the Tree

The following methods fetch a reference to the current or first node, or change the current node.

– $currentnode()
returns an item reference to the tree's current node

– $first()
returns an item reference to the first root node

– $setcurrentnode(itemref)
sets the current node to that specified in itemref

The following methods go to the next or previous node.

– $nextnode(itemref, recursive)
returns the next node in the tree given a previous node itemref; if itemref is NULL, returns first root node; if recursive is kTrue, operation steps into nodes with children

– $prevnode(itemref, recursive)
returns the previous node in the tree given a node itemref; if recursive is kTrue, the operation will step back into node parents

```
Do TreeRef.$nextnode(pNodeItem,0) Returns NextNode
Do TreeRef.$nextnode(Null,0) Returns RootNode
```

Interchanging Data with Lists

Using the methods $setnodelist() and $getnodelist() you can either populate the whole tree or a node from a list variable, which must contain a sorted list, or retrieve data from the tree to a list variable.

– $setnodelist(listmode, noderef, listname)
  lets you populate the tree or a tree node from a list: listmode is either kRelationalList or kFlatList; noderef is either NULL to populate the entire tree, or a node reference; listname is name of a list variable e.g. tList

– $getnodelist(listmode, noderef, listname)
  lets you retrieve information from the tree, or from a node and its children into a list: listmode is either kRelationalList or kFlatList; noderef is NULL to retrieve the entire tree, or a node item reference; listname is name of a list variable e.g. tList

The list mode is either *relational* or *flat*. For a relational list, you supply a Null node reference for the whole tree, and list data such as:

RootNode	Child 1	
RootNode	Child 2	Child 1
RootNode	Child 2	Child 2
RootNode	Child 2	Child 3
RootNode	Child 3	
RootNode 2	Child 1	
RootNode 2	Child 2	

To populate the whole tree from a relational list, you would use the line

```
Do $cwind.$objs.TreeList.$setnodelist(kRelationalList,0,tList)
```

In this case the tree contains all the information but the nodes are all in the default state. The flat list option lets you specify the node property settings $iconid, $ident, $enterable, $expandcollapsealways and $textcolor as the final 5 list columns. For example:

Name	Name	$iconid	$ident	$enterable	$expand collapse	$textcolor
RootNode		0	100	0	0	0
RootNode	Child 1	0	101	0	0	0
RootNode	Child 2	0	102	0	0	0
New Root		0	200	0	0	0
New Root	Child 1	0	201	0	0	0
New Root	Child 2	0	202	0	0	0
Last Root						

The command

```
Do $cwind.$objs.TreeList.$setnodelist(kFlatList,0,tList)
```

draws the tree list with the node properties set as specified in the list. This example assigns the list to an existing node:

```
Set Reference CurrentNode to TreeRef.$currentnode
Do CurrentNode.$setnodelist(kFlatList,0,tList)
```

To retrieve data from the tree, $getnodelist() does the opposite to $setnodelist() and transfers data from the tree to a list, either as a relational list or a flat list, as above. There is no need to define the list first. This line retrieves the whole tree and its node properties to a list:

```
Do TreeRef.$getnodelist(kFlatList,0,tList)
```

This code retrieves the current node data but no node properties to a list:

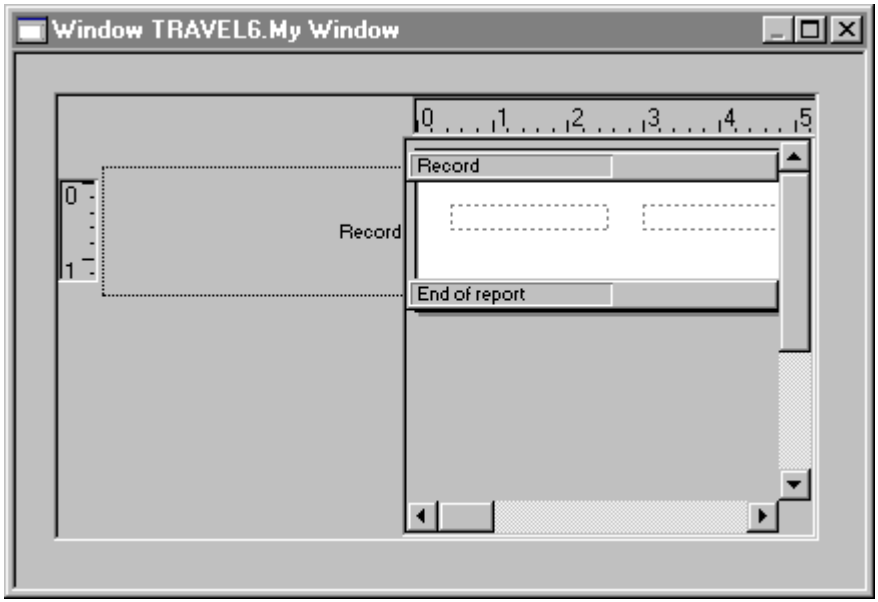
```
Set Reference currentNode to TreeRef.$currentnode
Do currentNode.$getnodelist(kRelationalList,0,tList)
```

Modify Report Fields

A *modify report field* is a type of window field that lets you display a report class on an open window. This allows your users to change certain aspects of the report class at runtime, including the height of the Record section, the contents of headers and footers, the position and color of graphics on the report, and so on. When you create a modify report field you specify the classname of the report to be displayed in the field.

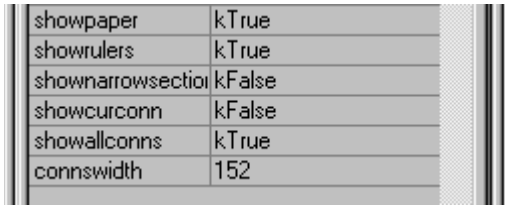
To create a modify report field

- Open your window in design mode
- Drag a *Modify Report Field* from the Component Store onto your window



- Open the Property Manager or press F6/Cmnd-6 to bring it to the top
- Select the **classname** property and enter the name of your report class

The modify report field has all the properties of a standard window field in addition to the following Appearance properties.



You can hide or show the outline of the paper and the rulers with **showpaper** and **showrulers**. You can hide or show the current or all connections for associated report sections with **showcurconns** and **showallconns**, and you can set the width of the connections shown in the left margin by setting **connswidth**. You can show the report sections as narrow lines by enabling the **shownarrowsections** property. You can also change these properties at runtime.

To make the modify report field fill the entire window you can set its **edgefloat** property to kEFposnClient.

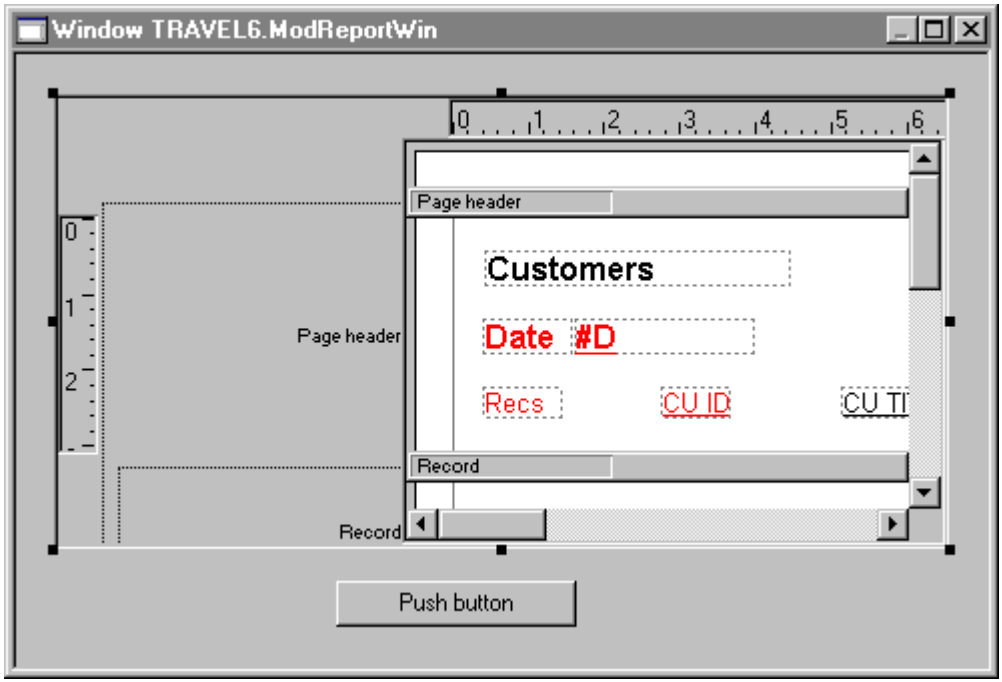
Along with the common $redraw() method for a field, an instance of a report modify field has the methods $sortfields() which opens the OMNIS Sort fields dialog for the report contained in the field, and $pagesetup() which opens the standard Page Setup dialog.

A modify report field generates an evSelectionChanged event which you can detect in the $event() method for the window field.

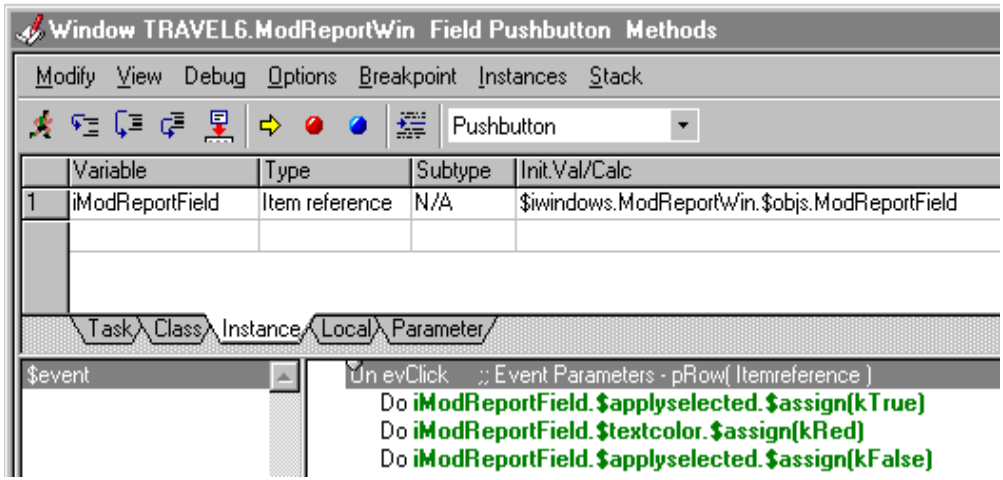
Applying changes to selected objects

To change individual objects inside a modify report field at runtime you need to set its $applyselected property. When the $applyselected property is set to kTrue any property changes you direct at the modify report field, such as font and appearance changes, apply to the currently selected object inside the modify report field. For example, the following window contains a modify report field; its **classname** property is set to contain a simple summary style report that lists data from a Customers file. The window also contains a single pushbutton, and an instance variable that stores a reference to the modify report field.

Here’s the window



The pushbutton contains the following $event() method. Note that the variable iModReportField stores a reference to the modify report field on the open window.



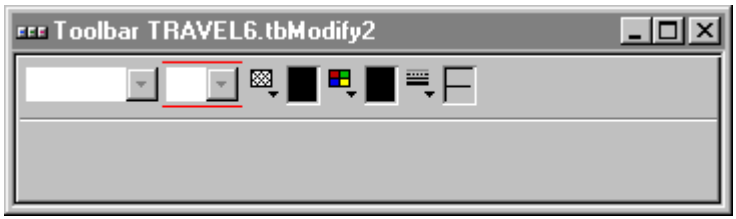
When you open this window, select an object inside the modify report field, and click on the pushbutton, the method changes the text color of the currently selected object to red. Note

that you have to set the $applyselected property to kFalse when you have finished your changes.

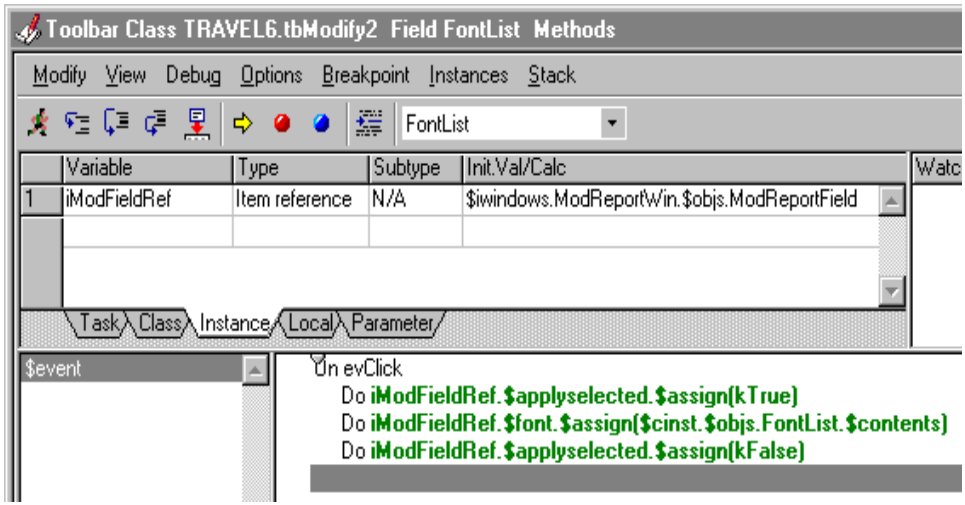
Font and Color Tools

Rather than using pushbuttons to change a Modify Report Field as above, you can create your own set of toolbars and install them in your window containing the report field. The Component Store contains a number of toolbar controls and pickers for setting fonts, lines, and colors that you can use with the modify report field. A further example will demonstrate using toolbars with the modify report field.

You can create the following toolbar class containing the appropriate font, line, and color pickers, add suitable icons from the icon data file or use the default ones, and add it to your window. Here's the toolbar class



Each toolbar control contains a method that applies the current settings from the control to the selected object in the modify report field. The following method is for the font list, the first control on the toolbar.



Note that the toolbar class also contains an instance variable of type Item reference that stores a reference to the modify report field on the open window, and that the method sets

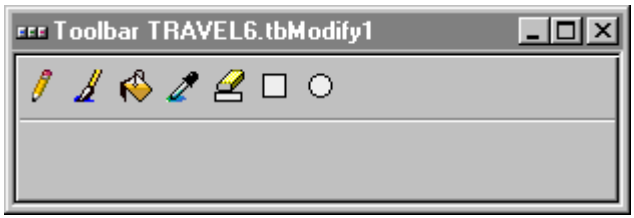
$applyselected. The methods behind the other tools on the toolbar are very similar; here's the method for one of the color pickers

```
; $event() method for forecolor picker control
On evClick
   Do iModFieldRef.$applyselected.$assign(kTrue)
   Do iModFieldRef.$forecolor.$assign($cobj.$contents)
   Do iModFieldRef.$applyselected.$assign(kFalse)
```

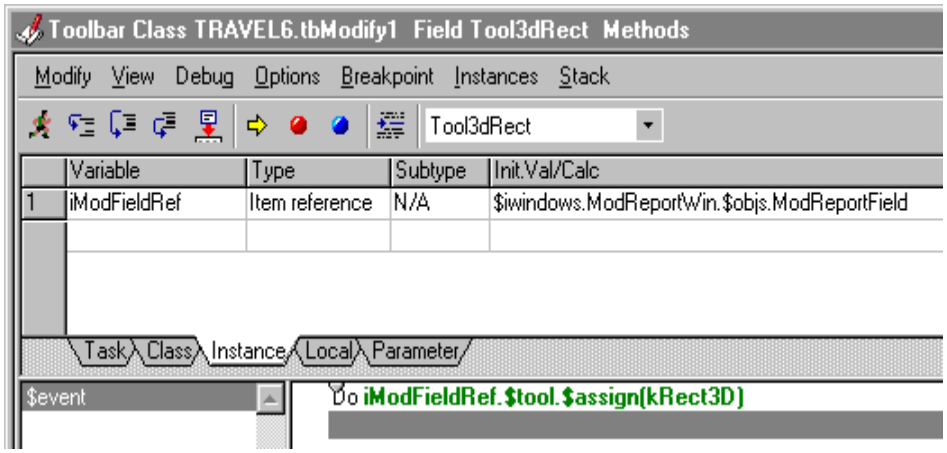
Note that the current selection in a picker control is returned in its $contents property, therefore as the user makes a selection you can use $cobj.$contents to return the value.

Graphics Tools

At runtime, a modify report field has the $tool property which you can set to allow users to place graphics or background objects on your report; you cannot add fields and other foreground objects to a modify report field. You can create another toolbar that uses the $tool property and add it to your window. The toolbar class can contain various button controls, with suitable icons from the icon data file.



Each button in your toolbar class contains a single method that assigns to the $tool property and switches the cursor to the appropriate tool. For example, a 3D Rectangle button could contain the following method; note that the toolbar class also contains an instance variable of type Item reference that stores a reference to the modify report field on the open window.



When you open this window, select one of the tools, and move the cursor over the modify report field, the cursor changes to a cross-hair. The user can draw objects on the modify report field which are saved to the underlying report class automatically.

To use the modify report field to its fullest potential you need to build a number of toolbars that allow the user to change every aspect of the report class, including margins, page setup, sort fields, as well as the color and style of objects on the report. The modify report field is used extensively in the Ad hoc report library supplied with OMNIS.

Screen Report Fields

A *screen report field* is a type of field that lets you display the output of a report on a window, rather than sending the report to a standard screen report. You use the *Send to a window field* command to direct output to a screen report field. The user can copy data from a screen report field instance by dragging the mouse on the report to select some data.

To create a screen report field

- Open your window in design mode
- Drag a *Screen Report* Field from the Component Store onto your window

The screen report field has all the properties of a standard window field in addition to the **showpaper** under the Appearance tab in the Property Manager. If you set $showpaper to true, it changes the field to page preview mode.

This type of field does not have a **dataname** or **classname**, and it does not generate any events of its own apart from the events for a standard field.

You could put the following method behind a pushbutton on your window or a toolbar control to print to your screen report field.

```
On evClick
   Set report name ReportName
   Send to a window field {ScreenReportFieldName}
   Print report
```

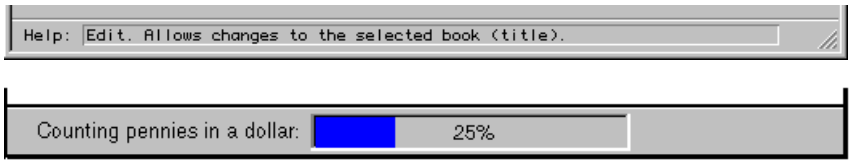
The screen report field has two methods:

– $zoom(bZoomOn=kTrue)
enables zoom mode when the screen report field is in page preview mode

– $redirect(bPrompt=kTrue)
redirects the current report by prompting for a different print device, rather than the device specified in default preferences

Window Status Bars

A *window status bar* is an area at the bottom of a window in which you can display data, text, help messages, progress or thermometer bars, and so on. For example



A status bar is a property of the window itself which you enable in the Property Manager. You can set how many panes should appear in the status bar, and the size and style of each pane.

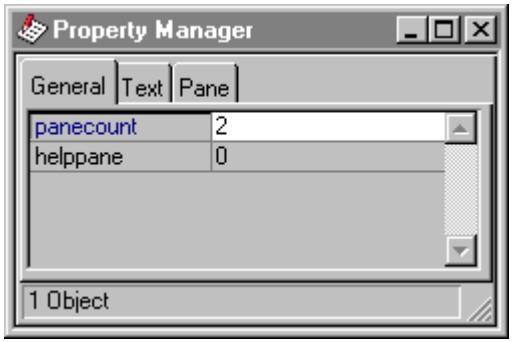
To enable a window status bar

- Open your window in design mode
- Click on the background of the window to show its properties, or press F6/Cmnd-6 to bring the Property Manager to the top
- Set the **hasstatusbar** property to kTrue
- Click on the Appearance tab in the Property Manager and set the **statusedge** property: it can be flat (the default), plain, inset, or chisel border style

To set the number of panes in the status bar and their style you need to edit the properties of the status bar in the Property Manager.

To set the number of panes in the status bar

- Click on the status bar and bring the Property Manager to the top

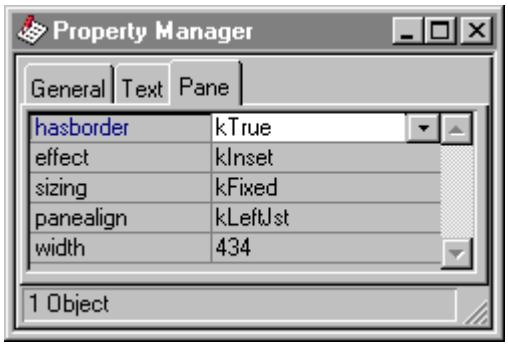


The **panecount** property specifies the number of panes in the status bar. The **helppane** property specifies the pane in which any help messages should appear, held in **helptext** for menu lines, for example. On the Text tab you can set the **font** and **fontsize** properties for the whole status bar.

To change the properties of individual panes you should click on the pane and edit its properties in the Property Manager.

To change the properties of a pane

- Click on a pane in the window status bar and click on the Pane tab in the Property Manager



You can change the pane's border, alignment, and width in pixels. The **sizing** property sets the pane to fixed or elastic when the window is sized at runtime. The minimum size of an elastic pane is the size of the pane in design mode: it cannot be made smaller in runtime.

The height of the status bar changes to accommodate the status bar font size with a two-pixel buffer above and below. In design mode you can change the width of a pane by dragging the handle that appears in the selected pane.

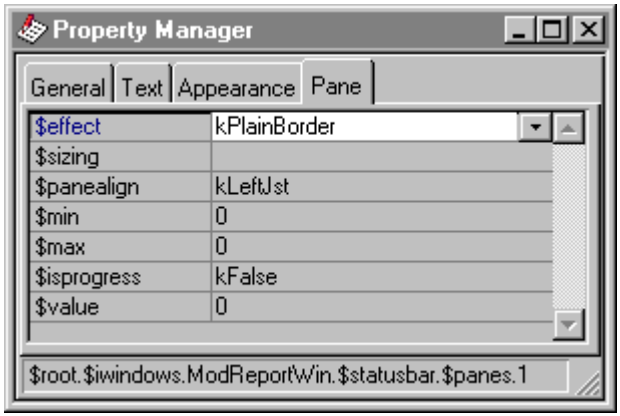
Every window instance contains the $statusbar property containing the window status bar in runtime. The $hasstatusbar property lets you hide and show the status bar at runtime. The $statusbar property also contains a group $panes containing the panes in the status bar numbered consecutively from the left. For example, pane 2 is $iwindows.WindowName.$statusbar.$panes.2. Each pane has width, text and appearance properties which you can set at runtime. For example

```
; declare item references to the panes
Set reference Pane1 to $cinst.$statusbar.$panes.1
Set reference Pane2 to $cinst.$statusbar.$panes.2
Set reference Pane3 to $cinst.$statusbar.$panes.3
Do $cwind.$statusbar.$panes.$remove(Pane3) ;; removes the third pane
Do Pane2.$text.$assign('Click Save button to save your work')
Do Pane3.$sizing.$assign(kElastic)
Do Pane1.$hasborder.$assign(kFalse)
```

The $align property for a pane specifies whether to position the pane either after the previous left-hand pane or before the right-hand pane.

Progress Bars

In runtime, you can make a pane into a progress bar by enabling its **isprogress** property. If you want to view the properties of the status bar in runtime you can view it in the Notation Inspector under $iwindows. The properties of the status bar on a window instance are



– **min** and **max**
sets the minimum and maximum value for the progress bar

– **isprogress**
enables the pane as a progress bar

– **value**
sets the current value on the progress bar

When $isprogress is set, $min and $max default to 0 and 100 respectively, but if you set them after setting $isprogress, your values will override the default settings. For example, to set $max for the second pane:

```
Do Pane2.$isprogress.$assign(kTrue)
Do Pane2.$max.$assign(200) ;; default for $min is zero
```

The defaults for $min and $max are useful for percentages, for showing the percentage completed for an operation. The following method sets up a progress bar in the second pane and uses the default values for $min and $max:

```
Set reference Pane1 to $cinst.$statusbar.$panes.1
Set reference Pane2 to $cinst.$statusbar.$panes.2
Do Pane1.$hasborder.$assign(kFalse)
Do Pane1.$text.$assign("Doing Loop")
Do Pane2.$isprogress.$assign(kTrue)
; now set max if required e.g. Do Pane2.$max.$assign(maxvalue)
Do Pane2.$backcolor.$assign(kRed)
Calculate Pane2.$value as Pane2.$min  ;; resets value of pane2
Repeat
   Calculate Pane2.$value as Pane2.$value+1
Until Pane2.$value>=Pane2.$max
Do Pane1.$text.$assign("Ready")
Do Pane2.$isprogress.$assign(kFalse)
```

The resulting progress bar looks something like this:

You can add an icon or picture from the USERPIC.DF1 data file or #ICONS to the progress bar, either from the Property Manager or with a command. For example, to have a show of hands as your bar add the line:

```
Do Pane2.$iconid.$assign(1072)
```

As a further refinement, you can add a ‘% Done’ message to the progress bar using the current $value of the pane inside the loop.

```
Repeat
   Calculate Pane2.$value as Pane2.$value + 1
   Calculate Pane2.$text as
      con(rnd(((Pane2.$value/Pane2.$max)*100),0),”% Done”)
Until Pane2.$value = Pane2.$max
Calculate Pane2.$text as “Finished!”
```

Field Styles

If you are going to deploy your application on more than one platform the objects in your application should use the correct fonts for each platform. You can do this using field styles, which is a flexible alternative to using font tables. A field style is a style definition, like a wordprocessing or DTP style, that you can apply to window and report objects. Each field style contains a definition of its name, font, size, text color, typestyle, and alignment. You can create a separate definition for each platform under the same style name, so objects will display in the appropriate font and point size under different platforms.

OMNIS has some default styles for standard entry fields, pushbuttons, and lists, but you can add your own styles. The style for a particular window or report field or text object is stored in its **fieldstyle** text property.

You can create as many field styles as you like; they are stored in the #STYLES system table. Having set up the styles in the style table, you can copy the table to any library and use its styles throughout your whole application. When you copy an object from one library to another, its field style is also copied automatically if there is not already one with the same name in the destination library.

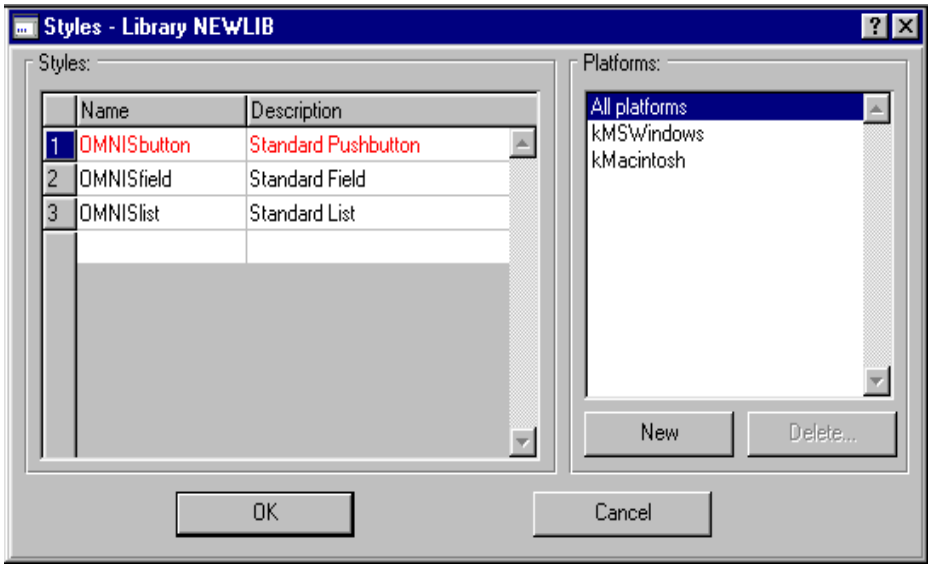
Note that the fonts shipped with previous versions of OMNIS are not supported in OMNIS Studio. Field styles should be used instead.

To view the field styles system table

- Press Ctrl/Cmnd-A to show all classes in the Browser

or

- Open the Browser Options dialog (press F7/Cmnd-7 while the Browser is on top) to show the system tables
- Double-click on #STYLES



Defining Field Styles

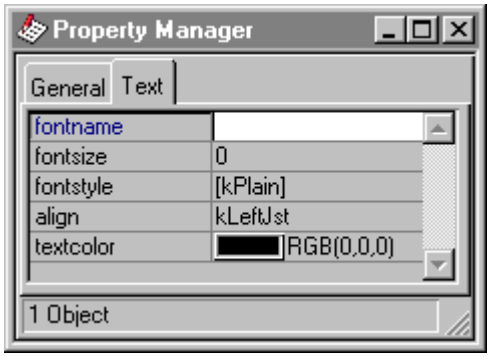
The Styles dialog lists all the styles in the current library, including the default styles. To define a style, first you enable the characteristics for that particular style, that is, you specify whether or not a style has a font name, size, style, alignment, and text color. Then you set the font characteristics for the style for each platform.

To define a new style

- Click in the first empty line in the table and enter a name and description for the style
- With the kAllplatforms constant selected in the right-hand list, open the Property Manager, or bring it to the top
- Set the **hasfontname**, **hasfontsize**, **hasfontstyle**, **hasalign**, and **hastextcolor** properties to kTrue as required

For example, if you want a style to set the font name and size only, enable only the **hasfontname** and **hasfontsize** properties.

- Go back to the Styles dialog and select a platform from the list on the right
- Bring the Property Manager to the top and on the Text tab define the font characteristics for the style



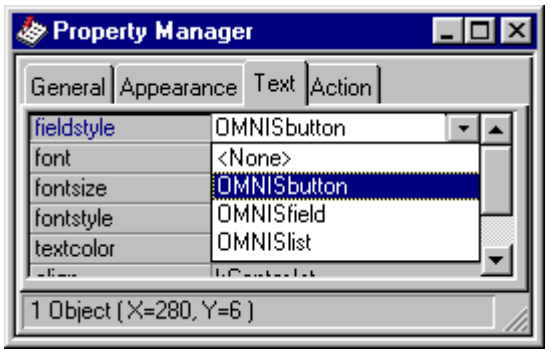
You type in the font name and size and select the style from the checklist. This definition is for the platform you selected. To define the characteristics for another platform for the same style, go back to the Styles dialog and select another platform, and define its font characteristics in the Property Manager as described. When you have set up the characteristics for the style click on OK in the Styles dialog.

You can change a default style by clicking on its name in the Styles dialog and editing its properties in the Property Manager.

Applying a Style to an Object

The field style for a window or report object is stored as a property. You can set this for window fields under the Text tab in the Property Manager.

- Open your window or report class
- Click on a field or text object and view its properties
- On the Text tab set the **fieldstyle** property to the required style



The properties that you have enabled using the **hasfont...** properties will override the text properties in the object. For example, if you have set **hasfontname** and **hasfontsize** in your style definition, these properties will apply to an object with that style name, whereas the other text properties will remain unaffected. Once you apply a field style to an object you can no longer set the text properties controlled by the style; these become grayed in the Property Manager.

If your style does not appear in the fieldstyle property list it usually means your style has no font definition; go back to the Styles system table and make sure you have specified a font name, size, and so on, for the style.

The library preference **styleplatform** controls which set of text characteristics defined in the style is used on the current machine. For example, if you are running OMNIS under Windows, **styleplatform** will be set to kMSWindows and the text characteristics defined in the style for Windows will be used.

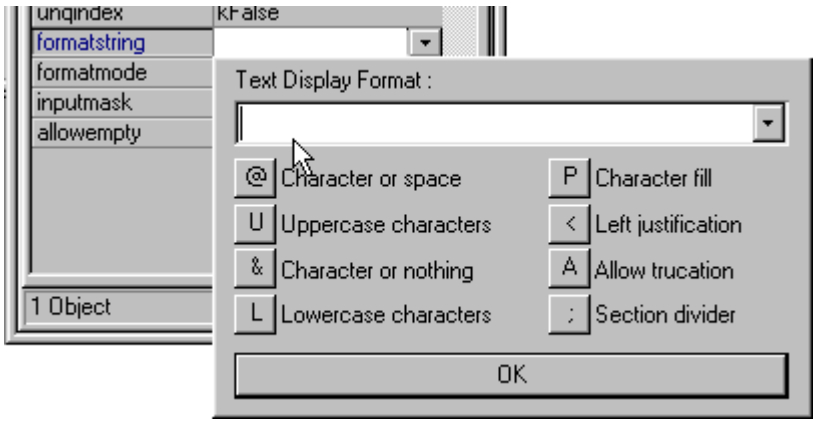
Format Strings and Input Masks

A *format string* is a set of characters or symbols that formats the data in a field for display, regardless of how the data is stored. The string is stored in the **formatstring** property for the field. An *input mask* formats data as you enter it into a field, and is stored in the **inputmask** property. On a window, only the masked entry field allows a formatting string or input mask. When a user enters data into a field controlled by an input mask, OMNIS rejects any text that does not conform to the format you've specified in the mask. Report data fields also support format strings.

To enter a format string for a field, you need to specify the type of data represented in the field, that is, its **formatmode**: this property can be Character, Number, Date, or Boolean. You can enter a format string manually or use one from the dropdown list in the format string dialog: the default formats in this dropdown are stored in the appropriate system table.

Character Format Strings

To format a text field you have to set its **formatmode** property to Character. Character format strings have one or two sections. The first section contains the value display format; the second section contains the format to display for NULL or empty values. When you click on the **formatstring** property, a dialog appears that lets you select a format. The dialog is the same for all the different formats.



You can enter a format directly into the Text Display Format field, either from the keyboard or using the buttons in the dialog. Alternatively, you can click on the down arrow in the Text Display Format field and select a format from the #TFORMS system table. The character formatting strings for the current library are stored in #TFORMS. You can edit #TFORMS by double-clicking on it in the Browser.

You can use the following symbols in character formats:

- **@**
  represents a single character or space
- **&**
  represents a single character but not a space
- **U**
  forces all characters in the field to upper case
- **L**
  forces all characters in the field to lower case
- **<**
  fills placeholders from left to right for left adjustment of the field; must be leading characters in field
- **A**
  truncates the value if it exceeds format length. It truncates the front of the string; use the sequence <A to truncate the end of the string
- **P**
  character fill; Px fills the front of the string with the character x to make the string the required length
- **;**
  section separator

Example character format strings

Format string	ANT	adder	Antelope	Null
@	ANT	adder	Antelope	
U	ANT	ADDER	ANTELOPE	
L'Text: '&	Text: ant	Text: adder	Text: antelope	
Px&&&&&&&&&	xxxxxANT	xxxadder	Antelope	
<Px&&&&&&&&&	ANTxxxxx	adderxxx	Antelope	
A&&&&	ANT	dder	lope	
<A&&&&	ANT	adde	Ante	
&;'Null text value'	ANT	adder	Antelope	Null text value

Number Format Strings

To format a numeric field you have to set its **formatmode** property to Number. Number formats can use the following symbols in a format string.

– **0**
zero; displays a digit; displays leading or trailing zeros for the format length; rounds to number of decimal places

– **#**
a digit as for 0 but does not display leading or trailing zeros

– **?**
a digit as for 0 but displays a space for leading or trailing spaces for the format length

– **.**
decimal placeholder

– **%**
percentage placeholder

– **E-, E+, e-, e+**
displays the number in scientific notation

– **$, -, +, (,)**
display exactly as you type them in

– **P**
character fill; Px fills the front of the string with the character x to make the string the required length

– **;**
section separator

The Numeric format string contains up to four sections: which format positive values, negative values, zero values and NULL values respectively. An empty format section consisting of two contiguous semicolons will cause the positive format section to be used. Additionally, if the format string contains less than four sections, the positive section will be used for the unspecified sections. Null values will only be formatted using the NULL section.

Example numeric format strings

Format string	1234.47	-1234.47	0	Null
0	1234	-1234	0	
0.0	1234.5	-1234.5	0.0	
#,##0.00	1,234.47	-1,234.47	0.00	
#,##0;(#,##0)[red]	1,234	(1,234)	0	
0;(0);'Zero';'Nil'	1234	(1234)	Zero	Nil
0.00E+00	1.23E+03	-1.23E+03	0.00E+00	
+Px#,###,###;-Px#,###,###	+xxxx1,234	-xxxx1,234	+xxxxxxxxx	

The number formatting strings for the current library are stored in #NFORMS. You can edit #NFORMS by double-clicking on it in the Browser.

Date Format Strings

To format a date field you have to set its **formatmode** property to Date. The display formats of all date and time fields are controlled by date format strings. #FD is the date format string which is used to display short dates, #FT is the date format string which is used to display short times, and #FDT is the default date format string which is used to display long dates.

Date format strings contain twenty special characters that denote the positions where the string displays the year, month, day, hour, minute, second or hundredths of second. All other characters in the date format string display unchanged (note, for example, the colons in the sample strings below). The Date codes item on the Constants tab in the Catalog contains a list of all the special date format characters. There are options to display the hour in 24 or 12 hour format with an AM/PM position.

N is the character for displaying minutes; M and m indicate the month.

Using the date and time of 20 minutes past 1 p.m. on the 12th of January 1994, a date time value displays as:

– 12 JAN 98 13:20 if #FDT is `D m Y H:N'

– 12 JAN 98 1:20 PM if #FDT is `D m Y h:N A'

– 12th 01 1998 13:20:00.00 if #FDT is `d M y H:N:S.s'

The date formatting strings for the current library are stored in #DFORMS. You can edit #DFORMS by double-clicking on it in the Browser.

Boolean Format Strings

To format a boolean field you have to set its **formatmode** property to Boolean. Format strings for boolean fields contain up to three sections: the first formats True values, the second formats False values, and the third formats NULL or Empty values. You can use the following formatting symbols:

– t
  displays “T” or “F” for true or false values

– T
  displays “True” or “False” for true or false values

– y
  displays “Y” or “N” for true or false values

– Y
  displays “Yes” or “No” for true or false values

– 1
  displays “1” or “0” for true or false values

– O
  the letter “O”; displays “On” or “Off” for true or false values

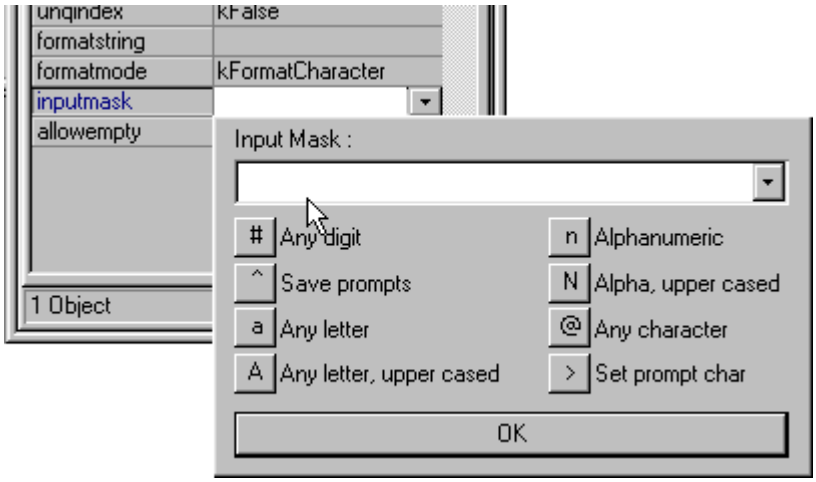
Example Boolean format strings

Format string	1	0	Null
T	True	False	
'True';'False'	True	False	
T;Y;'Null Boolean'	True	NO	Null Boolean

The boolean formatting strings for the current library are stored in #BFORMS. You can edit #BFORMS by double-clicking on it in the Browser.

Input Masks

Input masks control the format of data entered by the user. The input field for a field is stored in its **inputmask** property. When you click on the **inputmask** property, a dialog appears that lets you select a mask.



You can enter a mask directly into the Input Mask field, either from the keyboard or using the buttons in the dialog. Alternatively, you can click on the down arrow in the Input Mask field and select a mask from the #MASKS system table. The input masks for the current library are stored in #MASKS, which you can edit by double-clicking on it in the Browser.

An input mask can contain a number of characters together with literal display characters. The literal characters are presented to the user when the mask is used for data entry in order to provide context to the surrounding mask placeholder characters. The mask characters can either consist of placeholders or mask control characters. Placeholders are replaced by user characters of the appropriate type during data entry.

You can use the following mask placeholders:

Placeholder	Meaning
#	any digit
@	any character
a	any letter
A	any uppercase letter
n	alphanumeric
N	alphanumeric, upper-cased
"ABC"	any character from list, i.e. either A, B or C
"A-D"	any character from A to D inclusive

Mask control characters control how the mask is presented to the user and how the data is saved to the underlying field. You can use the following control characters:

Symbol	Meaning
^	stores literal characters in underlying field
\C	displays next character literally
>C	uses following character to prompt user
>>	displays default prompt characters

By default, the underscore character is used at data entry to represent placeholder characters yet to be entered. You can configure this character on a per placeholder basis using the '>' symbol. When the character sequence '>>' occurs at the start of the input mask, the default numeric prompt will be a hash sign; other placeholders are displayed as an ampersand.

The following table contains some example input masks, together with the string that is initially displayed to the user and an example value that can be entered to satisfy the mask.

Input mask	Initial display	Example value
(###) ###-####	(___) ___-____	(717) 321-8745
>>(###) ###-####	(###) ###-####	(717) 321-8745
aa ## ## ## a	__ __ __ __ _	xy 12 34 56 z
>>aa ## ## ## a	@@ ## ## ## @	xy 12 34 56 z
>?AA >*## ## ## >?A	?? ** ** ** ?	XY 12 34 56 Z
Enter digit #	Enter digit _	Enter digit 1
>>aaaaaaaa	@@@@@@@@	Antelope
>>aaaaaaaa	@@@@@@@@	Baboon
>?"0-5"	?	4

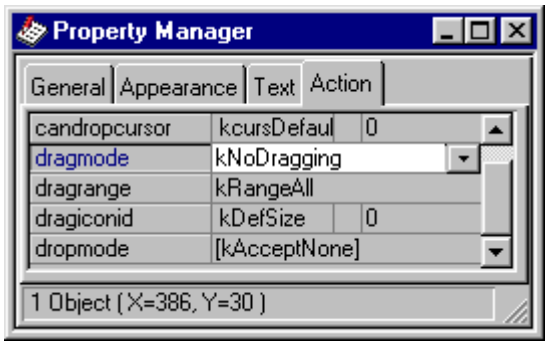
By default, literal characters occurring in the input mask are simply used to aid data entry. They are not saved to the underlying variable or field. Therefore, when performing queries on the saved data the user must remember not to search for the literal characters. In the first example above, the string '7173218745' would be saved. To show this data correctly, you must add a display format to the entry field.

To enable the user to save literal characters to the underlying variable field, you can put a circumflex character '^' in the input mask. In the first example above, an input mask of '^(###) ###-####' would cause the string '(717) 321-8745' to be saved. In this case it would be inappropriate to place a display format on the associated entry field.

Drag and Drop

Drag and drop is a powerful feature that lets the user copy data and objects from one field to another, or from one window to another. For example, in a human resources application you could build a list of employees and allow the user to select certain employees and drag them onto a print button to print those employee details; in a stock control system, the user could add items to a dispatch note by dragging the items from a stock list into the dispatch window; and so on.

The drag and drop capabilities of a field are properties of the field itself. Windows also have some drop properties. You can set these properties under the Action tab in the Property Manager, or you can use the notation. The field properties are



– **dragmode**
  sets whether or not the data or whole object is dragged and/or duplicated: includes kNoDragging (the default), kDragData (drags the data only), kDragDuplicate (drags a copy of the object), kDragObject (moves the object without copying)

– **dragrange**
  limits the scope of where a field can be dragged to: includes kRangeAll (can be dragged anywhere in the application), kRangeTask (within the current task), kRangeSubwindow (within a subwindow if the field is in a subwindow), kRangeWindow (within the current window only)

– **dragiconid**
  sets the icon for the object while it is being dragged

– **dropmode**
  determines what types of object or objects the field will receive: includes kAcceptAll, kAcceptButton, kAcceptComboBox, kAcceptDroplists, kAcceptEdit, kAcceptGrid, kAcceptList, kAcceptNone, kAcceptPicture, kAcceptPopMenu, kAcceptSystem

For a field which is not in a subwindow, kRangeSubwindow is equivalent to kRangeWindow. The drag range is ignored when the drag mode is kDragObject since a field can be moved only within its own window or subwindow.

Drag and Drop Events

Having set the drag and drop properties of fields and/or windows, you need to write event handling methods for these objects to handle events when dragging and dropping occurs. Drag and drop actions generate four events, in the order

– evDrag
the mouse is held down in a field and a drag operation is about to start, the event parameters are: pEventCode, pDragType, pDragValue. It is sent to the field being dragged.

– evCanDrop
a drag operation has started to test whether the field or window containing the mouse can accept a drop, the event parameters are: pEventCode, pDragType, pDragValue, pDragField. It is sent to the field might receive the drop.

– evWillDrop
the mouse is released at the end of a drag operation, the event parameters are: pEventCode, pDragType, pDragValue, pDropField. It is sent to the field being dragged.

– evDrop
the mouse is released over the destination field or window at the end of a drag operation, the event parameters are: pEventCode, pDragType, pDragValue, pDragField. It is sent to the field being dropped on.

If a field can accept the object or data that you are currently dragging onto it, it will become highlighted and the appropriate event messages are sent to the field. Depending on the drag and drop mode, evDrag and evWillDrop are both sent to the dragged field, and evDrop is sent to the drop field. For the kDragObject and kDragDuplicate modes, the move or duplicate is not performed if you discard the evDrop message.

The event parameters are

– pDragType
the drag mode of the field being dragged

– pDragField
a reference to the field being dragged

– pDragValue
the object or data being dragged: text, numbers, list data, and so on

– pDropField
a reference to the destination field

All the drag and drop events supply pDragType and pDragValue event parameters. Initially pDragType contains the drag mode of the field being dragged, but you can change it in any of your event handlers. If pDragType is changed by evDrag, the subsequent evCanDrop,

evWillDrop and evDrop will see the changed value, but changing it does not affect the drag mode. To avoid confusion with built-in drag operations it is recommended that your drag types are all greater than 1000.

Using Drag and Drop

Consider a window in which the user selects a substring of text in one field fDrag and drags it onto another text field fDrop, which will then highlight the inserted string. First you must set the drag and drop mode of the fields, either in the Property Manager or using the notation

```
Do Win1.$objs.fDrag.$dragmode.$assign(kDragData)
Do Win1.$objs.fDrop.$dropmode.$assign(kAcceptEdit)
```

You must also set up a variable name in the **dataname** property for each field, perhaps String1 and String2.

You can trap the events in the field methods, but it may be more convenient to handle them all in the window $control() method, in the case when all the fields in the window have consistent drag and drop handling. The line

```
Quit event handler (Pass to next handler)
```

at the start of each field method will pass all events to the window $control() method. In the window $control() method, you can add event handlers to detect the drag and drop event messages evDrag, evDrop, evCanDrop, and evWillDrop, with the following structure.

```
; $control() method in window
On evDrag
   ; do this
On evDrop
   ; do this
On evCanDrop
   ; do that
On evWillDrop
   ; do the other
```

The evCanDrop event is sent frequently during a drag operation, so this handler must be short and efficient: it should *not* do anything to change the appearance of the user interface, such as displaying a message or opening or closing a window.

You could choose to handle evDrop only, which provides the value of the dragged substring in the parameter pDragValue, and ignore the other events. When the mouse is released over the fDrop field, triggering evDrop, you can use the *mouseover()* function to return the position of the mouse pointer in the text string. The $mouseevents library preference must be turned on for your library to send and receive mouse events. You can also use the string functions *con()* and *mid()* to insert the dragged string into the fDrop field at the right place. You can highlight the inserted substring using the properties $firstsel and $lastsel.

```
; $construct() method for the window
; declare variable Pos of type Number
On evDrop
   Calculate Pos as mouseover(kMCharpos)
   Calculate String2 as con(mid(String2,1,Pos),
           pDragValue,mid(String2,Pos+1,len(String2)-Pos))
   Do $cobj.$firstsel.$assign(Pos)
   Do $cobj.$lastsel.$assign(Pos+len(pDragValue))
   Redraw (Refresh now) {fDrop}
```

Another frequent use of drag and drop is moving selected lines between lists. Consider two fields lDrag and lDrop that use the list variables List1 and List2. lDrag should have its $multipleselect property set and a drag mode of kDragData, and lDrop should have its $dropmode property set to kAcceptList.

When the drop occurs, pDragValue has a copy of List1 and not just the selected lines: these can be merged with List2 and removed from List1.

```
; $control() method for the window
; declare variable iList of type List
On evDrop
   Set search as calculation #LSEL
   Set current list List2
   Calculate iList as pDragValue
   Merge list iList (Use search)
   Set current list List1
   For each line in list (Selected lines only) from 1 to #LN step 1
      Delete line in list        ;; remove dragged lines
   End For
   Redraw lists (All lists)
```

Having merged the dragged lines into List2, you can sort the list.

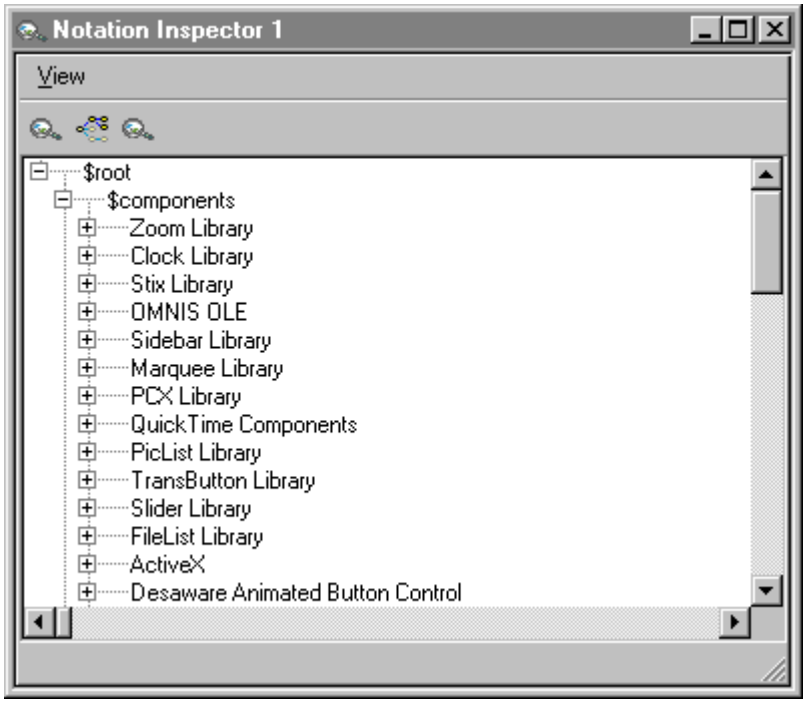
Alternatively, you might wish to insert the line or lines at a specific place in List2. In this case, you need to use the *Insert line in list* command to insert each required line at the mouse pointer position in List2 using the *mouseover(kMLine*) function

```
Set current list List1
Calculate InsertPoint as mouseover(kMLine)
For each line in list (Selected lines only,Descending) from 1 to #LN
   step 1  ;; descending order, so as to insert in ascending order
   Load from list
   Delete line in list
   Set current list List2
   Insert line in list {InsertPoint}
   Set current list List1
End For
Redraw lists (All lists)
```

A drag mode of kDragObject can be useful to give users the chance to rearrange fields on a window.

External Component Notation

The $components group under $root contains all the installed external components available in your XCOMP folder. You can view the contents of the $components group using the Notation Inspector.



Note that you manipulate an external component via its custom field properties, as shown above, not via the $root.$components...$compprops or $compmethods groups for the control. The groups under $root.$components is simply a convenient way of viewing the contents and functions of any external library or control.

The $components group has the standard group properties and methods, including $add() and $remove(), and you can list the components using the $makelist() method.

```
; declare variable cCompList of type List
Do $root.$components.$makelist($ref.$name) Returns cCompList
```

You can drag a reference to any of the components from the Notation Inspector to your code, in the same way as other built-in objects. You can click on a component library in the Notation Inspector and view its properties in the Property Manager. Each component library has the following properties

– $name
the name of the component which must be unique

– $pathname
the name and path of the external library file; this will vary across different platforms

– $functionname
the name of the external function

– $controlhandler
Boolean that indicates whether the external is a control handler, for example, an ActiveX is a control handler

– $constprefix
String used as a prefix for all constants within the external

– $flags
indicates the external flags, for example, whether it is loaded

– $usage
Current number of controls that are using this external

– $version
the version information

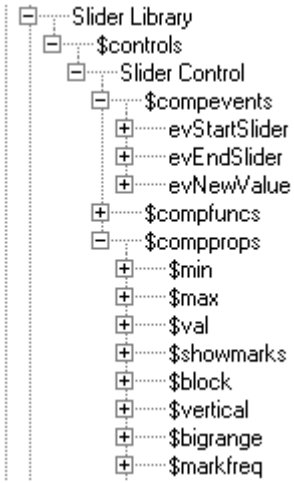
You can view the contents of an external library in the Notation Inspector. Each component library has a group called $controls containing all the controls in the library. Some libraries may contain only one control, for example, the Slider Component Library contains the Slider Control only. A control contains its own events, functions (or methods), and properties in their own respective groups, as follows

– $compevents
group of events for the control

– $comprops
group of properties for the control

– $compmethods
group of methods for the control

For example, the Slider Control has the following events and properties



In the notation you treat an external component property or function as you would a standard built-in property or method, that is, you can use property and method names in the notation to manipulate and send messages to an external component field. Note that property and method names should include a dollar sign when you use them in the notation.

```
Do $cwind.$objs.ClockField.$facecolor.$assign(kBlue)
; assigns a color to the face of a clock component
; using the $facecolor property
Do $cwind.$objs.QTfield.$Play()
; executes the $Play() function for a QuickTime component
```

In general, the properties of an external component are unique to the object and their names will not clash with standard OMNIS field properties. However when an external component property has the same name as an OMNIS property, you must access the external property using a double colon (::) before its name. For example, the Icon Field control has the property $backcolor which you must refer to using

```
Do $cinst.$objs.iconfield.$::backcolor.$assign(kRed)
; would not work without the ::
```

At runtime you can add an external component to an open window using the $add() method. You need to specify the kComponent object type, external library name, external control name, and the position of the field. For example, the following method adds the Marquee

Control to the current window instance, positions the new object, and sets some of its properties

```
; declare local variable Objref of type item reference
Do $cinst.$objs.$add(kComponent,'Marquee Library','Marquee
   Control',0,0,15,100) Returns Objref
Do Objref.$edgefloat.$assign(kEFposnStatusBar)
; repositions the object at the bottom of your window
Do Objref.$message.$assign('I hope you can read quickly!')
Do Objref.$steps.$assign(20)    ;; number of pixels to step
Do Objref.$speed.$assign(20)    ;; lower number is faster
Do Objref.$::textcolor.$assign(kBlue)   ;; note :: notation
Do Objref.$::backcolor.$assign(kRed)
```

Version Notation

All external components have the $version property. To get the version of an external component you must access it via the $root.$components group, not the external component field on a window or report. For example

```
Do $root.$components.Marquee Library.$version Returns lvXversion
; returns “1.2” for example
```

If you have created any external components of your own to run under OMNIS Studio version 1.x, you must recompile them for OMNIS Studio 2.0.

Java Beans

The Java Bean external component has commands that let you control it in a Runtime OMNIS. You request a command using the $cmd() method as follows:

```
$root.$components.JavaBean.$cmd(parameter list)
```

The parameters can be:

Parameter list	Command
"GetPaths", List	Populates the specified single column list with the Java Bean search paths; no return value
"AddPath", NewPath	Adds the specified path to the Java Bean search paths; returns true for success, or if the path is already present in the search paths
"DeletePath", DelPath	Deletes the specified path from the Java Bean search paths; returns true for success
"EnumBeans"	Enumerates Java Beans; returns the number of Beans found
"StartVM"	Starts the Java virtual machine (to test if Java is installed); returns a string containing an error, or an empty string to indicate success
"SetupDialog".	Opens the Java Bean component setup dialog
"RequestPath"	Opens the “Prompt for Java Bean Path” dialog; returns a string containing the new path; empty if none selected

HWND Notation

All window instances and their objects, except background objects, have the $hwnd and $framehwnd properties. In addition, window instances have the $toplevelhwnd property. These three properties all identify child windows or parts of a window. Each window object has a $framehwnd, which is the outermost enclosing child window of the object. Each window object also has an $hwnd, which is the child window which typically contains the main information displayed by the object. $hwnd is always contained in $framehwnd, and in many cases $hwnd and $framehwnd are the same child window.

For example, in the following field $hwnd is not the same as $framehwnd: $hwnd refers to the client window excluding the title window, and $framehwnd refers to the frame window.

For a window instance, $toplevelhwnd is the outermost enclosing child window of the window instance, that is, it corresponds to the complete window, including title bar and sizing border, if present. $framehwnd of an open window instance is the window contained in the $toplevelhwnd; it excludes the window title bar and sizing borders. $hwnd of an open window instance is contained in $framehwnd, together with the window menu bar, toolbar and status bar, if present. For example, when you use the notation

```
Do $cwind.$hwnd Returns lvNumber
```

$hwnd, $framehwnd, and $toplevelhwnd return a number. The number is a unique identifier that represents the child window.

$hwnd, $framehwnd, and $toplevelhwnd can also return an item reference to the child window, for example

```
Set reference myRef to $cinst.$hwnd.$ref
```

For $hwnd and $framehwnd, the item reference supports the following properties: $left, $top, $width, $height, $clientleft, $clienttop, $clientwidth, and $clientheight. $toplevelhwnd supports the following properties: $left, $top, $width and $height.

A child window can have a client area and a non-client area. The non-client area of the window can contain features such as the window border and scroll bars. Sometimes the non-client area is empty, such as in a borderless entry field with no scroll bars. Consider an entry field with a 2 pixel inset border. Its client area sits inside the non-client area, as follows.

The entry field with a 2 pixel inset border may be 100 pixels wide and 50 pixels high. The client area would be 96 pixels wide and 46 pixels high. Therefore

```
$width = 100    $clientwidth = 96
$height = 50    $clientheight = 46
```

$top and $left are the coordinates of the child window, relative to the child window which contains it (in the case of $toplevelhwnd, these coordinates are relative to the area in which OMNIS displays window instances).

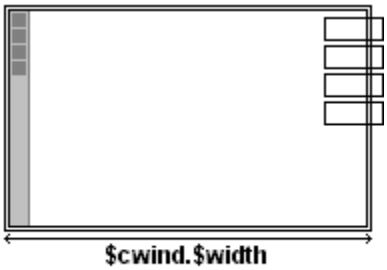
$clientleft and $clienttop are the coordinates of the client area of the child window. These always return the value zero.

The properties of $hwnd and $framehwnd are not assignable. For $toplevelhwnd, $left, $top, $width and $height are assignable.

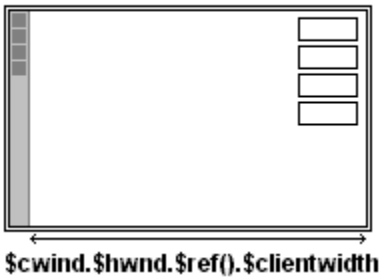
Using HWND

Sometimes it is important to know the exact size of the client area. For example, if a window has a toolbar on the left and you want to create controls right-justified down the right edge f the window, $cwind.$width would not be good enough, as it includes the width of the toolbar. This would cause you to add controls (using $add) too far to the right.

In this first example you call $add() to add objects and the $left for the objects would be the width of the window less the $width of the objects you are adding. As $width included the toolbar space, the objects would be added too far to the right.



In this second example, using the $hwnd of the window, you get the exact width excluding the toolbar width, allowing you to right justify the controls correctly.



Enter Data Mode

You can place window instances in enter data mode on a case-by-case basis. A window instance has the $enterable attribute; if true, the window is in enterable mode. Enterable means the window is in ‘enter data mode’ so data can be typed into the entry fields and any OK and Cancel buttons are enabled. Normally windows with modeless enter data are always enterable and other windows are enterable when there is an executing *Enter data* command on the method stack. When you set $enterable to kTrue for a window instance it is never changed automatically by OMNIS at an *Enter data* command, therefore

```
Do $cinst.$enterable.$assign(kTrue)
```

in the $construct() method of the window is equivalent to putting the window in modeless enter data mode.

It is possible to go into enter data mode without the top window (or any window) being enterable. Sometimes this might be desirable, but beware OMNIS provides no protection against this situation.

If a window is not enterable for Enter data it is also not enterable for Prompted find.

Floating edges for Windows

Window classes and instances have the $edgefloat property. Therefore the edges of a window can float using any of the floating edge constants, except the kEFposn… values. When the size of the area available to window instances changes, that is, the OMNIS application window is sized, open window instances and window classes float according to the value of their $edgefloat property. The default value of $edgefloat is kEFnone.

For example, in conjunction with some code in the $construct() method of a window, you can use $edgefloat to attach a window to the right-hand side of the main OMNIS window (or desktop under MacOS). Set $edgefloat to kEFleftRightBottom, and use the following code in the $construct() methods of the window.

```
Set reference item to $cinst.$toplevelhwnd.$ref
Calculate item.$left as $root.$modes.$width – item.$width
Calculate item.$top as 0
Calculate item.$height as $root.$modes.$height
```

If you resize the OMNIS window (under Windows), or change the monitor resolution (MacOS), the window remains attached to the right-hand side.

Lookup Windows

The *Lists and Grids* chapter in the Using OMNIS Studio manual describes how to build a list, display it in a List Box field, and use evDoubleClick to transfer data from the selected list line to the Current Record Buffer using the *Load from list* command. However there may be limited space on your window for a large list field, therefore you might want to place the list on a separate *lookup window*. You can force such a window to open when the user needs to look up the data, and close it as soon as a line is chosen. As a further refinement you can allow the user to enter some data directly into a field and not popup the window, or if the field is left empty popup your lookup window containing a list of possible choices.

An entry window wBookings for the BOOKINGS file, for example, might have the foreign key BkCuId to the primary key CuId in the CUSTOMERS file. The BkCuId field method will check if the code entered by the user is valid and allow the user to choose from a list of

customer names if it is not. The important point here is that method execution must be held up until the user has made a choice. These are the methods to implement this: they will be described together since they interact.

```
; $event() method for wBookings field BkCuId
On evAfter
   Do CheckCustCode Returns Valid ;; check on code entered
   If not(Valid)                  ;; if invalid or no code
      Open window instance {wCustList}   ;; open lookup window
      Enter data                         ;; until item is selected
      Close window instance wCustList
      Calculate BkCuId as CuId           ;; set foreign key ..
      Redraw { BkCuId }                  ;; and show value
      Queue set current field {BkCuId}   ;; reposition cursor ..
      Queue tab                          ;; to next field
   End If
   Quit event handler (Discard event)
```

The Customers List window wCustList is a Simple or NoFrame style window filled by a list box. The list is defined and built as the window opens. There's no need to show CuId in the list box but it must be in the list.

```
; $construct() method for wCustList
Set current list cCustList
Define list {CuId, CuLname, CuCountry}
.. build list from OMNIS or SQL data

; $event() method for list box field
On evDoubleClick     ;; Event Parameters -  pRow ( Itemreference )
   Load from list    ;; transfer list line values to CRB
   Queue cancel      ;; terminate enter data mode
   Quit event handler (Discard event)
```

When wCustList opens, the list is built. At this point *Enter data* is necessary to halt execution of the method until the user has chosen from the list. When the list box receives a double-click, *Load from list* transfers the list line data to the CRB. *Queue cancel* now terminates the enter data state so that execution resumes and closes the window. BkCuId is set from CuId entered from the list and the field is redrawn. The cursor will still be in BkCuId so *Queue set current field* and *Queue tab* can be added to place the cursor at the next field in the tabbing order.

This enter data state is needed whether or not the parent window has $modelessdata set.

Timer Methods and Splash Screens

A splash screen makes a more friendly introduction to an application than presenting the user with a blank screen and a menu bar. It involves opening an introductory window, usually called wAbout, from the $construct() method in the startup task of your library. You can keep the About window on screen either for a predetermined time or until the user clicks on it. The wAbout window contains a button area field to detect clicks and the following methods

```
; $construct() method for wAbout
Set timer method (8 sec) {Timer Control}

; $event() method for the button area field
On evClick
   Do method Close Window

; Close Window method
Clear timer method
Close window {wAbout}

; Timer Control method
Do method Close Window
```

When the $construct() is called, the *Set timer method* command sets a time delay in seconds and nominates a method, called Timer Control, which is run at the end of the delay. The Timer control method then calls Close Window which clears the timer and closes the window. If the window has been clicked on before the end of the delay, the button area method calls the Close Window method which closes the window immediately.

The button area should have its $noflash property set to kTrue to avoid flashing when the user clicks on it.

Pictures

You can place button area fields over a graphic, which you can paste onto your window or load into a Picture field. If the library preference $sharedpictures is set, pictures are converted to a format accepted under both Windows and MacOS. To optimize the drawing of a picture so that the system palette matches the picture colors on the Windows platforms, add the following line to the window $construct() method.

```
Set palette when drawing (Color shared pictures)
```

Chapter 8—Internet Programming

The client/server model, and hence your OMNIS applications, readily translate to the Internet. OMNIS Studio includes a number of wizards in the Component Store to help you get started with Internet access, and these are described in the *Using OMNIS Studio* manual. This chapter goes deeper into the programming aspects of Internet access.

The key facts about the Internet are self-evident; cross-platform compatibility, near-universal access, cheap bandwidth, simple and well-disseminated standards for all.

OMNIS Studio supports high-level application-layer services, such as FTP, E-mail (POP3 and SMTP), and HTTP, as well as low-level communication protocols such as TCP/IP sockets for operations like peer-to-peer information exchange and HTTP access through proxy servers that redirect URLs. OMNIS provides a full-featured set of external commands for enabling your OMNIS application for the Internet and World Wide Web.

The Internet is built on standards, therefore you should always refer to and understand those standards, which are readily accessible. So before you start to web-enable your OMNIS application, you should learn about HTML and query forms, about HTTP headers, server responses, and methods of requesting information from HTTP servers (GET, POST, and HEAD). You should also become familiar with CGI programming which is essential to Web database applications.

Internet Protocols

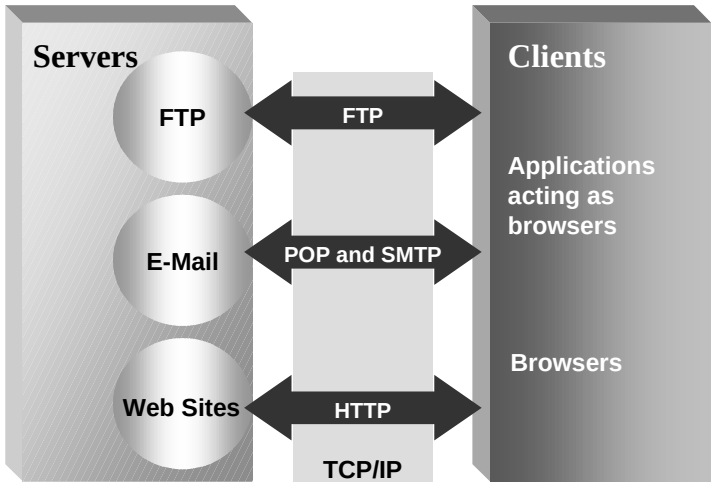
This section discusses the protocols used for network communications, e-mail, and file transfer over the Internet, and takes a closer look at HTTP on the Web, including an explanation of server names and addresses and the role of CGIs. You can get more detailed descriptions and definitions for Internet protocols from the Internet itself.

Overview

The Internet is the largest public network, a global network of networks. All of these interconnected networks rely on a standard underlying network-layer protocol called TCP/IP (Transmission Control Protocol/Internet Protocol) to move information. The most important feature of this low-level protocol is its platform independence. TCP/IP allows Windows Intel-type, Macintosh, Unix, and other computers to exchange information using the client/server Internet model.

In the TCP world, a connection is made using a *socket*, a dynamically generated reference to network layer resources necessary for a conversation between a client and server.

The Internet client/server model also uses higher-level standardized protocols to enable data transfer of different types. Servers and their clients use special-purpose applications protocols, such as FTP (File Transfer Protocol), that sit on top of TCP/IP as it handles network communications. While the FTP protocol is used to download and upload files from and to FTP servers, e-mail servers often use two protocols, SMTP (Simple Mail Transport Protocol) and POP3 (Post Office Protocol, version 3). SMTP handles sending mail, and POP3 retrieves mail. HTTP (Hypertext Transport Protocol) is the applications protocol used to transfer information among World Wide Web servers and client.



Note for MacOS users

OMNIS supports Macintosh Open Transport 1.1.1 and above. Open Transport replaces the current MacOS implementations of AppleTalk and TCP/IP (including the protocols and the Network, MacTCP, and Admin TCP control panels). Open Transport is also designed to replace the Connection Manager and the Communications Resource Manager of the current Communications Toolbox architecture.

HTTP and HTML

People often use the term *World Wide Web* loosely to refer to the whole Internet. More precisely, however, it encompasses Web servers that use HTTP (Hypertext Transport Protocol) to transfer information specified by *URLs* (Uniform or Universal Resource Locators, pronounced “earl”). HTTP defines the formats and transmission methods of interactions and how servers and clients respond to commands. When you point your browser to an URL, an HTTP command goes to the specified server and tells it to find and return the resource you need on the server.

HTTP servers normally listen for requests on Port 80, so a port number is not needed in a client request unless the server listens on a port other than 80.

The Common Gateway Interface (CGI) is a convention that specifies how gateway programs or scripts are integrated. Gateway programs or *CGIs* allow HTTP servers to handle queries or forms requests from clients; they operate outside the HTTP protocol. CGIs are frequently used to allow an HTTP server to interact with a database to store or retrieve information in response to a client request.

HTTP servers accept queries. There are two methods for querying HTTP servers:

– A GET usually requests the path to an HTML. This type of request generally can return a maximum of 1,024 characters and is often used to perform “canned”, repetitive queries.

– A POST usually includes field information from a form and is more flexible.

A server’s response includes a MIME (Multipart Internet Mail External) or MIME-like header, which specifies the type of data contained in the body of the response.

In the Web environment, *client* is often synonymous with *browser*. However, a client need not be exclusively a browser (as is Netscape Navigator, for example). OMNIS can use commands to act as a client and retrieve information from a server.

Web browsers also provide access to servers running protocols other than HTTP, such as SMTP and FTP.

Unlike FTP servers, which remember the location of each client in the file system, HTTP servers do not track client activities. HTTP has no state—each connection to a Web page is made, closed, and forgotten—so every request must fully specify the desired resource.

An application can also use CGIs to track client interactions with an HTTP server, overcoming some of the “statelessness” of HTTP (see the section on Caching for more information about tracking HTTP sessions).

HTML provides a standard, text-based method of tagging content to specify the layout of a Web page, as well as hyperlinks to resources (graphics, for example) and embedded directives to the browser. The language is simple and straightforward, much like early word-processing programs.

HTML version 3.2 and earlier is designed for portability to every type of graphical interface and operating system. For this reason, HTML specifies some styles such as font attributes (for example, <B> and </B> delimits bold text) and general size and positioning relationships. However, other formatting aspects, margins for example, are left up to the browsers displaying the page. Most Web browsers allow easy viewing of the HTML source for a page, so that you can easily see how HTML translates into a Web page.

Internet Commands

The OMNIS Internet commands, located in the method editor in the *External Commands… g*roup, simplify coding for Internet and Web-based applications. Each Internet command has a prefix to identify its command group. For example, the command to send a message on a socket is *TCPSend*. To locate a command of any particular type you can type the prefix and then scroll the command list in the method editor to find the command you want. See the OMNIS Help for a complete description and the syntax for each command.

FTP Commands

The File Transfer Protocol (FTP) commands let you download and upload files from and to remote computers. Finding a file requires navigation of the directory/file structure on the remote machine. On many servers, Unix file commands perform this activity. However, the Internet commands let the user locate files using local OMNIS navigation screens that you create. Alternatively, the application may specify exactly which file to download.

HTTP World Wide Web Commands

The HTTP (HyperText Transport Protocol) commands let you access Web content without a detailed knowledge of the Web itself. For example *HTTPPage* retrieves the HTML text of a Web page specified by an URL, *HTTPSplitHTML* parses HTML from a Web page into an OMNIS list, and *HTTPSplitURL* splits a full URL into a host name and path.

E-mail Commands

The e-mail commands let you send and receive Internet e-mail using the popular Internet SMTP and POP3 protocols.

TCP/IP Sockets, DNS, and Ping Commands

The TCP commands let you manipulate TCP/IP sockets using equivalents to the Berkeley Sockets Interface. In addition, the Internet commands provide an OMNIS application with commands for Domain Name Services (DNS). These commands allow a client or server to find information about a peer computer to which a socket is connected and perform general DNS operations:

The Internet commands let you send an Internet Control Message Protocol (ICMP) Echo (also called *Ping*) message to a named computer. Intercomputer transfer times can vary widely and for that reason, the *TCPPing* command is useful when timing certain intercomputer transfers. Sending a ping to a distant computer determines whether a time-out might close the socket or otherwise interfere with downloading a file.

Utility Commands

The Utility commands include Common Gateway Interface (CGI) encoding and decoding, Binary file handling, and error handling.

The CGI standard establishes rules for running external programs on a Web HTTP server and returning the results to the requestor. CGI encoding formats data so that all characters may be transferred over the Internet and not be confused with special characters such as field separators. These extended OMNIS commands allow an application to encode and decode data from HTML forms and URLs.

Many of the HTTP commands automatically encode and decode CGI information, as well as performing other common operations (that is, POST/GET, parsing an HTTP header, and so on).

Information is frequently sent on the Internet in a format known as *UUencoded*. This format preserves binary characters, primarily in e-mail, from accidental changes by transfers on the Internet. This type of encoding prevents some servers from, for example, interpreting and altering characters or imposing different byte-ordering on binary data.

UUencoding is a means to represent binary information as common printable ASCII characters. It is often used at FTP sites.

The *WriteBinFile* and *ReadBinFile* commands let you read and write binary data forks (Macintosh) or files (PC). UUEncoding provides a convenient method of encoding before transmitting this binary information.

The *WebDevError* command lets you specify an OMNIS method to handle error that occur in the HTTP, e-mail, FTP, and sockets commands. The *FTPGetLastStatus* command also issues error messages when FTP commands return errors.

Sending and Receiving E-mail

This section presents some OMNIS methods that demonstrate the Internet e-mail commands. To try out any of the sample code, change the initial values to suit your own requirements including server names, account names, passwords, and so on.

Sending E-mail

This example demonstrates how to use the *SMTPSend* command to send e-mail to yourself. To try out this example, you must have access to an SMTP mail server, though it need not be the SMTP server that is hosting the accounts. As long as the account information is correct, any SMTP mail server forwards the e-mail to the appropriate server.

```
; Declare the following local variables
; LV_SERVER_NAME (Char) = 'smtp.domain.com'
; contains name of SMTP mail server
; LV_ACCOUNT_NAME (Char) = 'me@domain.com'
; contains personal e-mail account name
; LV_TO_ACCOUNT (Char), LV_CC_ACCOUNT (Char), LV_BCC_ACCOUNT (Char)
; LV_CALLBACK_METHOD (Char) = 'method name'
; LV_PRIORITY (Short Int)  ;; 1 is high, 5 is low priority
SMTPSend (LV_SERVER_NAME,LV_ACCOUNT_NAME,LV_ACCOUNT_NAME,
   'Hello world test', 'This is the body of the message'
   ,LV_CC_ACCOUNT, LV_BCC_ACCOUNT, "My Full Name",
   LV_CALLBACK_METHOD, LV_PRIORITY)
```

Replace the default values with real values and run this example. OMNIS pauses as the e-mail is going to the server. If you specify a callback method, the command calls that method at least 11 times throughout the process to let you know that the e-mail was sent. If the number of CC and BCC recipients is 0 (zero), then it is called 11 times. If you include any CC and/or BCC recipients then:

11 + (<number of CC recipients> + 1) = Number of times called

and/or

11 + (<number of BCC recipients> + 1)= Number of times called

If you use the command to send a message with three CC: recipients and two BCC: recipients, the total would be 18, as follows:

11 + (3 + 1) + (2 + 1) = 18

Finding Out How Many Messages are Waiting

The following method uses the *POP3Stat* command to retrieve the number of e-mail messages that are waiting on the server for a specified e-mail account. It does not change the e-mail messages in any way.

```
; Declare the following local variables
; LV_SERVER_NAME (Char) = 'smtp.domain.com'
; contains name of SMTP mail server
; LV_ACCOUNT_NAME (Char) = 'me'
; LV_PASSWORD (Char) = 'secret'
; Checking the number of messages that are waiting on the server
POP3Stat (LV_SERVER_NAME,LV_ACCOUNT_NAME,LV_PASSWORD)
```

Receiving E-mail

The following method demonstrates how to retrieve e-mail messages for a specified account. The *POP3Recv* command requires that you have access to the POP3 server that contains the e-mail messages for the account.

```
; Declare the following local variables
; LV_SERVER_NAME (Character   10000000) = 'smtp.domain.com'
; contains name of mail server
; LV_ACCOUNT_NAME (Char) = 'me'
; LV_PASSWORD (Char) = 'secret'
; LV_EMAIL_LIST (List)
; contains  single column list of entire e-mail with header info
; LV_CALLBACK_METHOD (Char) = 'method name'
; LV_EMAIL_MESSAGE (Char)
; contains entire e-mail with header
; LV_DELETE_EMAILS (Boolean) = kTrue
; determines whether or not the e-mail should be deleted
Begin reversible block
   Set current list LV_EMAIL_LIST
End reversible block
Define list {LV_EMAIL_MESSAGE}
; Retrieving e-mail from POP3 server
POP3Recv(LV_SERVER_NAME, LV_ACCOUNT_NAME, LV_PASSWORD,
   LV_EMAIL_LIST, LV_DELETE_EMAILS, LV_CALLBACK_METHOD)
```

Parsing E-mail Headers

MAILSplit parses the headers from an e-mail message into a two-column list. Without MAILSplit, this process would require a substantial amount of OMNIS code.

```
; Declare the following local variables
; LV_EMAIL_HEADER_LIST (List)
; contains 2-column list of header values
; LV_FIELD (Char), and LV_VALUE (Char)
; first and second cols in the e-mail header list
; LV_EMAIL_MESSAGE (Char)
; contains entire e-mail with header
; LV_BODY (Char)
; contains just the body of the e-mail
MailSplit (LV_EMAIL_MESSAGE,LV_EMAIL_HEADER_LIST,LV_BODY)
```

Additional Information

The common hostname/port for SMTP servers is:
smtp, port 25

The common hostname/port for POP3 servers is:
POP3, port 110

Windows Users

If you have upgraded to Windows NT from Windows 3.x or Windows 95, you may have several Services files in orphaned directories. POP3Stat may try to access one of these unused files, which might not contain the valid port address a particular service. If this happens, you receive the following error messages:

```
getservbyname() failed
```

followed by, for example,

```
POP3Stat: can't connect, invalid socket from connect TCP
```

You should delete any extraneous Services files. In Windows, look in the residual windows\system directories.

The Services file should have the following entries:

```
     smtp mail    25/tcp       mail
     pop3         110/tcp      Pop-3   postoffice
or
     smtp         25/tcp       mail
     pop3         110/tcp      postoffice
     pop-3        110/tcp
```

Working with FTP Sites

This section presents sample code for retrieving information from an FTP server. It is a simple and brief introduction to the use of FTP commands.

First you need to connect to an FTP site and check for errors. Use the commands *FTPConnect* and *FTPGetLastStatus*.

```
; Declare local vars fFtpSiteUrl,fFtpUser,fFtpPassword all (Char)
Calculate fFtpSiteUrl as 'ftp.omnis-software.com'
Calculate fFtpUser as 'anonymous'
Calculate fFtpPassword as 'your_mailid@your system'
Working message (High position,Large size) {Connecting to
   [fFtpSiteUrl]}
FTPConnect (fFtpSiteUrl,fFtpUser,fFtpPassword) Returns fFtpSocket
If fFtpSocket<0     ;; an error
   FTPGetLastStatus (fFtpSocket) Returns fStatus
   OK message (High position,Large size)
        {FTPConnect failed, status is   [fStatus]}
End If
Close working message
```

Next locate the desired directory on the FTP site. Get the name of the remote server's current directory using *FTPPwd*.

```
FTPPwd (fFtpSocket) Returns fFtpCurrentDirectory
```

Get an OMNIS list of file information in the current directory on the remote server using *FTPList*.

```
Begin reversible block
   Set current list fDirectoryList
End reversible block
Clear list
Working message (High position) {Getting Directory Info}
FTPList (fFtpSocket,fDirList,fFtpCurrDir,0) Returns fStatus
If fStatus<>0
   OK message (High position) {FTPList failed, status is [fStatus]}
End If
Close working message
Do method FixFileNameList (fDirectoryList)
Calculate #L as 0
Redraw fDirectoryList
```

Next you change the working directory on the connected FTP server. The working directory is the one for which the *FTPList* command shows a directory listing. Files are transferred to and from the remote working directory using *FTPCwd*.

```
FTPCwd (fFtpSocket,FtpNewDirectory) Returns fStatus
If fStatus<>0
   OK message (High position) {FTPCwd failed, status is [fStatus]}
End If
```

Download files from the FTP site. Specify that the next file transfer on this socket is of either ASCII or binary type using *FTPType*.

```
FTPType (fFtpSocket,fFileMode) Returns fStatus
If fStatus<>0
OK message (High position,Large size) {FTPType failed, status is
   [fStatus]}
End If
```

Download either a text file or a binary file from an FTP site using *FTPGet*.

```
PutFile (lSaveAsFileName,'Save file as:','',fTextFileName) Returns
   fStatus
If len(lSaveAsFileName)
   Working message (High position) {Downloading: [fTextFileName]}
   FTPGet (fFtpSocket) Returns fStatus
   If fStatus<>0
      OK message (High position) {FTPGet failed, status is [fStatus]}
   End If
   Close working message
End If
```

Download a binary file into a Binary field in OMNIS using *FTPGetBinary*.

```
Working message (High position) {Downloading: [fBinaryFileName]}
FTPGetBinary (fFtpSocket,fBinaryFileName,fPicture) Returns fStatus
If fStatus<>0
   OK message (High position) {FTPGet failed, status is [fStatus]}
Else
        Redraw named fields fPicture to fPicture
End If
Close working message
```

Disconnect from the FTP site using *FTPDisconnect*.

```
FTPDisconnect (fFtpSocket) Returns fStatus
```

Additional Information

The common hostname/port for FTP servers is:
ftp, port 21

When using *FTPList*, keep in mind that there are several standards among FTP protocols (Unix, NT, and so on) for displaying directories. You may have to parse out some contents, for example, read/write information, so that users don't see it. If you are dealing with a Unix FTP server, try the Common Code in the m_cc_ftp menu class to parse automatically.

Some FTP daemons accept system-specific directory path formats, that is, Macintosh colon-separated as in Macintosh HD:My Folder:My File or VMS-style path and file specifications as in

```
SOME$DISK:[USER.SUBDIRECTORY]FILENAME.EXTENSION;1.
```

Consult the documentation for the server to determine the acceptable directory path specifications. When in doubt, try the Unix style.

The *FTPChmod* command lets you change the permissions for a file on an FTP server. If you've never done this before, consider using the supplied Common Code example window w_cc_permissions to set the correct string value for this call.

Note to Windows users

FTP operations require the following entry in your Windows Services file:

```
ftp     21/tcp
```

If you experience problems connecting to an FTP server, consult this file.

Working with HTTP Servers and Clients

This section presents sample code for HTTP servers and clients.

HTTP servers and clients use HTTP (Hypertext Transport Protocol) to transfer information specified by URLs. CGIs allow HTTP servers to interact with databases.

The headers in HTTP server responses contain information about what kind of data is being returned to a client. In the case of an HTTP call that is sending HTML files, for example, the header specifies content-type:text/html. This tells the receiving application that an HTML file is being sent.

The format of the headers in HTTP is exactly the same as that in e-mail protocols. The recipient uses header information to determine what to do with the file when it is received. In the case of HTML files, the browser first displays the text, then opens separate channels for each additional file that is referenced by the HTML page. For example, a Web browser might receive an image, decode the binary information, and display it in the appropriate area of an HTML document.

One of the best reasons for programming HTTP applications in OMNIS is that you can create server applications to respond to requests from your end users' Web browsers.

HTTP Server Commands

This example uses the *HTTPServer* command to listen for incoming client requests and return "Hello world" regardless of the request.

First the method that starts listening.

```
StartListeningMethod
--------------------
; Declare local variable vStatus (Long integer)
; Declare class variable fPort (Long integer) = 80
; Declare class variable fListeningSocket (Long integer)
TCPSocket  Returns fListeningSocket
If fListeningSocket<0
   Quit method (flag clear)
End If
```

Creates a socket and checks for an error.

```
TCPBind (fListeningSocket,fPort) Returns vStatus
If vStatus<>0
   HTTPClose (fListeningSocket) Returns vStatus
   Quit method (flag clear)
End If
```

Binds the socket to a particular port and checks the status.

```
TCPListen (fListeningSocket) Returns vStatus
If vStatus<0
   HTTPClose (fListeningSocket) Returns vStatus
   Quit method (flag clear)
End If
```

Puts the socket in listening mode.

```
Set timer method 1 sec 5 {Listen}
Quit method (flag set)
```

Calls the Listen method every five seconds.

```
Listen
------
; Declare the following local variables:
; vWorkSocket (Long int0), vClientRequest (Char)
; vReadStatus (Long int), vStatus (Long integer)
; vAmountSent (Long int), vResponse (Char)
; vResponseStatus (Long int), vResponseHeader (List)
; vName (Character 1024), vValue (Character 1024)
; vContentLength (Long int)
TCPAccept (fListeningSocket) Returns vWorkSocket
```

Accepts an incoming connection on the port and returns the newly assigned socket number.

```
If vWorkSocket=-1    ;; -1 is error
   HTTPClose (fListeningSocket) Returns vStatus
   Quit method (flag clear)
Else If vWorkSocket<>-10035
   HTTPRead (vWorkSocket,vClientRequest) Returns vReadStatus
   If vReadStatus<0
      HTTPClose (vWorkSocket) Returns vStatus
      Quit method (flag clear)
   End If
   Calculate vResponse as '<HTML><BODY><H2>Hello,
   world!</H2></BODY></HTML>'     ;; Prepares response
   Calculate vResponseStatus as 200
   Set current list vResponseHeader
   Define list {vName,vValue}
   Calculate vContentLength as len(vResponse)
   Add line to list {('Content-type','text/html')}
   Add line to list {('Content-length',vContentLength)}
   HTTPHeader (vWorkSocket,vResponseStatus,vResponseHeader)
   HTTPSend (vWorkSocket,vResponse) Returns vAmountSent
   If vAmountSent<0
      HTTPClose (vWorkSocket) Returns vStatus
      Quit method (flag clear)
   End If
   HTTPClose (vWorkSocket) Returns vStatus
   If vStatus<0
      Quit method (flag clear)
   End If
End If
Quit method (flag set)
```

To stop listening you can call a stop listening method such as.

```
StopListening
-------------
HTTPClose (fListeningSocket)
Clear timer method
```

Accessing a Proxy Server

A proxy server protects an internal corporate network while allowing people inside the company to serve Web pages. Proxy servers translate outside requests for use internally so that outsiders do not gain access to information that an organization must keep secure. They also can increase performance by distributing messages among multiple servers.

– HTTPPage
  is a high-level command that allows you to grab a specified Web page transparently from a remote server using a specific URL. No additional coding is required.

– HTTPGet and TCPReceive
  are low-level tools for programming your application to handle communications through a proxy server.

When a URL is redirected via a proxy server, the HTTP server originally contacted will respond with a status code of 302, and a new URL to use as the value of the "Location:" header field. To honor this when using the HTTPGet/TCPReceive combination, the caller should loop through the redirections and replace the URL being gotten. Note that it is possible to go through multiple redirections.

```
Test HTTP Proxy
---------------
; Declare the following local variables:
; getHost (Char) = 'localHost'
; getURL (Char) = '/TryProxy'
; sock (Long integer), and message (Char)
; len (Long int), respCode (Long int), and newURL (Char)
While getURL<>""
   HTTPGet (getHost,getURL) Returns sock
   Calculate getHost as ""  ;; Empty the string
   Calculate getURL as ""   ;; Empty the string
   If sock>=0
      TCPReceive (sock,message) Returns len
      ; Check for the 302 status code, indicating a redirection
      Do method getHTTPResponseCode (message) Returns respCode
      If respCode=302
         Do method getHTTPProxyURL (message) Returns newURL
         ; Finds the value for the "Location:" field in the
         ; HTTP response header.
         If len(newURL)
            HTTPSplitURL (newURL,getHost,getURL)
         End If
      End If
      TCPClose (sock)
      End If
End While

getHTTPResponseCode
-------------------
; Declare parameter var HTTPResponseHeader (Field name)
; and Local variable lHTTPVersionString (Char) = 'HTTP/1.0'
Set return value
   mid(HTTPResponseHeader,len(lHTTPVersionString)+2,3)}
```

```
getHTTPProxyURL
---------------
; Declare parameter var pHTTPResponseHeader (Char)
; and Local variable lEOL (Char) = pick((sys(6)='W')+2*(sys(6)='U'),
   chr(13),chr(13,10),'eol for unix')
; lSearchStr (Char) = con(lEOL,'Location: ')
; lPos (Long Int) = pos(lSearchStr,pHTTPResponseHeader)
If lPos
   Calculate pHTTPResponseHeader as  mid(pHTTPResponseHeader,
   lPos+len(lSearchStr), len(pHTTPResponseHeader))
   Set return value {mid(pHTTPResponseHeader, 1,
   pos(lEOL,pHTTPResponseHeader)-1)}
Else
   Set return value {""}
End If
```

Submitting a CGI Request

OMNIS acting as an HTTP client is a powerful way to obtain information from the Web. To do this, you use HTTP commands.

A typical CGI request uses an HTML page that contains a form. This form is a list of fields with names and general datatype attributes. A form also includes an *ACTION* tag, which consists of a URL for the server that is accepting the CGI call for the form.

Some sample code, based on the US Postal Service site for accessing zip codes follows and illustrates some of the commands.

The U.S. Postal Service is located at http://www.usps.gov/. Use a Web browser to connect to this page on the server: http://www.usps.gov/ncsc/lookups/lookup_zip+4.html. The browser displays a form for getting zip codes.

To gather the information needed for the application, use the browser's built-in feature to view the HTML source for this Web page. Scroll down and find the HTML tag FORM. In this HTML, the tag looks like this:

```
<FORM METHOD="POST" ACTION="/cgi-bin/zip4/zip4inq">
```

This tag says that the POST method is used to send the zip code to the Internet server. This means that the *HTTPPost* command can request information from the server. And the CGI is /cgi-bin/zip4/zip4inq which the OMNIS application will call in its request.

Find the list of fields that the server is expecting in order to provide the requested information. The labels that describe the form's fill-in blanks and the field specifications are included in the form, but you only need the names:

```
<INPUT SIZE="30" MAXLENGTH="50" NAME="company">
<INPUT SIZE="30" MAXLENGTH="50" NAME="urbanization">
<INPUT SIZE="30" MAXLENGTH="50" NAME="street">
<INPUT SIZE="30" MAXLENGTH="50" NAME="lastline">
```

There are also two buttons in the HTML source for the form: <INPUT TYPE="submit" VALUE="Process Address"> and <INPUT TYPE="reset" VALUE="Clear the Form">. The SUBMIT button sends the filled-in information data to the server. The RESET button tells the HTML Web browser to reset the fields in the form to the default value. Note that, in the case of a SUBMIT button, the name of the button goes to the server when that button is pressed. In this case, the SUBMIT button sends the value Process Address.

Next, create a two-column list of fields and values to send to the server. The first column describes the field name, and the second column contains the value.

```
Do LV_CGI_LIST.$define(FIELDNAME,LV_FIELD_VALUE)
Do LV_CGI_LIST.$add("company",PV_COMPANY)
Do LV_CGI_LIST.$add("urbanization",PV_URBANIZATION)
Do LV_CGI_LIST.$add("street",PV_STREET)
Do LV_CGI_LIST.$add("lastline",PV_LASTLINE)
Do LV_CGI_LIST.$add("submit","Process Address")
```

When the list is complete, you can request information from the Postal Service's HTTP server, using the information collected from the HTML form using *HTTPPost*.

```
HTTPPost ("www.usps.gov","/cgi-bin/zip4/zip4inq",
LV_CGI_LIST) Returns LV_SOCKET_NO
```

To receive the information, only a few commands are needed. This loop gathers information from the Web server using *TCPReceive*.

```
Repeat TCPReceive(LV_SOCKET_NO,LV_RESPONSE) Returns
   LV_RESPONSE_LENGTH
```

Concatenate the results into a local variable LV_HTML. The return value of LV_RESPONSE_LENGTH monitors the amount of information that is coming back. When the response length drops to 0 (zero), the read is complete and you close the socket using *HTTPClose*.

```
Calculate LV_HTML as con(LV_HTML,LV_RESPONSE)
Until LV_RESPONSE_LENGTH<=0
HTTPClose (LV_SOCKET_NO)
```

Additional Information

Common hostname/port for HTTP servers is:
httpd, port 80.

Port 80 is the standard port on an HTTP server where clients connect. Once you have decided which port to use for listening on your server, you can accept HTTP connections by using the *HTTPServer* command. When a request arrives, HTTPServer opens a new socket and passes the socket number to the processing routine.

Common hostname/port for echo servers is:
echo, port 7.

To get started with the *HTTPServer* command, try the HTTPServer Common Code method in the m_cc_webserver class.

When you use HTTPPost or HTTPGet to specify a hostname URL, you do not need to include the protocol specification (http://).

When you use *HTTPParse* to obtain a URL, the leading slash makes a simple OMNIS equality string comparison to the name of the URL fail. Use the pos() function or similar parsing mechanism to find the URL name.

The trailing question mark of a GET-method CGI, which separates the URL path from the CGI arguments, is stripped by HTTPParse.

If you have used HTML already, you may have considered hard-coding all the HTML tags inside string-manipulating methods until you've created the desired page. However, this is a cumbersome method that often results in code duplication. Instead, try the Common Code class m_cc_htmlgen, which creates HTML for you. If you know HTML, it shouldn't take long for you to master it. If you don't know HTML, it can help you learn. The m_cc_htmlgen Common Code menu class works through four basic actions: *Initialize*, *Insert*, *Add*, and *Make*:

– *Init*(ialize) initializes a character variable with an HTML base page.

– *Insert* places HTML tags into the HTML page at the proper location.

– *Add* appends HTML tags to the end of a specified character variable.

Make encloses a character variable (text string) within the specified HTML tags, for example, to make an entire string bold.

The m_cc_webserver Common Code class includes an infrastructure to maintain a list of CGIs that you support. This system uses two methods to manage the CGIs. To add a CGI name to the list, use the Register CGI method. To remove one, use the Remove CGI method.

Once you have registered your CGI by supplying the method names to call for each CGI request, use a listening method to start servicing requests from HTTP clients.

TCP Socket Programming

The following section demonstrates a TCP/IP client and echo server, Domain Name Services, a TCP client HTML retrieval using a GET request, and sending e-mail using TCP/IP.

TCP Echo Server and Client

This example uses basic TCP/IP commands to implement an echo server, a program that simply responds with “Hello, world!” followed by the contents of any message it receives. The example also shows a client accessing the echo server. A simple protocol consists of establishing a connection to a client and echoing all messages until a message containing only a period tells the echo server to close the connection. A client connecting to the server may also send a message containing only an exclamation point, signaling the server not only to stop the current session, but to stop listening for any more connections.

Whereas many TCP/IP stack implementations contain a built-in echo server, this example implements a simple version of that command in an OMNIS method.

Server Code

```
; Declare the following local variables:
; listenSocket (Long int), message (Char)
; connectedSocket (Long int), portNumber (Short num 0 dp) = 1234
; status (Long int), messageLength (Long int)
TCPSocket Returns listenSocket
```

Creates a socket on which OMNIS listens for connections.

```
TCPBind (listenSocket,portNumber) Returns status
```

Binds that socket to a port, in this case port 1234. Next allow the socket to accept connections from clients.

```
TCPListen (listenSocket) Returns status
Repeat
   ; until you get a message containing only an exclamation point
   Repeat
      ;; listen for connections via TCPAccept
      TCPAccept (listenSocket) Returns connectedSocket
   Until connectedSocket>0
   ; a socket number <0 means there was no client
   ; Get the message from the client
   TCPReceive (connectedSocket,message) Returns messageLength
   While message<>"."&message<>"!"
      ; while you don't see a stop flag
      ; Get the contents of the client's message to us
      TCPSend (connectedSocket,con("Hello, world!",message))
           Returns messageLength
      ; Concatenate "Hello, world!" and send the message back
      TCPReceive (connectedSocket,message) Returns messageLength
   End While
   ; Close the connection to the current client
   TCPClose (connectedSocket) Returns status
Until message="!"
;  You have been told to stop listening, close the listening socket
TCPClose (listenSocket) Returns status
```

Client Code

This code is an implementation of a single client.

```
; Declare the following local variables:
; ServerAddress (Char) = 'TheServerAddress'
; ServerPort (Long integer) = 1234
; Socket (Long integer)
; Buffer (Char), and Status (Long integer)
TCPConnect (ServerAddress,ServerPort) Returns Socket
```

Connects to the remote machine.

```
If Socket>=0     ;; If successful...
   TCPSend (Socket,'This is message 1 connection 1') Returns Status
   Repeat    ;; Wait for a response
      TCPReceive (Socket,Buffer) Returns Status
   Until Status>0
   ; Send another message
   TCPSend (Socket,'This is message 2 connection 1') Returns Status
   Repeat    ;; Wait for a response
      TCPReceive (Socket,Buffer) Returns Status
   Until Status>0
   ; Signal the end of this session
   TCPSend (Socket,'.') Returns Status
   ; Now clean up
   TCPClose (Socket) Returns Status
End If
```

Domain Name Services

The TCP/IP Domain Name Services (DNS) commands resolve server names when given IP addresses and vice versa.

Address-to-Name Example Code

```
; Declare the following local variables:
; LocalPort (Long integer) = 1111
; AcceptSocket (Long int), ListenSocket (Long int)
; Buffer (Char), Status (Long int)
; RemoteAddress (Char), RemoteName (Char)
; LocalAddress (Char), LocalName (Char)
TCPSocket Returns AcceptSocket
```

Creates a socket for accepting connections.

```
TCPBind (AcceptSocket,LocalPort) Returns Status
```

Specifies the port on which to listen.

```
TCPListen (AcceptSocket) Returns Status
```

Puts the socket in listen mode. Now wait for an incoming connection.

```
Repeat
   TCPAccept (AcceptSocket) Returns ListenSocket
Until ListenSocket>=0
```

Having made a connection get the remote machine's address and domain name.

```
TCPGetRemoteAddr (ListenSocket) Returns RemoteAddress
TCPAddr2Name (RemoteAddress) Returns RemoteName
```

Now clean up…

```
TCPClose (ListenSocket) Returns Status
TCPClose (AcceptSocket) Returns Status
```

Now get the local machine's address and domain name, and display the result.

```
TCPGetMyAddr Returns LocalAddress
TCPAddr2Name (LocalAddress) Returns LocalName
OK message (High position,Large size) {My address is '[LocalName]'
   ([LocalAddress])// I was contacted by '[RemoteName]'
   ([RemoteAddress])}
```

Name-to-Address Example Code

The following gets the address of the OMNIS Software web server using the *TCPName2Addr* command.

```
TCPName2Addr ('www.omnis-software.com') Returns lvIPAddress
; returns something in the form 192.135.236.7
OK message (High position,Large size) {The IP address of the OMNIS
   software web server is '[lvIPAddress]'}
```

TCP HTML Retrieval

The following example uses the basic TCP/IP commands to retrieve an HTML page from an HTTP server. The protocol consists of establishing a connection to the server and issuing a GET-type request, specifying the pathname of the HTML document to be downloaded.

Although OMNIS includes the *HTTPPage* command to accomplish the same task, this example illustrates how a simple version of the protocol can be implemented using low-level commands.

```
; Declare the following variables:
; Server (Char) = 'www.omnis-software.com'
; Path (Character   10000000) = '/index.html'
; that is, the pathname of the HTML document to retrieve
; Socket (Long integer), Page (Char)
; Temp (Char), Status (Long int)
TCPConnect (Server,'http') Returns Socket
```

Connects to the server.

```
TCPSend (Socket,con('GET ',Path,' HTTP/1.0')) Returns Status
```

Sends a GET request. Next, retrieve the page being sent back by the server.

```
Repeat
   ;; Get the data currently in the buffer
   TCPReceive (Socket,Temp) Returns Status
   ;; Append it to what's already received
   Calculate Page as con(Page,Temp)
   ; Loop until all of it is received
Until Status<=0
```

When you're done, close the socket.

```
TCPClose (Socket) Returns Status
OK message (High position,Large size) {Page = "[Page]"}
```

TCP Client E-mail

The following example demonstrates how the basic TCP/IP commands can send a mail message via the SMTP protocol. This protocol simply consists of establishing a connection to a server and issuing commands as you might do with a line-oriented command shell such as DOS or Unix. The difference is that you use OMNIS commands to issue commands and look for responses. Listening for responses at the appropriate time is mandatory!

This example implements a simple version of the *SMTPSend* command in an OMNIS method. Before running this example, you should change the contents of the vars senderAddress, recipientAddress, and smtpServerAddress to addresses that are specific to your site.

```
; Declare the following local variables:
; Socket (Short integer   (0 to 255))
; senderAddress (Char) = "me@mydomain.com"
; recipientAddress (Char) = "you@youraddress.com"
; smtpServerAddress (Char) = "smtp.somedomain.com"
; status (Long integer)
; recvBuffer (Character   2048)
; sendBuffer (Character   2048)
; myAddress (Char)
; numCharsReturned (Long integer)
TCPConnect ("interserve.com","smtp") Returns Socket
```

Opens a socket to an e-mail server at the SMTP socket.

```
TCPReceive (Socket,recvBuffer) Returns numCharsReturned
```

Gets the connection response.

```
TCPGetMyAddr Returns myAddress
```

Determines your IP address - it is required by the SMTP protocol. Now send your IP address and begin the protocol with the HELO command.

```
Calculate sendBuffer as con("HELO ",myAddress,chr(10))     ;; Lines
   end with a carriage return character
TCPSend (Socket,sendBuffer) Returns numCharsReturned
TCPReceive (Socket,recvBuffer) Returns numCharsReturned
```

Say who the e-mail message is from with the MAIL FROM command, and who is the recipient with one or more RCPT TO: commands.

```
Calculate sendBuffer as
        con("MAIL FROM: <",senderAddress,">",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
TCPReceive (Socket,recvBuffer) Returns numCharsReturned
Calculate sendBuffer as con("RCPT TO:
   <",recipientAddress,">",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
TCPReceive (Socket,recvBuffer) Returns numCharsReturned
```

The DATA command begins the actual e-mail message.

```
Calculate sendBuffer as con("DATA",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
```

Even though the e-mail server gets the sender and recipient in the MAIL FROM and RCPT TO commands, every e-mail message must contain at least this information in a message header along with a Subject: line.

```
Calculate sendBuffer as con('From: "Hello,world! Example"
   <',senderAddress,'>',chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
Calculate sendBuffer as con('To: "Pop Test"
   <poptest@www.omnis-software.com>',chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
Calculate sendBuffer as con("Subject: Hello, world!",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
```

The header is separated from the message by a blank line.

```
Calculate sendBuffer as chr(10)
TCPSend (Socket,sendBuffer) Returns numCharsReturned
;
Calculate sendBuffer as con("Hello, world!",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
```

And the message is terminated by a line containing only a period.

```
Calculate sendBuffer as con(".",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
```

The SMTP protocol is terminated with the QUIT command.

```
Calculate sendBuffer as con("QUIT",chr(10))
TCPSend (Socket,sendBuffer) Returns numCharsReturned
```

Now, close the socket.

```
TCPClose (Socket) Returns status
```

The complete SMTP protocol is very complicated and allows for sending to multiple recipients (see *SMTPSend*). Recipients on the CC and BCC lists are taken care of by the e-mail message header. In addition, special message content and other information may be contained in an e-mail header, though this example restricts the header contents to the absolute minimum required by the protocol and e-mail servers.

Additional Information

The “TCP Listener” Common Code method in the m_cc_webserver class speeds up the process of listening for client requests using TCP commands.

Sockets are numbered (allocated) sequentially, starting with 0 (zero) under MacOS or 1 under Windows. The maximum number of connections is 32 under MacOS and 64 under Windows.

Many of the HTTP commands automatically encode and decode CGI information. Refer to those commands before assuming that you have to do any work for the most common operations, that is, POST/GET, parsing an HTTP header, and so on.

Internet Utilities

This section shows how to UUencode and decode information for transmission via e-mail or other Internet facilities, how to CGI-encode and decode text into a form acceptable as an argument to a Web server CGI, and how to use the binary file commands for writing to and reading from binary files in OMNIS methods.

UUEncoding and UUDecoding

UUencoding is used to move binary and text data between computers of all types, avoiding byte-ordering problems. It is also used to pass username and password from a Web browser in response to an HTTP 404 Authentication challenge.

```
; Declare the following local vars for en/decoded chars:
; lvEncodedText (Char)
; lvDecodedText (Char)
```

Encode text using the *UUEncode* command.

```
UUEncode ("Hello, world!",lvEncodedText)
OK message (High position,Large size) {Hello, world! UUEncodes as
   [lvEncodedText]}
; Message reads: Hello, world! UUEncodes as SGVsbG8sIHdvcmxkIQ==
```

Now decode the text using *UUDecode*.

```
UUDecode (lvEncodedText,lvDecodedText)
OK message (High position) {[vEncodedText] UUDecodes as
   [lvDecodedText]}
; Message reads: SGVsbG8sIHdvcmxkIQ== UUDecodes as Hello, world!
```

CGIEncoding and CGIDecoding

CGI Encoding is used to pass special characters in URLs, especially spaces and characters, which might otherwise be mistaken for delimiters.

```
; Declare the following local vars for en/decoded chars:
; lvEncodedText (Char)
; lvDecodedText (Char)
```

Encode characters (in this case, text) using *CGIEncode*.

```
CGIEncode ("Hello, world!") Returns lvEncodedText
OK message (High position,Large size) {Hello, world! CGIEncodes as
   [lvEncodedText]}
; Message reads: Hello, world! CGIEncodes as Hello%2C%20world%21
```

Now decode the text using *CGIDecode*.

```
CGIDecode (lvEncodedText) Returns lvDecodedText
OK message (High position,Large size) {[lvEncodedText] CGIDecodes as
   [lvDecodedText]}
; Message reads: Hello%2C%20world%21 CGIDecodes as Hello, world!
```

Reading and Writing Binary Files

This example shows how you use the binary file commands to write and read from binary files in OMNIS methods. For simplicity, the example below writes a text string to disk and reads it back into memory. However, these commands would normally be used to read and write binary-type data, such as graphical images, spreadsheet documents, and so on.

```
; Declare the following local variables:
; Text (Character   10000000) = 'Hello world!'
; PathName (Character   10000000) = 'hello.txt'
; BinaryField (Binary)
; Status (Long integer)
```

Copy the text string to a Binary field using *UUEncode*.

```
UUEncode (Text,BinaryField)
```

Write the contents of the Binary field to disk using *WriteBinFile*.

```
WriteBinFile (PathName,BinaryField) Returns Status
```

Clear variables, then read the file's contents back into a Binary field using *ReadBinFile*.

```
Calculate Text as ''
Calculate BinaryField as
ReadBinFile (PathName,BinaryField) Returns Status
```

Copy the contents of the Binary field to a text variable using *UUDecode*.

```
UUDecode (BinaryField,Text)
OK message (High position,Large size) {[Text]}
```

Programming Tips

This section discusses the ways that you can adapt your programming style and practices to the demands of the Internet or Intranet. It is designed for people who are just beginning to Web-enable their applications.

Error Handling

An essential fact to remember about all Internet connections is that your environment is uncontrolled. Factors such as network traffic are unpredictable and may interrupt the current activity at any time. For this reason, you must constantly *check* return values, *monitor* and *handle* errors. Consider the following factors as you plan your application.

Whenever a TCP or HTTP command performs any type of socket operation, your code must check the command's return value for an error and, if one occurs, close the socket. For example, TCP commands return a value of 0 (zero) to indicate success. The following code checks for this value and closes the socket if there is an error.

```
TCPBind (fListeningSocket,fPort) Returns vStatus
If vStatus<>0
   TCPClose (fListeningSocket) Returns vStatus
   Quit method (flag clear)
End If
```

Be sure to define all variables containing socket numbers and command return values as Long Integer. If you use Short Integer values, you cannot check for error conditions, because an OMNIS Short integer cannot store a negative value.

When you perform an operation on a socket, reading information for example, the socket can operate in one of two modes, *blocking* or *non-blocking*:

– A blocking socket keeps waiting until it receives data, and the function is not completed until the data arrives.

– A non-blocking socket does not wait but instead returns immediately with an error message to indicate that there is no data to receive.

When you are using TCP or HTTP commands to connect to a socket, first decide whether to use blocking or non-blocking sockets and program accordingly. Blocking sockets have their uses, but non-blocking sockets are generally better. If a socket blocks, it stops processing until it receives what it requires, while a non-blocking socket keeps returning the WinSOCK error -10035 until it receives what it needs or times out. Under Windows, use the *TCPBlock* command to toggle socket-blocking on or off. The following example shows code for blocking sockets:

```
TCPConnect ('ServerAddr',123) Returns Socket
If Socket >=  0
   TCPSend (Socket,LongMessage) Returns Status
   TCPReceive (Socket,TheResponse) Returns Status
   TCPClose (Socket) Returns Status
End if
```

And for non-blocking the code might look like this:

```
TCPConnect ('ServerAddr',123) Returns Socket
If Socket >=  0
   Repeat
      TCPSend (Socket,TheMessage) Returns Status
   Until Status <> -10035
   Repeat
      TCPReceive (Socket,TempBuffer) Returns Status
   Until Status > 0
   Repeat
      Calculate TheResponse as con(TheResponse,TempBuffer)
      TCPReceive (Socket,TempBuffer) Returns Status
   Until Status = -10035
   TCPClose (Socket) Returns Status
End if
```

By default, Windows sockets are non-blocking. Under MacOS, sockets are always non-blocking.

OMNIS cannot determine whether a particular command is being executed on a machine acting as a server or as a client. Therefore, when writing server applications, you should use the *WebDevError* command to specify an error-handling routine, otherwise OMNIS

displays a modal dialog each time an error occurs, effectively disabling the server until the error is acknowledged.

The error codes returned by the Internet commands are categorized as follows:

-1 to –999	Deterministic programmer errors such as insufficient parameters, invalid parameter values, and so on
-1000 and below	Non-deterministic runtime errors such as insufficient memory, inability to connect to server, and so on.

You need to know immediately about deterministic errors, whereas you don't want runtime errors to interfere with the server's operation. Here is an example of an error handling method that does this.

```
Error Handler
-------------
; Define the following parameters:
; pWE_ErrText (Character 500)     ;; error text
; pWE_ErrCode (Long integer)      ;; error code
; pCommand (Character 50)         ;; Command name
; pWS_ErrCode (Long integer)      ;; WinSock error code
If pErrCode > -1000   ;; Check for programmer errors
   Ok message (Large size, sound bell)
        {Error! Command = [pCommand].Code = [pWebDevErrCode]
        //Text = "[pWebDevErrText]"}
Else                  ;; It's a runtime error, so log it
   Begin reversible block
      Set current list cvErrors
      ; List cols defined as cvCommand,cvCode,cvText,cvWSCode
   End reversible block
   Add line to list (pCommand, pWE_ErrCode, pWE_ErrText,
   pWS_ErrCode)
End if
```

Client/Server Connections

A system of sockets and ports categorizes the communications among Internet computers. A telephone switchboard offers an analogy. When you call a company's main listed phone number, a person answers and then directs your call to the employee you are trying to reach. The main line is then free to take another call. Similarly, when a client accesses an HTTP server, for example, on Port 80 at a given socket, the server accepts the request and routes it to a different local port and socket for processing.

Client/server connections vary according to the type of protocol you are using. FTP operations are session-based and straightforward. Your client connects to the server,

performs the operations it requires, then disconnects. The client can perform multiple operations using a single connection.

For Mail operations, the e- mail commands handle the opening and closing of connections to POP3 and SMTP servers. A single command is used to send one message or receive all waiting messages.

HTTP and TCP connections require the most adjustment to standard LAN client/server programming styles. Keep the following points in mind when using these commands to handle client/server interactions.

A TCP connection typically involves many operations. Some of the higher-level HTTP commands reduce the number of steps required to implement the HTTP protocol using TCP commands:

– At the server, opening a socket

– At the server, listening for client requests at a port and socket. At the client, specifying the server address and port

– At the server, accepting the client requests and returning a new socket number for receiving them. At the client, establishing the connection with the server

– Sending and receiving information

– Closing the socket

Opening a connection is something like opening a file: When the transfer is complete, the socket must be closed. However, the consequences of leaving a socket open are more serious—at the very least, a tied-up resource until the connection times out.

Sockets are either blocking or non-blocking, as described above. Although it's often easier to program using blocking sockets, you surrender control of the socket until the operation is complete. For this reason, non-blocking sockets are usually preferable.

To transmit information, the TCP/IP protocol monitors the exchange of information (in *packets*) and attempts to estimate how long to wait for a reply before re-sending a packet. For this reason, you will encounter limits on how many characters you can send in a single operation. If you send very long character streams, eventually the protocol will drop packets and force them to be retransmitted. A good rule of thumb is to keep your messages short. When you are querying a Web server, break long queries into small ones and cache the results.

Intercomputer transfer times can vary widely and for that reason, you may need help when timing certain intercomputer transfers. Sending a ping to the distant computer allows the program to determine whether a time-out might close the socket or otherwise interfere with downloading a file. Use *TCPPing* to send an ICMP request packet to a specified IP address or named host.

Caching

Caching has been described by some people as the most important technique you can apply to your Internet applications. It's obvious to anyone who has surfed the Web that a browser cache saves the client a lot of time and spares resources on HTTP servers.

Proxy servers often play a part in caching by maintaining a network-level cache that is accessible to all the browsers in an organization. A cache server of this type maintains URLs in a database and uses them to satisfy local requests. When you do not want a proxy server to cache a reply to a request, you can include pragma:no-cache in the response header.

When information does not change rapidly, re-querying your server can affect performance. Whether or not you cache queries and responses depends upon a number of factors, primarily related to the type of application. It's a good idea to plan caching after doing some analysis.

– Are many requests repetitive? If most queries are unique, there is little point in caching.

– Measure the number of hits you get on information that you may want to cache, model your measurements, and ask yourself whether you require the speed.

– Look at the time sensitivity of information that people request. For example, if users require a daily sales report, you can simply run it at the same time each day, convert it to HTML, and make it available as a static URL. It's not necessary to create the report on demand each time that it's needed.

In designing your applications, caching is also important because of the short-lived connections you must create in order to maintain reliable and monitored data exchanges among Internet data resources. What was a single connection within a LAN becomes a series of short connections on the Internet. You break a long query into shorter queries, then retrieve each response and cache it.

Your application can cache URL requests to your database. For example, you can keep requested URLs in a two-column list, with one column holding the request and the other the data response. When a user queries the database, your application consults the list and looks for a match. If there is no match, the server retrieves the information from the data source rather than returning a cached value. To implement expiration dates and times for cached requests, you might set up a trigger that fires and makes an entry in a second flag table. When a flag is set by the date and time of a request, the application knows that it must search anew for the query response rather than using the cached data.

You can also increase performance by using an OMNIS datafile an as intermediary cache resource. When a request is made to, for instance, your SQL database, the response data is stored in the intermediate OMNIS datafile. Then when a client makes the same request again, the data is retrieved from the OMNIS datafile much faster than it would be from the originating SQL database.

You can also use *cookies* to help users request information that they have requested before. A cookie is a piece of information sent from an HTTP server to a client for storage. It can provide your application with functionality similar to that of Web browser caches, except that it works for queries to a database. When a client makes a request to a server, a cookie is set in the HTTP server response header, telling the client to store the request. Subsequently, when the client requests an URL from the server, the client application looks for a match with any stored cookies and places the URL/response in the client request header. A cookie can store the expiration date and time along with the query.

The cookie technique is generally implemented through a CGI script. Cookies are used on the Internet, for example, to store user registration information so that once a user has registered at an HTTP site, they need not re-type an ID to gain registered access. Here is a sample showing a HTTP server header response that specifies a cookie to be stored by the client for future use.

```
Set-Cookie: NAME=MyCookie; expires=01/01/98;
path=/; domain=omnis.com;
```

A cookie can bring a stored database record by referencing a lookup key, for example:

```
Set-Cookie: KEY_FIELD=0056; expires=01/05/98;path=/;
```

To revoke a cookie, a new cookie is set with the same name as the original one, but with an expiration date in the past.

Caching and cookie expiration techniques dictate the need for a *consistent* date and time setting on your server. Be careful: Daylight Savings Time adjustments, for example, can wreak havoc with caching and cookies. Servers that are accessed globally should use Greenwich Mean Time (GMT). A commonly accepted format is: `Sun, 06 Nov 1997 18:26:02 GMT`.

Troubleshooting Common Problems

The Internet relies upon agreed standards for naming and protocols. Problems can arise when your application connects to the Internet and does not behave as expected by different types of servers. Standards and conventions are well-documented at various Internet sites.

Many systems, including Windows systems, have a TCP/IP Services file that lists certain types of servers and specifies service names corresponding to port numbers that are used in accessing them. In this way, if the underlying port number changes (as it has for POP mail servers, for example), programs still function, because they refer only to the service name. The TCP/IP system translates the name by looking in the Services file dynamically. Internet mail and FTP commands, when used under Windows, refer automatically to the Services file and use the entries when accessing mail servers. In addition, some TCP commands optionally use a service name from the Services file. You can override the Services file entry by using a port number instead.

When you update your operating system or communications software, you may find that you have more than one Services file on your hard disk drive(s). The operating system TCP/IP software will only use one of them, which lists the service names and port numbers. Some old TCP/IP systems' Services files will have out-of-date port listings.

The following table lists a few of the problem symptoms that you may encounter, the most common cause or causes of those problems, and some solutions.

Symptom	**Possible Cause(s)**	**Possible Solution(s)**
OMNIS or system freeze	Attempting to quit OMNIS with open sockets	Track all open sockets and ensure that they are closed
The server is not servicing client requests	You started listening on a socket, stopped to do something else, attempted to listen again, but the listening socket was already bound to a listening port and wasn't closed. The *TCPBind* operation is failing.	Ensure that the listening socket is closed, by closing all recorded sockets. If you don't know which sockets are open, guess a number range to close, or check the documentation for your sockets implementation and close all sockets that could be open.
The server hangs	The OMNIS modal *Enter data* command on the server stops processing until its modal dialog is dismissed.	Consider using modeless data entry.
	The server encountered an error that was not handled by the *WebDevError* command. If you don't use *WebDevError* to set up a method for handling errors, Internet commands report errors by displaying a modal message that stops processing until the dialog is dismissed.	Use *WebDevError* to call a method when a command returns an error.
Some client requests are never answered or they time out.	Long, blocking asynchronous processes are taking place on the server, for example, large database queries, mail deliveries, or DNS lookups.	Minimize blocking asynchronous processes. If they are database requests, constrain your query system so shorter queries are performed. If you are performing network operations, ensure the remote resources are as accessible as possible to your server. This may mean running multi-threaded mail or DNS server applications directly on your server.

Chapter 9—Extending OMNIS

This chapter describes various features and third-party applications you can access and use in OMNIS to extend its functionality. Some of these features are loaded as external components, while others access an interface available under one platform only. The features you can access include

– *OLE and Automation*
under Windows, OLE lets you link or embed different types of object in your server or OMNIS database, including spreadsheets, pictures, and wordprocessing documents

– *DDE*
lets you exchange data and commands with other programs running under Windows

– *Lotus Notes*
lets you manage and update Lotus Notes databases and e-mail

– *Apple Events*
lets you send standard commands and data to and from OMNIS, and interact with other MacOS applications including word processors and databases

– *Publish and Subscribe*
is a feature of the MacOS that lets you make data available to other applications or access such data

– *Help*
describes how you create help for your own application

OLE Pictures

You can use OLE in your applications to link or embed OLE-aware objects in your database. The Windows implementation of OLE2 provides features such as Linking and embedding, and in place activation. The OLE Picture external component is loaded by default and found under the External Components button in the Component Store.

Using the OLE Picture component you can link or embed many different types of object into your database, including spreadsheets, charts, pictures, and wordprocessing documents. You link or embed the object in an OLE Picture field while your window is in enter data mode. Note that *embedded* objects are stored in your database whereas *linked* objects store the link or location of the source file only.

Placing an OLE Picture

To place an OLE Picture on your window

- Open your window in design mode
- Open the Component Store, or bring it to the top, by pressing F3
- Click on the External Components button in the Component Store toolbar
- Click on the OLE Picture icon and create the field on your window
- Select the field and set its properties in the Property Manager

The OLE field is an external component of type kComponent. You can set its dataname property to a variable of Picture type.

Inserting Objects

You can link or embed an object in an OLE Picture field using the Edit menu. The window containing the picture field must be in enter data mode when you insert the object. You can create three types of object.

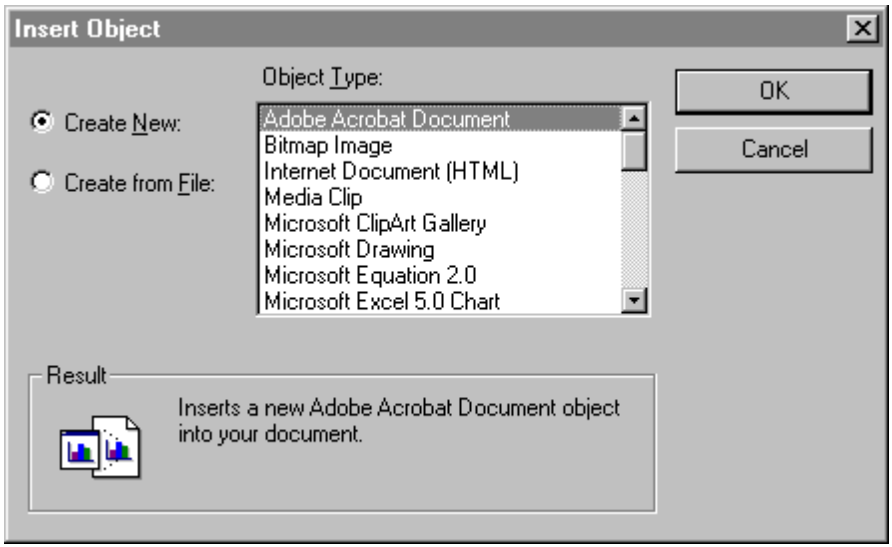
– A new embedded object

– An embedded object from an existing file

– An object linked to an existing file

To insert an OLE Picture component

- Put OMNIS in enter data mode (this is set in the window by default as the **modelessdata** property)
- Tab to your OLE field, or click in the field (the focus is shown as a dotted border)
- Choose the Edit>>Insert Object menu item



The Insert Object dialog lists all the OLE-aware applications on your system.

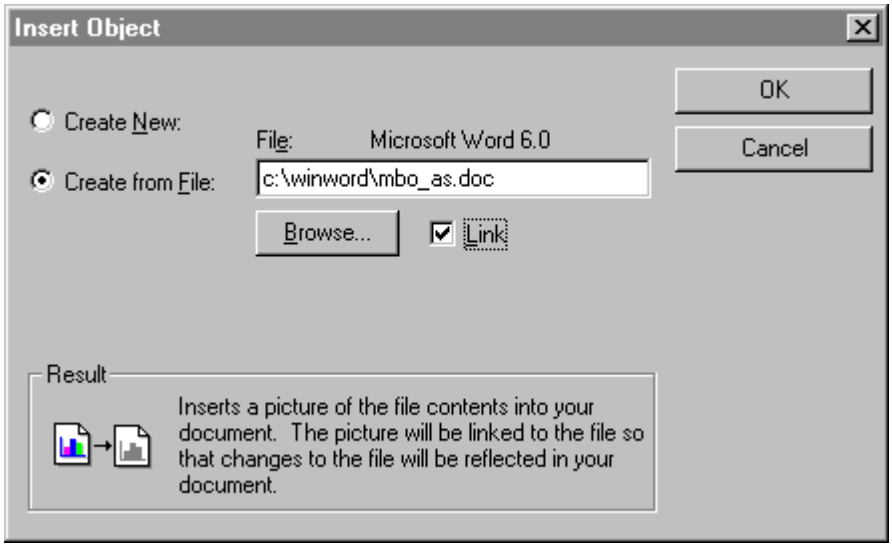
To create a new embedded object

- Select the Create New radio button (the default)
- Double-click on the application type, for example, Microsoft Excel 5.0 Worksheet

The server application is launched. You enter the data into the server application and close the application (you may also want to save the source file). The object appears in the OLE Picture field. To save the object you should update the record.

To insert an object from an existing file

- Select the Create from File radio button in the Insert Object dialog



- Enter the path and name of the file you want to insert or click on the Browse button to locate the file

If you want to create a link, instead of embedding the object

- Select the Link check box and click on OK

The linked object appears in the OLE Picture field. To save the linked or embedded object you should update your server or OMNIS database.

Alternatively, you can paste an OLE object into the OLE Picture field if you have placed OLE data on the clipboard (your window must be in enter data mode and the cursor must be in the picture field).

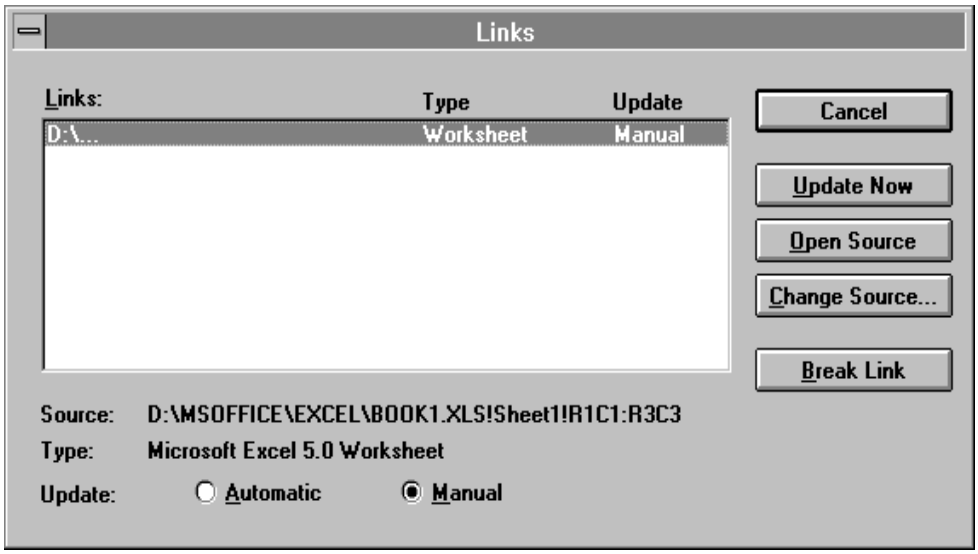
- Select **Edit>>Paste** to create an embedded object, or
- Select **Edit>>Paste Link** to create a link to the object

Linking Objects

To create a link to a source file, you must first open the server application, create the source file, and Copy the data you require. This places the OLE data on the clipboard.

To place the link

- Put OMNIS in enter data mode
- Place the cursor in your OLE Picture field
- Select the Edit>>Links menu item



The Links dialog lists all the current links and lets you specify whether the link is updated manually or automatically.

- Double-click on the link you want to insert

When you first link to an object, OMNIS sets the link to Manual update. You click on the Update Now button to update the link manually.

To start the application to edit the data, click on the Open Source button.

To change the link to another source, click on the Change Source button and use the resulting dialog to specify a new link as before. You select the Automatic radio button to force the link to update when the source changes.

Note that if the linked object is large, automatic updating can take quite some time, and this can happen at unexpected moments.

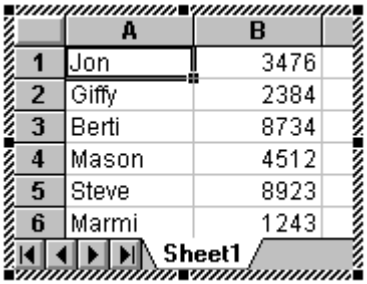
To copy the contents of an OMNIS field to another application, click in the field while in enter data mode, and select **Edit>>Copy**. Switch to the other application and paste the link.

Edit Menu Verbs

When an OLE Picture field contains a linked or embedded object, the Edit menu contains an item corresponding to the object, for example, Picture or Linked Picture. This submenu gives you up to four verbs, such as Edit, Play, and so on. It can contain a Convert item that lets you convert the type of the OLE2 object to another server application if applicable.

The Edit or Open item on the Edit menu always opens the server application as a separate application rather than activating the object in place.

In place Activation



You can edit an object *in place* (if the server application permits this), which means the OMNIS menu bar becomes partially controlled by the server application, with OMNIS supplying only the **Edit** menu. Also, OMNIS removes all its tool bars (including floating ones) for the duration of the in place editing.

To edit an object in place

- Locate the record/object you want to change
- Place OMNIS in enter data mode (or you may already be in enter data mode)
- Double-click on the object

If the object is larger than the OLE Picture field, in place activation *does not* occur. In this case the server application is launched as a separate window.

If you add data to the object you may need to resize it by dragging its handles to display the additional data in the OLE field. (For example, if you add more cells of data to an Excel worksheet, you must resize the field to include the new cells.)

To return to OMNIS and save the changes to the object, click somewhere in your OMNIS window outside the object.

Embedding and Linking using Drag and drop

You can drag objects from other applications onto an OMNIS picture field while it is in enter data mode. There are three possible situations with the following consequences:

– **No keys held down**
moves the object to create an embedded object

– **Ctrl key held down**
copies the object to create an embedded object

– **Ctrl and Shift keys held down**
creates a linked object in the picture field

Properties and Methods

OLE Picture fields have the following runtime properties and methods which let you manipulate the OLE object.

– **$classapp**
application name of the current OLE object, for example, on inserting a Word 97 object, this property would have the value ‘Microsoft Word 97’

– **$classfull**
full name of the current OLE object, for example, on inserting a Word 97 document object this property would have the value ‘Microsoft Word Document’

– **$classshort**
short name of the current OLE object; for example, on inserting a Word 97 document object this property would have the value ‘Document’

– **$getobject()**
returns an automation object (refer to OLE Automation for further information on automation objects). For example

```
; myObject is a variable defined as Object with no subtype
; myOLEfield is an item reference for the OLE Picture field
Do myOLEObject.$getobject() Returns myObject
Ok Message {The name of the OLE objects' server is
   [myOLEObject.$appname]}
```

– **$doverb(constant)**
invokes the specified action on the OLE object and returns an HRESULT error code (refer to OLE Automation for further information concerning HRESULT error codes); the following constants are available

kOLEPrimary	the default action for the object
kOLEShow	activates the object for editing; if the server application supports in-place activation, the object is activated in the OLE Picture field
kOLEOpen	opens the OLE object in a separate application window, even if the server application supports in-place activation
kOLEHide	causes the object to remove its user interface, toolbars, etc., from the view; applies to in-place activated objects only
kOLEUIActivate	if the object supports in-place it is activated, otherwise an error occurs
kOLEDiscardUndoState	discards any undo state without deactivating the object

OLE Automation

Under Windows 32-bit platforms, OMNIS Studio supports Automation, formerly known as OLE Automation, for Component Object Model (COM) objects which expose the dispatch (IID_IDispatch) interface, such as Microsoft Word and Excel.

In addition, under Windows 32-bit platforms only, OMNIS Studio supports remote automation, which means that the object resides on another machine, via the Distributed Component Object Model (DCOM).

OLE Automation is provided via non-visual external objects which you can use at any time without the need for a window or report instance.

Automation objects have the following basic life-cycle:

1. Construct the automation object (such as Microsoft Word).
2. Communicate with the automation object via methods and properties.
3. Terminate the automation object.

OMNIS is always the client of the automation objects it constructs.

You access automation objects by adding a new object variable and selecting the appropriate subtype.

If an error occurs, for example the construction of an automation object fails, you can inspect both #ERRCODE and #ERRTEXT to determine the error.

The following code is a simple example of using an automation object.

```
; objExplorer is an Object with the subtype set to
; 'InternetExplorer.Application.1'
Do objExplorer.$createobject()   ;; Create the object
Do objExplorer.$navigate   ; Invoke method
If objExplorer.$visible=0        ;; Is explorer visible?
   Do objExplorer.$visible.$assign(kTrue)   ;; Show explorer
End If
Do objExplorer.$quit()        ;; Quit/Shutdown explorer
```

OLE Automation Methods

You can use $createobject() to instantiate the automation object. This is not the only way to construct a new automation object. Every automation object in OMNIS has the following methods:

– **$createobject()**
  creates the automation object

– **$getobject(filename)**
  creates an automation object from the file; if the file is specified by empty (“”), a new object is created, otherwise the object is created and restored to the state in the file

– **$getactiveobject()**
  constructs an object to the active server object; note that this method relies on the server implementing RegisterActiveObject

– **$isavailable()**
  returns kTrue if the automation object is available for use, that is, $createobject() or $getobject() have been invoked successfully

OMNIS to Automation Variable Type Conversion

Automation has a data type called VARIANT, which can hold any type of data. Unfortunately, with this flexibility comes a price, namely that some objects, Excel for example, state that they handle any type of data, but in fact they may be expecting data passed by reference or of a limited subset of the VARIANT types. The OMNIS automation component takes the same approach as Visual Basic, meaning that everything is passed by VARIANT, making certain assumptions based on the OMNIS data type you use. You can override these assumptions if you wish.

The default approach is as follows.

– Everything is passed in a VARIANT type by value.

– Convert to and from automation data types using the following conversion table.

OMNIS Data Type	Automation Data Type
Boolean	VT_BOOL
Integer (0..255)	VT_I1
Integer (Long)	VT_I4
Number	VT_R8
Character	VT_BSTR
List	VT_ARRAY | VT_VARIANT
Row	VT_ARRAY | VT_VARIANT
Binary	VT_ARRAY | VT_UI1
Object	VT_DISPATCH

Passing Parameters by Reference

Passing by value ensures that the server cannot change the value of variables in OMNIS. However, sometimes you want to pass parameters by reference, so that their value can be updated. To do this, simply precede the variable with the appropriate constant from the 'Automation Library' constant group.

For example, suppose the object has a method *getName*, which requires a parameter passed by reference.

```
; This fails to change myVar
Do myObject.$getname(myVar)
; This successfully changes myVar
Do myObject.$getname(kAutoBSTRREF,myVar)
```

Coercing Data Types

You can coerce parameters, and values to be assigned to properties, to one of the following automation data types:

Automation Data Type	Constant	Description
VT_BOOL	Either kAutoBOOL or kAutoBOOLREF	Boolean value (True or false)
VT_I1	Either kAutoI1 or kAutoI1REF	Signed byte (-128 to +127)
VT_UI1	Either kAutoUI1 or kAutoUI1REF	Unsigned byte (0 to 255)
VT_I2	Either kAutoI2 or kAutoI2REF	Signed short (-32768 to +32767)
VT_UI2	Either kAutoUI2 or kAutoUI2REF	Unsigned short (0 to 65535)
VT_I4	Either kAutoI4 or kAutoI4REF	Signed long (-2147483648 to +2147483647)
VT_UI4	Either kAutoUI4 or kAutoUI4REF	Unsigned long (0 to 4294967295)
VT_R4	Either kAutoR4 or kAutoR4REF	4 byte real (1.175494351E-38 to 3.402823466E+38)
VT_R8	Either kAutoR8 or kAutoR8REF	8 byte real (2.2250738585072014E-308 to 1.7976931348623158E+308)
VT_BSTR	Either kAutoBSTR or kAutoBSTRREF	Binary string (Limited by system resources)
VT_EMPTY	Either kAutoEMPTY or kAutoEMPTYREF	Corresponds to no valid data type
VT_CY	Either kAutoCY or kAutoCYREF	Currency value
VT_DISPATCH	Either kAutoDISPATCH or kAutoDISPATCHREF	IDispatch FAR * (4 bytes)

For example, suppose an object method called "FastSort" only accepted a VT_BSTR array:

```
; myList passed as a BSTR array, by value
Do myObject.$fastsort(kAutoBSTR,myList)
; myList passed as a BSTR array, by reference
Do myObject.$fastsort(kAutoBSTRREF",myList)
```

You can also coerce properties to a data type and calling convention (by reference or by value) by preceding the assignment variable by the datatype constant.

For example, suppose a visible property accepts an unsigned short:

```
; Boolean passed as unsigned short, by value
Do myObject.$visible.$assign(kAutoUI2,myBoolean)
; Boolean passed as unsigned short, by refernce
Do myObject.$visible.$assign(kAutoUI2REF,myBoolean)
```

Automation Errors

If an error occurs, the automation component sets #ERRCODE. A value of minus one indicates that the error occurred in the component, whereas other error codes are the HRESULT value returned from the server. In both cases, #ERRTEXT describes the error.

HRESULT codes are difficult to document as they can be defined by both the server application and the operating system. However, here are a few of the more common codes:

0x8000FFFF	Unexpected error
0x80004001	Not implemented
0x8007000E	Out of memory
0x80070057	Invalid argument
0x80004002	No such interface supported
0x80004004	Operation aborted.
0x80004005	Unspecified error
0x800401F3	Invalid class string

Limitations

Automation constants and events are not supported.

Using OLE Automation

Note that these examples do not include error checking for brevity.

The following example creates an Excel server object using Excel ‘97 and opens the Sample.xls workbook.

```
; excelObject is an Object with the subtype of Excel.Application.8
; workBookObject is defined as an object with no subtype
; Create Excel object
Do excelObject.$createobject()
; Make object visible
Do excelObject.$visible(kTrue)
; Obtain workbook interface
Do excelObject.$workbooks Returns workBookObject
; Open workbook
Do workBookObject.$open(“c:\samples.xls”)
; Mark as saved to avoid dialog
Do excelObject.$activeworkbook().$saved.$assign(kTrue)
; Quit excel object
Do excelObject.$quit()
```

The following example creates a Microsoft Word 95 object and opens a document.

```
; wordObject is defined as Object with a subtype
; of Word.Application.8
; documentsObject is defined as Object with no subtype
; Create word object
Do wordObject.$createobject()
; Make object visible
Do wordObject.$visible.$assign(kTrue)
; Obtain documents’ object
Do wordObject.$documents Returns documentsObject
; Open specified document
Do documentsObject.$open("c:\mydoc.doc")
; Quit word object
Do wordObject.$quit()
```

The following example uses Excel ‘97 to create a pie chart from a list variable.

```
; objExcel is an Object with the subtype of Excel.Application.8
; objWorkBook , objWorkSheet, and objRange are objects with
; no subtype.
; Create Excel object
Do objExcel.$createobject()
Do objExcel.$visible.$assign(kTrue)     ;; Make excel visible
Do objExcel.$workbooks().$xadd(xlWorkSheet) Returns objWorkBook
; Create a new worksheet
Do objWorkBook.$activesheet Returns objWorkSheet
; Obtain the object for the active sheet
; Setup area list
Set current list lAreaList
Define list {cArea}
Add line to list {("North")}
Add line to list {("South")}
Add line to list {("East")}
Add line to list {("West")}
Do objWorkSheet.$range("A1:D1").$value.$assign(lAreaList)
; Set a range of cells
Do objWorkSheet.$range("A2").$value.$assign(5.2) ;; Set cell value
Do objWorkSheet.$range("B2").$value.$assign(10)  ;; Set cell value
Do objWorkSheet.$range("C2").$value.$assign(8)   ;; Set cell value
Do objWorkSheet.$range("D2").$value.$assign(20)  ;; Set cell value
;  Create a pie chart from the cells
Do objWorkSheet.$range("A1:D2") Returns objRange
Do objWorkBook.$charts().$xadd().$chartwizard
   (objRange,xl3DPie,7,xlRows,1,0,2,"Sales Percentages")
OK message Excel {Example Finished}
; Print worksheet
Do objWorkSheet.$PrintOut()
; Mark workbook as saved to avoid excel prompting when
; application closes
Do objWorkBook.$saved.$assign(kTrue)
Do objExcel.$quit()     ;; Shutdown & quit excel
```

The following example creates a sort server object and demonstrates OMNIS and automation arrays.

```
; objSort is an object with the subtype of SafeArray.Application
; Create sort object
Do objSort.$createobject()
; Build list
Set current list #L1
Define list #S1
For #1 from 100 to 1 step -1
  Calculate #S1 as con(#1," - Contents of line")
  Add line to list
End For
;  COM Object expects BSTR array(passed by reference) so
; force conversion
; Sort list
Calculate #1 as objSort.$FastSort(kAutoBSTRREF,#L1)
; GetArray returns a list of 20 elements all with the same contents
; Get list
Calculate #L2 as objSort.$Getarray()
Set current list #L2
Redefine list {#S1}
; Quit/shutdown sort object
Do objSort.$quit()
```

DDE

Dynamic Data Exchange (DDE) is a Windows facility similar to OLE that lets OMNIS exchange data and commands with other programs running under Windows.

To set up a conversation as the client, OMNIS uses the *Open DDE channel* command. An acknowledgment from the server confirms that the conversation can take place, and OMNIS issues the transactions using commands such as *Request field*, *Send command*, and so on.

Before OMNIS can act as a server, the *Set server mode* command lets OMNIS respond to certain requests and commands from a DDE client.

Creating a DDE Link

Using the Edit menu, you can link data from programs such as Excel and Word to OMNIS entry fields and vice versa. In particular, you can link the data in an OMNIS list to a spreadsheet, so that it updates automatically when the values in the list change. Similarly, you can copy a spreadsheet to the clipboard and paste it into an OMNIS list field.

To create a DDE link to a spreadsheet

- In your spreadsheet, copy a range of cells to the clipboard
- In OMNIS, select a field such as a list
- From the Edit menu, select Paste Link

OMNIS as Client

To set up a conversation with a DDE server, you must open a channel to the server and give it a number. You can have up to eight channels open at the same time.

```
Test if file exists {EXCEL.EXE}
If flag true
   Start program normal (EXCEL)
   Set DDE channel number {1}
   Open DDE channel (EXCEL|SHEET1)
   If flag false
      Quit method kFalse
   End If
End If
```

The *Open DDE channel* command contains the name of the program, 'Excel', followed by the topic name. Note that the delimiter '|' must be entered as part of the parameter.

You use the *Send field* command to send data and the *Send command* command to send a command to a DDE server.

In this example, OMNIS is sending monthly sales totals from n1 to column two in an Excel spreadsheet. The value of CUSTOMER is sent to column one; char1 contains a string that includes the row and column numbers to which the data goes.

```
While PRODUCT=char2
   Calculate char1 as con('R',n2,'C',1) ;; Excel row [n2] Col 1
   Send field CUSTOMER {[char1]}
   Calculate n1 as Jans
   Calculate n1 as n1+Febs
   Calculate n1 as n1+Mars
   ....
   Calculate n1 as n1+Decs ;; totaling the monthly sales
   Calculate char1 as con('R',n2,'C',2) ;; Excel row [n2] Col 2
   Send field n1{[char1]} ;; send total month sales to row n2 col 2
   Calculate n2 as n2+1 ;; Row number increased by 1
   Next (Exact match)
End While
```

DDE makes extensive use of square brackets, for example, you must include the commands you send inside square brackets. This can lead to confusion since OMNIS tries to evaluate square bracket text as an expression. To solve this problem you can

– Put the text including brackets into a character field and send the contents of the field

– Use two initial square brackets

```
Calculate num2 as num2-1  ;; return to last row
Calculate char3 as con('R',num2,'C',2)  ;; last row column 2
Calculate char1 as
   con('[SELECT("R1C1:',char3,'")] [NEW(2)][GALLERY.PIE(5, TRUE)]')
; command to be sent to Excel to draw chart
Send command {[char1]}
Do method (wait)
Calculate char1 as '[OPEN("\EXCEL\SALES.XLM",0,0)]'
; command to be sent to Excel to open macrosheet
Send command {[char1]}
Calculate char1 as '[RUN("SALES.XLM!mclose")]'
; command to be sent to Excel to execute a macro
Send command {[char1]}
Close DDE channel
```

The DDEExecute message can contain one or more OMNIS commands. A string of commands sent in this way constitutes a method. The commands must follow a certain syntax in order to be understood by OMNIS.

The best way to create the script is to

– Create the method

– Copy the method to the clipboard

– Paste it into a text editor

Now you need only incorporate the method lines of the script in your application. Don't forget to edit out the title and error count lines. You can also print the method to a screen report and use the mouse to select the method script and copy in the usual way. This is a simple example written in Word for Windows:

```
DDEExecute ChanNum, "OK message {Word says Hi!}"
```

The script form for a method consists of a series of command lines separated by carriage return characters. '&' at the end of a line denotes that the next line is a continuation of the previous line. Each command line converts to a single OMNIS command.

The syntax for a command line is exactly the same as that in the right-hand list of the method editor. Case is not significant, and you can include extra spaces. The script cannot contain local, class or task variables since these must be already declared before using the variable. The method created from the script is in its own self-contained format that deletes as soon as the script finishes executing.

When you convert the text to a method, you can get a syntax error. OMNIS returns an error code of -1 plus an error string to the server. If there are multiple errors, OMNIS only reports the first. If the OMNIS debugger is available, OMNIS still creates a method on the method stack, replacing the syntax error with a *Breakpoint* command.

If there are no syntax errors, the created method is pushed onto the method stack and immediately executed. A reply is not returned to the sender of the Do Script event until the created method (and all its called methods) finishes executing, or an error occurs when executing the method (or any of its called methods), or until a command which returns control to the user is encountered (such as Enter data or any of the Prompt for... commands which open a non-modal window). If an error occurs and the debugger is available, the debugger is opened at the error in the usual way; otherwise, no error is reported to the user by OMNIS.

Methods which display an OK message or other modal windows do not count as returning control to the user, so the sender may be waiting for a reply for the period that the message is being displayed; this means that eventually the sender may receive a time-out reply to the DDEExecute message.

Requesting Data

You can request data from another program with the *Request field* command. It takes two parameters: the name of the data item in the server program and the OMNIS field name into which the data is placed:

```
Request field S3 {CCNAME}
Request field n5 {FEXCHANGE}
```

The data returned to OMNIS is read into the CRB.

Requesting Advise Messages

You can also ask a server to advise OMNIS whenever the value of a data item changes. If the request is accepted, the flag is set. From there on, if the requested item changes value, the server sends it to the CRB.

The following method sets up a channel to another OMNIS library and requests advise messages for three fields NAME, ADDRESS and TEL. The values are read into the fields CNAME, CADDRESS and CTEL in the client.

```
Set channel number {2}
Open DDE channel {OMNIS|COUNTRY}
If flag false
   OK message {Country library not running}
   Quit method
End If
Request advises TEL {CTEL}
Request advises NAME {CNAME}
Request advises ADDRESS {CADDRESS}
Prepare for insert
Enter data
```

```
; $event() method for the window
On evSent
   If $ctarget='ADDRESS'   ;; Last field has been sent
      Update files
      Queue cancel
      Redraw WindowName
   End If
```

The control method traps each DDE event (evSent) caused by the incoming field values. The fields are sent in the order they were requested, that is, TEL, NAME, ADDRESS. When the ADDRESS field has been received, the control method updates the files and cancels Enter data mode. The example is an interesting way of transferring data from one OMNIS library to another, and can be driven from the server library using *Send advises now* to specify when to send the field values.

OMNIS as the Server

When you want to run OMNIS as the server, you must specify the mode using the *Set server mode* command. By default, OMNIS accepts all commands and data from a DDE client.

OMNIS accepts the following DDE commands:

Accept advise requests lets OMNIS respond to a client program with values of requested advise messages.

Accept field requests sends the field value to the client program response to a Request message specifying a valid field name.

Accept field values responds to a Poke message specifying a valid field name by setting the value of that field to the value transmitted by the client program. Values are stored in the CRB and if the relevant field is on the top window, OMNIS redraws it.

Accept commands executes a command string sent by the client program.

Whenever **Accept field values** or **Accept commands** is enabled, either as an option under *Set server mode* or by the appropriate command, OMNIS processes the Take control command from the client. All conversations are terminated when the OMNIS library is closed.

Send advises now advises the client programs of all the values for all the fields for which it has received Advise requests, in the order that the client requested them.

Message timeout waits a specified length of time for responses to messages sent to other programs: default 30 seconds.

You use *Set advise options* to specify events that cause OMNIS to send values to a client (in addition to an active request, as above):

- **Find/next/previous** sends the requested advise values on *Find*, *Next*, *Previous*, or *Clear*
- **OK** sends the requested advise values when an *Enter data* or *Prompted Find* command ends with an OK event
- **Redraw** sends the requested advise values when OMNIS redraws

Printing Reports to a DDE Channel

You can send reports to an open channel either by selecting **Channel** in the **Print destination** hierarchical menu or by issuing a *Send to DDE channel* command. Each use of the command adds a field name to an internal list so that when you print the report, each field in the record section goes to a corresponding field in the server.

This method exports a report to a DDE channel using the tab-delimited format.

```
Set DDE channel number {2}
Open DDE channel {OMNIS|CLIENTS}
;  Check flag etc.
Send to DDE channel
Set export format {Delimited (tabs)}
Set report name DDEReport
Clear DDE channel item names
Set DDE channel item name {Name}
Set DDE channel item name {Address}
Set DDE channel item name {Telephone}
Set DDE channel item name {Town}
Print report
Close DDE channel
```

The System Topic

You must set up the server mode to enable OMNIS to respond as a server. DDE has however a special feature, the system topic, that is an exception to this. This is a tab-delimited CF_TEXT document containing several generic items. Whatever server mode OMNIS is in, when an Initiate message contains the name of the system topic, OMNIS always accepts requests to act as server.

OMNIS responds to any one of the four following items under the system topic:

– **SysItems**
  OMNIS returns a list containing the three items below in a Data message

– **Topics**
  OMNIS returns the name of the current library if there is one, otherwise a blank is returned

– **Formats**
  OMNIS returns a list of the clipboard format numbers which can be implemented

– **Status**
  OMNIS returns a DDE data item with one of the following strings, indicating the current status

  Topic open
  You have control status message
  No control status message
  Enter data
  Prompted find
  Import data

If OMNIS is in Enter data, Prompted find or Import data mode, the appropriate status is returned. If the client requesting the status has DDE control, OMNIS sends the status

message You have control. If no program has DDE control, OMNIS sends the status message No control. Alternatively, the data message contains an item indicating that OMNIS' status is busy.

Events during DDE

The message evSent is sent to the window control method to indicate that data has been updated via a DDE link. The parameter pChannelNumber provides the number of the DDE channel.

If the Data or Poke message does not contain a valid OMNIS field name, OMNIS rejects the message. The data item name need not correspond to an OMNIS field name on the enter data window for the message to succeed, but if the field is visible on the window there is an automatic redraw.

This example control method detects the arrival of a particular field and triggers an Update:

```
On evSent
   If $ctarget ='C_FIRSTNAME'
      OK message {Got first name}
      Update files
      Queue Cancel
   End If
```

Ack Bits

When OMNIS sends a Data message as server, the fAck bit is set which requests the client to send an acknowledgment.

When receiving Data messages as a client, OMNIS sends an acknowledgment if requested to do so by the incoming fAck bit set in the incoming Data message from the server. These acknowledgment bits are used by OMNIS to determine the value of the flag after DDE commands.

OMNIS as server returns a busy message when it is running methods, printing reports or running a menu option. This happens when field values are requested, poked or commands sent.

As a client, OMNIS returns a 'busy' Ack when running a method, printing a report, or executing a standard menu option.

OMNIS never returns a 'busy' Ack message in response to data which is the result of a Request message because OMNIS waits for the response.

Programming with DDE

This section has some tips for programming DDE calls to OMNIS.

Any client wishing to initiate a conversation with OMNIS must send a WM_DDE_INITIATE message with the Program name given as OMNIS. The topic name given in the message must be that of an OMNIS library without the .LBS extension. So, for example, an Initiate message addressed to OMNIS to open a conversation might read:

```
WM_DDE_INITIATE   'OMNIS'   'PERSON'
```

The client wishes to start a conversation with OMNIS on the topic of PERSON.LBS. For OMNIS to respond positively to an Initiate message, the name OMNIS must be present in the message. It is possible to send an Initiate message to OMNIS with the program name 'OMNIS', and a null topic. 'Null' in this context means that no value has been given for the topic, i.e. no library name has been supplied. OMNIS responds with a positive ACK message containing the name OMNIS, and the name of a currently supported OMNIS library. OMNIS sends one positive ACK message for each library which it supports.

If any parameter value sent to OMNIS as part of an Initiate message is invalid, OMNIS sends a TERMINATE message to the client.

To request data from the OMNIS CRB, the field name in the Request message must be a valid OMNIS field name as defined in the OMNIS file classes. The clipboard format required in the Request message must be CF_TEXT.

To send data to OMNIS via the Poke message, the field name must be currently valid in the OMNIS file classes, and the text type must be one of the supported clipboard formats.

Using DDE with Word

Under Windows, Microsoft Word has good support for DDE as a client through its DDE Field facility and its macro programming language. The easiest way to get a field value from OMNIS into Word is

- Start OMNIS and switch on the server options with *Accept advise requests (Accept)* and *Advise on Redraw (Advise)*
- In your Word document, select the **Insert>>Field** menu item
- From the list of Word field types, choose **DDE Auto**
- Add the name of OMNIS, the OMNIS library and OMNIS field to the parameter line for the Word field

```
ddeauto OMNIS DDE2 C_COMPANY
```

With the Show field codes option turned off, Word displays the OMNIS field and updates it when the value changes. This method uses Word as the client, and you need to enable

OMNIS as a DDE server. Word issues the 'advise request' when you create the field or when you open the document containing the field.

You can use Word Basic to set up DDE links to OMNIS, but remember to put OMNIS in your PATH statement. Here’s a simple example

```
Sub MAIN
Shell "OMNIS.EXE  \OMNIS\DDE\DDE2.LBS"
; Wait for OMNIS to start
For x = 1 to 1000
Next
Beep
ChanNum = DDEInitiate("OMNIS", "DDE2")
Print ChanNum
Print DDERequest$(ChanNum, "C_FRSTNAME")
DDEExecute ChanNum, "Next"
Print DDERequest$(ChanNum, "C_FRSTNAME")
DDETerminate ChanNum
End Sub
```

Lotus Notes

Lotus Notes is a client server package that lets you share data with other users. It contains a database facility, e-mail, and other features for sharing objects. In particular, you can replicate data on different notes servers and Notes synchronizes the replicated data automatically. OMNIS supports Lotus Notes version 4.0 and later on Win and MacOS. You must have the OMNIS Lotus Notes external in the EXTERNALS folder to access your Notes database; this is installed by default.

Data Types

The Notes data types map to the following OMNIS data types:

Notes	OMNIS
Text	Character (15000)
Date range	Character
Number range	Character
RTF	Character
Text List	first column of a single char column list in OMNIS
Date-time	Date time
Number	Number
Note ID	Number

There are some limitations on transferring data from Notes to OMNIS:

- Because Notes has a summary buffer size of 15,000 bytes, text fields are limited to this size. RTF fields don’t have that limitation.
- You can't create fields with date range or number range data types.
- The list definition the map command uses can't exceed 512 characters
- You can have up to 32 mapped fields
- You can have up to 8 open data files
- The *NSF Write composite* command chooses default style for the text and this is hard-wired
- The Lotus Notes API limits the number of items in a view that you can list to 32K.

Lotus Notes Commands

You need to create fields or variables in OMNIS with the same names as the Notes field names you want to use and with compatible data types.

Notes Command	Parameters	Returns
NSF Set Error Field	Error field name	Status
NSF Open File	Pathname	Status
NSF Close all files		Status
NSF Close File	Pathname or 'Mail_File'	Status
NSF Get Info		Information string
NSF Who am I		Information string
NSF Where's my mail?		Information string
NSF Make server path	Server and NSF File	Path to server/file
NSF List Open NSF Files	List name	Status (number open??)
NSF Map fields	List name	
NSF Select	Listname, Select macro, Date, View name	Status (number found/error)
NSF Build View	View name, List name[,Text key][,'Partial']	Number of notes found
NSF Make Note	None	Note_ID
NSF Copy Note	Path for destination	Note_ID
NSF Mail note	Note_ID	Note_ID (created)
NSF Delete Note	Note ID	Status
NSF Write composite	Note ID, Commit flag,Fields1...n	Status (fields updated)
NSF Add Fields	Note ID,Commit flag,Fields1...n	Status (fields updated)
NSF Attach file	NoteID, Filepath, Filename	Status
NSF Unpack file	NoteID, File, Filepath	Status
NSF Describe fields on form	Form name, List name, Field name	Status
NSF Find Forms	List name, Field	Status

Server Access

You can use the *NSF Open file* command to open any database normally available to the user. To determine the correct 'path' for the file, open the file in Notes and use the Synopsis... option to read the Path to that database.

NSF Make server path returns to OMNIS the best path to a given server and NSF file.

In order to access a database that is on a server through an API program, the user must have access to the server itself, otherwise Notes returns an error when you try to open the database. The open command returns integer values 0, 1, and 2. 0 means error, 1 indicates that the file is already open and is now the current file, and 2 indicates that the file is open for the first time.

```
NSF Make server path ('LANSERVE','Specs') returns NPATH
NSF Open Notes file (NPATH) returns #F
If flag false
   OK message {error opening note file. Specs}
End if
NSF Make Note ('SimpleDataForm') returns Note_ID
If not(Note_ID)
   ; error
End if
NSF Map fields ('DataList')
NSF Add Fields (Note_ID,'Commit') returns #F
If flag false
   OK message {Error adding fields to Note}
End if
Redraw (All windows)
OK message {Made [Note_ID]}
```

You can have a variable or field with the name Note_ID to store the note IDs as you create and update notes. The “Note_ID” name is case-insensitive.

The last opened data base is the *current* database, and reopening an open data base simply makes it current.

You must give the same path to the database NSF file each time.

Mapping Fields to Notes

Even when returning only one value from Notes, it is usual to load the result into a list. Normally it is the developer who sets up OMNIS variables with the same name as the fields in Notes. The *Find Forms* and *List fields on form* commands can automate this process.

Having set up the variables in OMNIS, define a list with the columns required:

```
Set current list List2
Define list {Note_ID,LastName,FirstName,PhoneNumber,Type
NSF Map Fields ('List2') returns #F
Calculate Type as 'Person'
;
; Class variable  Wtitle (Character)
; Class variable  LastName (Character)
; Class variable  FirstName (Character)
; Class variable  PhoneNumber (Character)
; Class variable  Type (Character)
```

The OMNIS field names *must* have the same names as the Lotus Notes field names, or you cannot get any data back.

Once you've set up the mapping, you can use *NSF Select* and *NSF Build view*. Some commands let you put individual data items into specific variables: this is slightly less efficient since the server needs to remap its variables and query the types again. Any command that you run repeatedly in your code should use *NSF Map fields* to set up the field mapping.

Views and Searching

NSF Build View lets you move the data from a view into an OMNIS list in a manner similar to *Build list from select table*. You can also search the primary index with a text key, either using a partial or full match on the index. The search is insensitive to diacritical marks. The first two parameters, the name of the view and the name of the list to hold the data, are required. As OMNIS opens each note, its fields are read into the CRB and added to the list. Thus, the last note found in the view is always loaded into the mapped variables. There is no way to prevent the values from being added to the list unless you were to redefine the columns of the list to be different to the map.

```
Set current list List2
Define list (Store long data)
   Note_ID,LastName,FirstName,PhoneNumber}
NSF Map Fields ('List2') returns Res
NSF Build View ('People','List2') returns Num
Redraw (All windows)
```

The *NSF Build View* command returns the number of notes found in the view in the specified return value. If you add a third parameter to the command, Notes searches for a matching value in the primary index for that view:

```
NSF Build View ('People','List2','Pon') returns Num
```

A fourth parameter Partial searches for a partial match beginning with the text value in parameter 3.

```
;  Beginning with 'P'
NSF Build View ('People','List2','P','Partial') returns Num
```

NSF Select scans all the notes in a database or files in a directory.

```
Set current list List1
Define list (Store long data)
   {PLAIN_TEXT,NUMBER,TIME_DATE,TEXT_LIST,RichStuff,Note_ID}
Clear list (All lists)
Set current list TEXT_LIST
Define list {S3}
NSF Select ('List1','@All') returns Found
For each line in list
   ; process list
End for
Redraw (All windows)
```

Error Handling

By default serious errors are reported with an OK message. If you define an error field with *Set error field*, OMNIS reports errors to this field and the command continues. Unfortunately, you can't track multiple errors in a single command once this option has been set, so you should use this option only once the application has been tested fully. Most commands return an integer value where 0 indicates an error.

```
NSF Set error field ('Error')
NSF Build view ('VIEW','LIST')
If flag false
   OK message {Error [Error]} ;; or call error routine to log it
End if
```

Creating and Deleting Notes

NSF Make note inserts a new note in the currently open file, sets its default form and returns the Note_ID to you:

```
NSF Make Note ('SimpleDataForm') returns Note_ID
OK message {Made [Note_ID]}
```

Once you have the Note_ID for the note you want to update, *NSF Add fields* writes new field values to the note:

```
NSF Add Fields (Note_ID,'Commit') returns Res
```

Adding the fields deletes and replace any fields that are already there. They set the flag and thus a text field bigger than 15K cannot be added and returns an error.

You can also specify a fieldname as a parameter, in which case OMNIS ignores the mapping and adds the value of the field directly to that field:

```
NSF Add fields (Note_ID,'NoCommit','Field1','Field2'
```

The “Commit” or “NoCommit” string controls the flushing of the note from the disk cache on the server. Any string other than “commit” gives better performance but less data security in case of a server failure.

Composite fields (RTF)

When reading RTF fields, OMNIS converts the RTF to plain text. To append a text value to an existing RTF field you can use *NSF Write Composite*.

```
NSF Write Composite (Note_ID,'Commit','RichStuff') returns Res
```

This command never uses the mapping and always appends the text in the OMNIS field to the composite Notes field with the same name. It converts the text using the default fonts and styles. There is no control over the style of the composite field.

Mail and Copying

The Lotus Notes mail system is a standard Notes database with fields for items such as To, From, CopyTo, and so on. Notes placed in the user's file with the correct fields are forwarded by the Notes Mail Gateway. You can insert a note directly into the mail and identify the mail server and the user's name:

```
NSF Where's my mail?  returns ServerName
NSF Who am I? returns MyName
```

NSF Copy Note lets you copy a note from the current database to a specified database. If the target database is not open, it is opened but not made current.

NSF Mail note copies a note to the mail file. After writing to the mail file, the data is not flushed to disk until the file is closed with *NSF Close file ('Mail_file').*

Once the Mail command is used for the first time, the database remains open until you close it.

Reading Data Dictionary Type Information

Forms in Notes contain a certain amount of data relating to the fields and their data types. There are two commands, *NSF Find Forms* and *NSF List fields on form* that you can use to read this information and create OMNIS windows on the fly. This example builds a list of forms, stripping out any aliases in the names.

```
Set current list FormList
Clear list
NSF Find forms ('FormList','Form')
For each line in list from 1 to #LN step 1
   If pos(';',lst(Form))
      Calculate FormList('Form',#L) as
        mid(lst(Form),1,pos(';',lst(Form))-1)
   End If
End For
```

This method builds a list of fields on the form:

```
NSF Describe fields on form (Form,'FieldsList','Field','Type')
```

Apple Events

Apple events is an event messaging system defined by Apple that allows applications, including OMNIS, to send commands and data to each other. For example, your current OMNIS library can launch a spreadsheet, open a document, and spellcheck it. Similarly, a Hypercard stack can send a method script to OMNIS, execute and print it.

Apple Event Groups

Apple events are divided by Apple into several categories, or 'suites', and this is followed closely by the OMNIS grouping; see Apple events in the command list. All applications that claim to handle Apple events must support the four 'required' events, Open Application, Open Documents, Print Documents and Quit. In addition to these compulsory events, OMNIS supports events in the following suites (OMNIS command in brackets, detailed in the OMNIS Help):

– Core events (*Send Core event)*

– Database suite (*Send Database event)*

– Finder suite (*Send Finder event)*

– Word Services suite (*Send Word Services event)*

Terminology

A number of Apple terms have been introduced into OMNIS when dealing with Apple events; in error messages, for example. However, some OMNIS terms have been used to replace System 7 terms, where these clash; the word 'attribute' is an example.

Initially, the client application starts sending Apple events in order to use the services of the server application. Since OMNIS can both send and receive Apple events, it can act as both client and server. Indeed, OMNIS can send Apple events to itself, and in this case is both

client and server. This is the default for many OMNIS commands using Apple events; for example, *Set event recipient* without a parameter sends events from OMNIS to itself.

– Source/target
  While the client is initially the source of Apple events, and the server the target, the server can, in fact, send its response to an event back to the client, in the form of a further event. When this happens the source and target are reversed.

– Recipient
  The recipient of an Apple event may be any object that can understand them; a named computer; a file in a server application; a field within that file. Note that OMNIS can be the recipient of its own events; it can also be the recipient of Apple events created by other applications, such as Hypercard.

– Recipient tag
  This is a parameter that allows OMNIS to store the path to a particular recipient with a name or tag, and so avoid constant re-prompting when using multiple recipients.

– Message/parameter
  Many commands include messages and parameters. For example:

```
Send DataBase event {Set Field ('THERE','HERE')}.
```

This OMNIS command is a member of the Send DataBase event group; the message 'Set Field' is selected from an option list, and the parameters are added, if required, in the Message box, including the plain brackets ().

Sending and Receiving Apple Events

This section discusses in general the way OMNIS handles Apple events as both client and server. The technique of sending and receiving events using procedural text is given later in the Scripts section.

Except for the four 'required' events, OMNIS only accepts Apple events if the *Enable receiving of Apple events* command has been run in the library that is to receive them. When opened, an OMNIS library has reception of Apple events disabled by default, and so *Enable receiving of Apple events* must be added to the library and run if all events are to be received. Once enabled in this way, *Disable receiving of Apple events* must be run within the library to return to the default. However, the four compulsory events are still accepted.

OMNIS must always respond to the four required Apple events, which may be created by another OMNIS library, or by another MacOS application. The actual way they are created depends on the client application.

When received by OMNIS the compulsory events do the following:

– the 'Open application' event launches OMNIS,

– the 'Close application' event quits OMNIS,

– the 'Open documents' event prompts the user to load a library or Ad hoc report,

– the 'Print documents' event prompts the user to open a library, and print a report or opens and prints the Ad hoc reports.

With reception enabled, OMNIS can respond to events in the following suites:

– Core events (usually sent from Finder to OMNIS),

– Database events.

– Error messages when receiving events

When an event is not accepted, the Apple message errAEEventNotHandled is returned to the sender.

There is a default timeout period when an event sender is waiting for acknowledgment. This period is set by Apple and is not altered by OMNIS; see Apple's *System 7 Reference* manual.

Numeric values are accepted as character strings and data conversion to numbers takes place.

A table is simply described as a collection of rows and columns in a database or spreadsheet. For OMNIS, this equates to the combination of an OMNIS file class and corresponding data file, and takes the name of the file class as a parameter. OMNIS keeps a table index (record pointer) for the table that is currently being used for database events in order that record (row) operations can be performed.

The *Send Database event {Use table ('TABLENAME)}* command must be issued with a valid table name (file class name for OMNIS) that is used for all subsequent record (row) operations. This OMNIS command sends the Does Field Exist event before setting the current active table to ensure that there is such a valid table, and also resets the 'table index' to point to the first record in that table. *Use Table* may also be used to reset the table index to the first record in a table.

The following example returns the type of the data in field CHARFIELD (Character, Boolean, etc.) and stores it in the OMNIS field DATATYPE.

```
Send DataBase event {Does field exist ('CHARFIELD')}
If flag true
   OK message (Sound bell) {Yes, CHARFIELD exists.}
Else If flag false
   OK message (Sound bell) {CHARFIELD not found.}
End If
Send DataBase event {Get field type ('CHARFIELD','DATATYPE')}
```

This example returns the size of the field CHARFIELD (character, Boolean, etc.) and stores it in the OMNIS field DATASIZE.

```
; mapping of Apple/OMNIS data types?
Send DataBase event {Get field type ('S5','DATATYPE')}
OK message {field [S5] is of type: [DATATYPE].}
Send DataBase event {Get field size ('CHARFIELD','DATASIZE')}
```

Get size CHARFIELD returns the size of the field/container in Bytes, e.g.

```
Local variable REMOVED DATASIZE (Character)
Send DataBase event {Get field size ('CHARFIELD','DATASIZE')}
OK message {field 'CHARFIELD' has size of: [DATASIZE]}
; The next command:
Send DataBase event {Set field ('S1', S5)}
```

Sets the value of the recipient field S1 to the value of the OMNIS variable S5:

```
; This example sets S1 from S5
Send DataBase event {Set Field ('S1', S5)}
Redraw (S1)
; and to get the value, use:
Send DataBase event {Get field ('S1','CHAR_FIELD')}
```

Gets the value of the recipient field S1 and returns it to the OMNIS variable CHAR_FIELD.

Similarly:

```
; Local variable Nets (Character)
; Local variable Message_date (Short date 1980)
Set event recipient {('DocsCI:Q4O:V2:Contractors')}
Send DataBase event {Get field ('NETWORKS','Nets')}
Send DataBase event {Get field ('DATE','Message_date')}
OK message {Number of networks is [Nets] on [Message_date]}
```

Note differences in use of quote marks in the *Get field* and *Set field* commands. If quotes are used for the second parameter of the Set command the actual string is sent; i.e., the characters 'S' and '5' in the example above.

The *Send Finder* commands let you send many standard events to the Finder. However, only two of the events can also be sent and received over a network. Both are compulsory commands and can be received by libraries even when **Disable receiving of Apple events** has been run. These two commands are

– *Send Finder event {Open Files*

– *Send Finder event {Print Files*

Sent without parameters, these commands prompt the user for filenames with a file open dialog.

Note that the command *Send Finder event {Empty Trash}* permanently removes deleted files. The “Are you sure?” warning is *not* given.

There is an error reporting mechanism within OMNIS which uses two hash variables #ERRCODE and #ERRTEXT; these allow target programs to pass back errors into your library and allow your methods to analyze errors in more detail than a simple 'flag false'.

Scripts

You can send a series of commands to a target program in the form of an executable script using the *Do script* commands, such as *Send Core event {Do Script (FIELDNAME)}* and *Send Core event returns ReturnField {Do Script (FIELDNAME)}*. OMNIS can send or receive scripts, and the format of the script depends on the particular program being used by the source.

The syntax for OMNIS scripts is defined simply by the form of the commands as displayed by OMNIS in the method editor.

A useful tip when creating scripts is to write and debug the method in OMNIS in the method editor, copy the required method lines to the clipboard, and paste into the appropriate field.

To split a command between two lines of script, you add '&' to the end of the line and continue the command after the carriage return character. When the script is read by OMNIS, the two lines are concatenated with a space in the position of the &. When each script is received by OMNIS, it is converted into method form and, if free from syntax errors, pushed onto the method stack and executed immediately.

This is a simple example written in Hypercard:

```
on mouseUp
   Send "OK message {Hypercard says Hi}" to program "OMNIS "
end mouseUp
```

Variables used in the script must be hash variables or % local variables.

To send this method:

```
Find first on CSEQ
Do method Total
Prepare for edit
Calculate %TOTAL as %TOTAL*17.5+%TOTAL
Update files
Next
```

you would type the commands into the field as they appear in the OMNIS method editor, one command per line, and send the value of the text field (LTEXT) to the target OMNIS program:

```
Set event recipient
Send Core event {Do script (LTEXT)}
; Sending to myself
```

This illustration assumes that you are sending events from OMNIS to Hypercard. The *Set event recipient* command is first used to see if the target program is already in the Application menu; if not, it is launched.

```
Set event recipient {Hypercard}
; Checks Application menu to see whether HC is running
If flag false
   Send Finder event {Open files}
   ; prompts the user to locate Hypercard, add it to the App
   ; Menu and open the Hypercard stack 'SalesStack'. Alternatively,
   ; the file path could be given as a parameter: Send Finder
   ; event {Open files ('MacOSHD:Apps:Hypercard:SalesStack')}
   ; in which case you could add the following condition:
   Set event recipient {Hypercard}
   If flag false
      OK message {Can't start Hypercard}
      Quit method flag false
   End If
End If
; with 'SalesStack'open, and Hypercard the recipient
; you can send a HyperText message
Send Core event {Do script ('Go Next')}
OK message {Error handling: Code is [#ERRCODE], text is [#ERRTEXT]}
; This has opened a stack and advanced one card
; Now publish a field on this card
Send Core event
   {Create publisher('Card Field TEXT','SalesStack-TEXT')}
; Second parameter not required here
; Next, send HC a value from OMNIS
Send Database event {Set field ('Card Field TEXT','OMNIS Text')}
; And finally, get a value back from HC
Send Core event returns S1 {Get data ('Card Field TEXT')}
```

Note the different uses made of the commands *Send Core event* and *Send Core event returns*. Clearly, the latter is capable of getting data from the target program. The data passing protocol uses either text or PICT and OMNIS uses its usual character-to-number conversion routine when receiving numbers from another program such as Hypercard.

If an event is sent which returns an error to OMNIS, the hash variables #ERRCODE and #ERRTEXT are used to store the error code and message. The flag is set if the event evokes a positive response (no errors are returned). OMNIS waits for a reply for the default timeout

period and, while waiting, allows the user to cancel by pressing Cmnd-period or **Cancel** if a working message with a **Cancel** button is visible.

If there is a syntax error when converting the text to a method, an error code of -1 together with an appropriate error string is returned in the reply message to the sender. If there are multiple errors, only the first is reported. If the OMNIS debugger is available, a method is still created and pushed on the method stack. A syntax error is replaced by a *Breakpoint* command containing the error text.

The debugger for a method created from a script is titled **External Script**; you can modify the method, but Save and Revert To Saved are not available.

If there are no syntax errors, the created method is pushed onto the method stack and immediately executed. A reply is not returned to the sender of the *Do Script event* until one of the following occurs:

– The created method (and all its called methods) have finished executing

– An error occurs when executing the method (or any of its called methods)

– A command which returns control to the user is encountered (such as *Enter data* or any of the *Prompt for* commands which open a non-modal window)

If an error occurs, an error code of -2 with the usual error string is returned in the reply message to the sender. If the debugger is available, the debugger is opened at the error in the usual way, but otherwise no error is reported to the OMNIS user.

Methods which display an OK message or other modal window do not count as returning control to the user, so the sender may be waiting for a reply for the period that the message is being displayed; this means that eventually the sender may receive a time-out reply to the *Do Script* message (the default time out period set by Apple is currently around a minute but the sender can override this).

If the script contains a *Quit method Returns Result* command, OMNIS returns this value to the caller as a keyDirectObject parameter in the return message. If the value is a picture, it is sent as typePict; else it is sent as typeChar (lists are converted into their tab-delimited forms). If, for example, there is an *Enter data* command in the script, the reply is returned to the sender when *Enter data* is encountered, and any return value which has been set at that stage is included in the reply.

You can use *Send Core event returns ReturnField {Do Script SCRIPTFIELD}* if the result of running a script returns a value (for example, sent to Hypercard), which is placed in the OMNIS ReturnField.

The following methods illustrate some simple uses of Apple Events

```
Set event recipient {'MagicMac'}
; Send to MagicMac already on my Application Menu
If flag false
   OK message {Error [#ERRTEXT], code [#ERRCODE]}
   Quit method flag false
End If
Send Finder event {Open files ('MagicMac:Sheet36')}
; This initializes MagicMac on my workstation
.. deal with errors etc.
Send Finder event {Print files} ;; prompts you for sheets to print
Send event {Create publisher ('Work')} ;; Publishes document 'Work'
Send event {Set data ('Spread1',LIST1)} ;; Sends LIST1 to
   spreadsheet
```

The following example method uses Apple events to open a remote library called 'Contractors' , which includes a file class (an OMNIS file class is a 'table' in Apple terminology) called 'Site details'. The local library has a field, S4, that is used to test the link, in particular the existence and value of the field CO_NAME (Company name) in 'Site details'.

If the script needs to use variables, as in this case, they must be ready-defined such as hash variables or % local variables.

```
; Open Library method, behind a pushbutton
On evClick
   Send Finder event {Open Files('DocsCI:Docs:V2:Contractors')}
   Prompt for event recipient {REM}
   Use event recipient {REM}
   Send DataBase event {Does table exist ('Site details')}
   If flag false
      OK message {No file class of that name}
      Quit method
   End If
   Send DataBase event {Use table ('Site details')}
   Send DataBase event {Does field exist ('CO_NAME')}
   If flag false
      OK message {No such field as CO_NAME}
      Quit method
   End If
   Send DataBase event {Get field ('CO_NAME','%S4')}
   OK message {CO_NAME IS [%S4]}
   Send DataBase event {Define Returns ('%S1','%S2','%S3')}
   Send DataBase event {Next}
   OK message {%S1 is [%S1], %S2 is [%S2], %S3 is [%S3]}
   Calculate %S1 as 'Holland & Duke Inc.'
   Calculate %S2 as 'The Old School House'
   Calculate %S3 as 'Benhall'
   ; Instead of calculating values you could use data entry windows
   Send DataBase event {Insert}
   ; Takes values in %S1, %S2 and %S3 and inserts new row/record
   If flag false
      OK message {Insert event failed}
   End If
   OK message {%S1 is [%S1], %S2 is [%S2], %S3 is [%S3]}
   Calculate %S1 as 'Beta Productions Plc'
   Calculate %S2 as '56 Lion's Road'
   Calculate %S3 as 'Ipswich'
   Send DataBase event {Update}
   ; Takes values in %S1, %S2 and %S3 and updates current record
   If flag false
      OK message {Update event failed}
   End If
   OK message {%S1 is [%S1], %S2 is [%S2], %S3 is [%S3]}
End If
```

Publish and Subscribe

Publish and Subscribe is a feature of MacOS that lets you make data available to other applications or access such data. A *publisher* in OMNIS is a field, list, or report whose contents you have made available to other MacOS applications.

When you publish a field or report, you create a file called an *edition* that stores the latest value of the object. If the edition is placed in a shared folder it becomes available to other users on your Network. When you publish an edition, you can specify a name for the edition. The default edition name is 'Document-Object name', for example 'MyLib-List1'.

A *subscriber* is an OMNIS field or list whose value is obtained from an edition. When a field is subscribed (that is, turned into a subscriber), an edition name is specified which could be in a shared folder of another Macintosh.

Edit Menu

The **Edit** menu contains three items that let you publish and subscribe OMNIS fields and reports: Create publisher, Subscribe to, and Publisher/Subscriber options (the latter changes from Publisher to Subscriber according to whether you tab into a field which has been published or into one which has been subscribed). When a field is published or subscribed, it is the value in the current record buffer which is used, not the value shown on the window. The value typed into a field is written to the buffer as the user tabs out of the field.

When fields (including lists) are published or subscribed, they are shown with a border when the user tabs or clicks into the field. The borders are gray and differ in darkness for publishers and subscribers.

Data is transferred as text or PICT. Lists are published as tab-delimited text and directly transfered into a spreadsheet or graph plotting package. You can easily publish one list and subscribe another list to the published one.

The **Open Editions List** item on the **Options** menu of the Debugger opens a method editor window listing all the current publishers and subscribers. The PROC column shows those editions which have been set up and are invisible to the user.

Publishers and subscribers are not canceled when the library terminates. The #EDITIONS system table class in the library stores a list of current publishers and subscribers.

To publish an entry field on a window

- Put your window into enter data mode, and **Tab** to or click in the required field
- While in the field, select **Create publisher** from the **Edit** menu

The edition name can be left at the default value.

- Select the **Publish** button

The field is shown with a 50% gray border when you tab into it. Next you need to set up the publisher options.

- Open the **Edit** menu and select **Publisher options** or double-click on the published field

The publisher options dialog lets you manually update the edition with the latest value of the field by clicking on the **Send Edition Now** button. Alternatively, you can select the **On Save** option which causes the field to be published when the value in the current record buffer is modified. In this case, that would be whenever you tab out of the entry field.

Lists are published in tab-delimited character format. The process is exactly the same as for an entry field except that you don't have to select enter data mode or tab to the field to select it. Simply click on the list field and select the **Create publisher** item. It is the data held in the underlying list data structure that is published; the list field displayed on the window is not relevant. The format for the published data is suitable for importing to a spreadsheet, that is, each row of the list is converted to text and forms a line of text in the edition, and each field is separated from the last by a tab character. For a list, **Publish On Save** means when you leave the field.

Reports are also published in tab-delimited character format. Each library can have only one report edition open at a time.

Subscribe to and **Subscriber options** on the **Edit** menu let you subscribe OMNIS fields to published data. The editions to which you can subscribe a field can be on the same workstation or on a shared network volume. The MacOS allows any user to share data with other users via the network.

To subscribe an entry field on a window

- Put your window into enter data mode and **Tab** to the required field
- Select **Subscribe to** from the **Edit** menu

The Subscribe to dialog lists all the available editions. As you click on each edition name, the contents of the edition is shown in the preview area. When you have located the data

- Select the **Subscribe** button

The field is shown with a 75% gray border when you tab into it. Next you need to set up the subscriber options.

- Open the **Edit** menu and select **Subscriber options** or double-click on the subscribed field

The subscriber options let you manually update the field with the latest value of the edition by clicking on the **Get Edition Now** button. Alternatively, you can select the **Automatically** option, which causes the field to be updated when the value in the edition is

modified. When there is a window open which contains subscribed fields, they are redrawn automatically when the field is updated.

Ctrl-clicking on a subscriber field is the same as selecting the **Open publisher** pushbutton on the subscriber options dialog window, that is, it launches the library responsible for publishing the edition, with the appropriate document. For example, where you have subscribed a character field to a word-processor document, Ctrl-clicking on the subscribed field opens the word-processor with the document as the top window.

Lists can be subscribed to an edition which uses tab-delimited character format. The process is exactly the same as for an entry field except that you don't have to select enter data mode or tab to the field to select it. Simply click on the list field and select the Subscribe to... option. It is the underlying list data structure that is subscribed; the order of the columns shown in the window are not relevant.

The data is read into the list in the same way as it would with a spreadsheet, that is, a line of text which forms a row in the list. If the fields used in the OMNIS list definition are not character fields, the text converts to the correct data type.

The evSent event is generated when a subscriber is updated. Using a simple task $control() method, you can check when a subscribed field changes:

```
On evSent
   OK Message {Subscriber [pChannelNumber] has been updated}
```

Publish and Subscribe Commands

The Publish and Subscribe group in the method editor contains the necessary commands for publishing and subscribing data. For example

```
Publish LIST1 {HD:Public Folder:MyApp-LIST1}
; Check flag etc after each command
Publish TOTAL1 {HD:Public Folder:MyApp-TOTAL}
; Subscribe a list
Subscribe LIST2 {FredsMac:Public Folder:Freds App-LIST1}
; Check flag etc.
Subscribe TOTAL2 {FredsMac:Public Folder:Freds App-TOTAL}
```

All the commands clear the flag and do nothing if your system software is earlier than System 7. For these commands, edition names can include a pathname (if a pathname is omitted, the usual searching rules apply). Field name lists have the same format as for the *Define list* command. Any publishers and subscribers set up by a command are invisible to the user. This means that no borders are shown for these fields on user defined windows and it is not possible to change their options from the Edit menu. However, it is possible to change the options for editions set up from the Edit menu using the commands. If a local variable is published or subscribed, then that edition is canceled as soon as the method terminates.

You can publish the current record buffer values manually using the *Publish now* command, or set up the automatic publisher and subscriber options, for example

```
Set publisher options (Publish on save) {LIST1,TOTAL1}
Set subscriber options (Subscribe automatically) {LIST2,TOTAL2}
```

When the current record buffer values of LIST1 and TOTAL1 change, the new values are written to the editions. This takes time and should be avoided unless you particularly want frequent updates to be made. Similarly, when the operating system informs OMNIS that the editions for subscribers LIST2 and TOTAL2 change, OMNIS updates the fields. The subscriber update is delayed if there is a design window on top, evSent is generated in the usual way and the task $control() method given earlier can be used to trap the incoming updates as before.

Various commands are available to control the automatic aspects of publish and subscribe.

```
Publish field C_CODE
Publish field C_LNAME
Publish field C_PICT
Publish now {FCUSTOMERS}
; Publishes text, tab-delimited
; and PICT fields in default edition names
; FCUSTOMERS means 'all fields in file FCUSTOMERS'

Set publisher options (Publish on save){FCUSTOMERS}
Calculate C_LNAME as 'Spratt'
; C_LNAME is published when the value changes
Disable automatic publications
; Turns off ALL auto-publishers
Cancel publisher
; With no field list this cancels all publishers
```

The command set for subscribers is exactly equivalent.

Publishing Reports

Any report can be printed to an edition file and shared by other applications. The *Send to publisher* command is part of the Select destination command group and directs all subsequent reports to an edition file for the library in tab-delimited form. An edition name is specified and if this is empty, the previous report publisher is used (or an edition named 'library name-report' is created if no previous publisher exists). The flag is set if a publisher is successfully set up. For example

```
Send to publisher {MyApp-Report}
Set report name RSales
Print report
Send to publisher
```

Creating your own Help

You can create your own help and incorporate it into your application for the benefit of your end-users. This section describes how you implement Context-sensitive help using the Help Project Manager in OMNIS.

Under Windows 95, you can implement What's This Help by adding a ? button to the title bar of some types of window using the **helpbutton** window property. The standard Help menu, available on the main menu bar in design mode, is not available in the runtime version of OMNIS. Therefore to include a Help menu in your application you need to create your own Help menu class that calls up your own help file.

The Help system in OMNIS Studio uses HTML files to store and display help text. To create help for your application, you must create your own help folder containing your HTML help files, as follows.

Creating the Help folder

Create a new folder inside the OMNIS\HELP folder. This folder should contain all the HTML files and sub-folders which make up the help for your application. You can create as many sub-folders as you like.

Creating your Help pages

Use a tool like Microsoft Front Page to create your help pages. Each help page must contain an HTML title, for example, <title>How to do x</title>, which OMNIS uses as the topic name in the help indexes and contents tree. As a basis for your help files, you may want to use the Gethelp.htm and Usehelp.htm files (found in the 'About Help' folder in the OMNIS help folder) which provide basic information about using OMNIS Help.

Creating a Contents Tree

The OMNIS Help displays the contents tree as an expandable and collapsible tree. You can create a contents tree for your help in one of two ways: create special HTML files that contain the contents information, or let OMNIS build a contents tree based on the exact folder structure in your help folder. With the latter you need to arrange your HTML files and sub-folders in the same hierarchy as you want to appear in your contents tree. OMNIS uses the folder names to populate the contents tree and sorts each node alphabetically, but with the second method you can replace folder names with longer descriptive titles.

Creating a contents tree from HTML Contents Files (Method 1)

In order to structure and control the ordering of a long contents tree you can create multiple HTML contents files with links to the sub contents. Contents file names must begin with "_C_", i.e. your main contents file might be called "_C_MAIN.HTM". Other sub-folders may contain other contents HTML files. Once you have created your help and contents pages your help tree may look like this:

MYHELP (folder)

_C_MAIN.HTM (Contains a title and links to all other contents pages)
WELCOME.HTM (An introductory help page)

USING (folder containing help on using your library)

_C_USING.HTM (Contains a title and links to help pages in folder)
PAGE1.HTM (A help page)
PAGE2.HTM (Another help page)

MAINT (folder containing help on maintaining data)

_C_MAINT.HTM (Contains a title and links to help pages in folder)
PAGE1.HTM (A help page)
PAGE2.HTM (Another help page)

There is no limit to the number of nested folders.

Creating a contents tree from your Folder Structure (Method 2)

If you choose *not* to create any contents pages, OMNIS will build a contents tree based on the tree of folders and sub-folders it finds in your main help folder. You can replace the folder names (usually limited to eight characters) by placing a special text file in each folder and sub-folder. The text file must be called _TITLES_.TXT and contain a list of real folder names and corresponding full names or text you want to appear in your contents tree. Your text file should look something like this:

:FOLDER1:Alternative title one
:FOLDER2:Alternative title two
:FOLDER3:Alternative title three

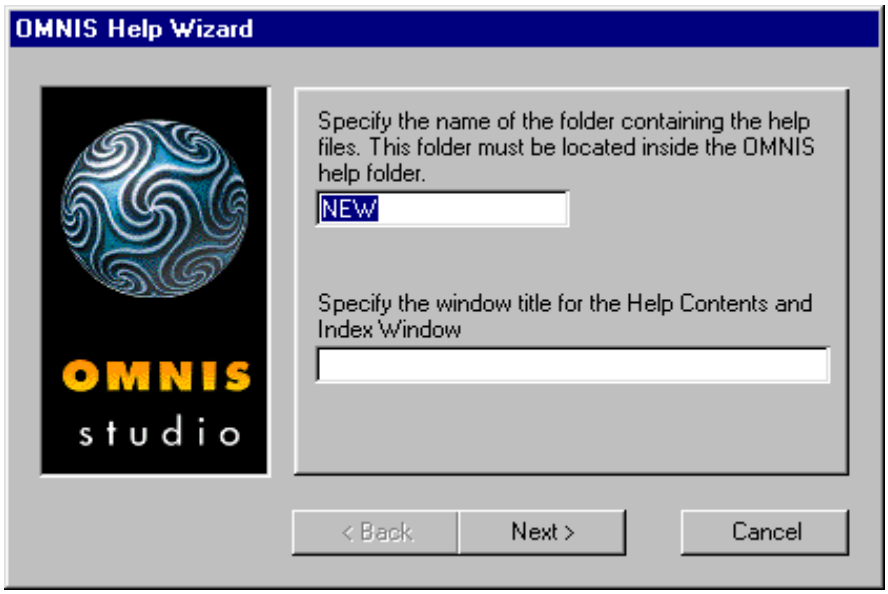
The folder name you want to override must be in ALL CAPS and surrounded by colons, followed by the title you want to appear. Each title must be terminated by a carriage return. The OMNIS Help system is created using the second method; examine the folder hierarchy under OMNIS\HELP\OMNIS and the _TITLES_.TXT files to see how it's done.

Creating your Help Project File

You create your Help using the Help Project Manager in OMNIS.

To create your Help project

- Select **Tools>>Help Project Manager** from the main OMNIS menubar, or click on the Help Project Manager button on the Tools toolbar
- Click on the **New Project** button in the Help Project Manager toolbar



- Enter the name of the folder containing your help files (this folder must be inside the OMNIS\HELP folder)
- Enter a window title for the Help window
- Click on Next and follow any further instructions

Bookmarks

The Topic pane in the OMNIS help window has a Bookmark menu that contains fixed bookmarks which link to specific HTML pages in the local Help system or on a website. You can add items to the Bookmark menu by specifying bookmarks in your help project. For example, the Bookmarks menu in the OMNIS Help contains links to pages on the OMNIS website. You can examine the OMNIS Help project file to see how you specify bookmarks. The Personal menu option can contain bookmarks entered by the user.

To create bookmarks

- Open your help project file in the Help Project Manager
- Click on the Bookmarks tab
- Enter the title or caption for the menu line from which your bookmarks will cascade
- Click in the bookmarks data grid to create an empty row for your first bookmark
- Enter the icon id, title, and so on, for the bookmark, and tab to create a new line

For each bookmark you must specify:

Icon id	the id of the icon for the menu line; the icon can be from the OMNISPIC.DF1, USERPIC.DF1, or the help library #ICONS file
Title	the title which appears in the Bookmarks menu
Is Help	True if the address specified in Address points to a help file in your help folder, or False for a full address to a web site
Address	the path to the HTML help page; for a help file in your help folder, include the name and partial path of the HTML file; for a web page enter the web address

Enabling Help in your Application

Having created your Help project, you need to set up the links in your library to your help pages. First, enter the name of your Help folder in the $helpfoldername property in your library preferences; this folder must be inside the OMNIS\HELP folder. Next, update the $helpfile property of each object for which you have written context help. The names you enter in the $helpfile property must include the names of all sub-folders under your main help folder, separated by forward slashes ‘/’, for example, MAINT/PAGE1.HTM.

Opening Help in your Application

You can open the OMNIS Help window from within your application using the $exechelp() method, which is a method under $root. It can take up to five optional parameters; the full syntax is:

```
Do $exechelp(cInstName,cWindowTitle,cHelpFolder,cDocumentName,cTopic)
```

where *cInstName* specifies the optional instance name of the help window; *cWindowTitle* specifies the optional window title; *cHelpFolder* specifies a help folder name, overriding the folder named in $helpfoldername; *cDocumentName* specifies the name and partial path of the help topic to be displayed, if empty help searches on the topic specified in *cTopic*; *cTopic* specifies the title or beginning of a topic title. If *cDocumentName* is empty and a topic title is specified, help attempts to locate the topic. If no topic is found, the help window enters the given text into the word search entry field and displays any topics found. If both *cDocumentName* and *cTopic* are empty, the contents list is displayed.

The following examples use the $exechelp() method.

```
Do $exechelp()
; displays help contents for the library

do $exechelp(‘’,’’,’’,’Reports/Printing.htm’)
; displays the specified help topic (note the extension .htm
; does not need to be specified)

Do $exechelp(‘’,’’,’’,’’,’Printing’)
; displays the topic whose title starts with or is equal
; to ‘Printing’. If non is found the word search tab is displayed
; with the word(s) entered and found topics are displayed

Do $exechelp(‘’,’’,’Docs’)
; displays the contents of the help in the folder docs inside
; the OMNIS help folder

Do $exechelp(‘’,’Alternative Title’)
; opens the help window of the current library with an
; alternative window title

Do $exechelp(‘instName’)
; opens a separate help window for the current library if
; a open window with that name does not exist
```

Chapter 10—OMNIS Data Files

This chapter describes how you access data in an OMNIS database using the OMNIS Data Manipulation Language, or DML. This language is specific to the OMNIS database, therefore it requires a greater understanding of the database structure than if you are using SQL.

The OMNIS database consists of one or more data files, and the OMNIS data manager handles all the data exchange with an OMNIS data file using the OMNIS DML.

Two OMNIS classes provide ways of structuring and accessing data files. File classes define the template for the types and lengths of the data fields that will be stored in an OMNIS data file. Search classes are used with OMNIS data whenever it is necessary to restrict the number of records to be used

File Classes

A file class defines the template for the types and lengths of the data fields that will be stored in an OMNIS data file, and is used exclusively with the OMNIS Data Manipulation Language. Once one or more file classes have been defined, a data file can be created and data, corresponding to the file classes placed in it.

You can define an unlimited number of file classes for a single library, each consisting of up to 400 fields.

Creating file classes is described in the *Using OMNIS Studio* manual. This section describes indexes and file modes.

Indexes and Keys

The **Index Options** button on the file class editor shows the index and key information.

For each indexed field, OMNIS maintains a table of record sequencing numbers (RSN) from which it can retrieve the indexed field in sorted order.

Consider the following data:

RSN	Code	Name	Description
1	C/100	Xerox 5305	Office copier
2	B/100	NEC 4X4c	CD changer
3	D/220	Tecra 700	Notebook PC
4	A/100	LaserJet 5L	Laser printer
5	H/100	Taxan 410LR	Color monitor

The Record Sequence Number is a unique number assigned to each record as it is inserted into the data file. If you index the fields for CODE and NAME, OMNIS maintains two tables of RSNs.

A *key* is the part of the field value used to sort the index. When you choose the indexed attribute for a character field, you can specify the number of characters, up to a maximum of seventy-four to use in the key.

For example, a character field that can have up to 255 characters may have an index that uses only six characters in the key. The following field values are therefore indexed with the same “Corpor” key value:

– Corporate Housing Ltd.

– Corporate Sales Inc.

– Corporate Technology

Using an exact match find to locate one of these records is now impossible, and the search string “Corporate Sales Inc.” will locate the “Corporate Housing Ltd.” record if it was the first record of the three in the file.

The unique index option prevents an index from having records with duplicate key values. When you execute the *Update files* command, OMNIS checks for a unique key value. This means that in multi-user mode, the data file is locked and safe against simultaneous inserts with the same key value. If you make an attempt to insert a new key value in an index or change an existing key value, and the new key value already exists in the index, then the *Update files* command stops and returns an error code, kErrUnqindex, which can be trapped with an error handler.

Case-insensitive indexes ignore the case of the characters in character and national data types. The key values for the following field values are the same:

– Hexadecimal

– HEXADECIMAL

– HeXaDeCiMaL

An exact match find on this index with a search string “HEXADECIMAL” finds any one of the three records, depending on the order in the data file.

You can create compound indexes with a key value derived from more than one field in the same file class. When in the file class editor, supply the names of the fields for the compound index in one of two ways:

- Check the **Indexed** option for a field in the file class and then select the **Index Options** button
- Select the **New Index** option in the **Modify** file class menu when the focus is in the list of indexes

You cannot use compound indexes with the standard **Find** button on a window because it allows you to enter only the value of one field. However, when you use the *Find* command in a method, and the compound index is the only index for the indexed field in the *Find*, OMNIS matches the values of the fields in the compound index against the index key when it carries out an exact match find.

If the only index for a given field is the compound index, OMNIS uses that index in all *Find* commands that specify that indexed field. If you specify two or more sort fields that happen to correspond to the fields used in a compound index, OMNIS uses the compound index to order the records.

Modify Menu

The file class editor **Modify** menu has the following options:

Set Connections opens the **Set Connections** window, where you can connect other file classes in the library to the current file class. To connect a file, double-click on the file class name when the dialog opens; an asterisk appears next to the file name. To deselect a file class, double-click again and the asterisk disappears. See the Setting Connections section later in this chapter for more on Connections.

Set File Mode determines how OMNIS opens the file: the default setting for a new file is Read/Write. You can also use method commands to change the file mode of a file within a library. There are four file modes:

– **Read/Write**
  you can read and write to the file
– **Read-only Files**
  you can read but not write to the file

You use this mode when you are only displaying or printing the data. OMNIS does not write the records in read-only files to disk when it executes an *Update files* command, and any attempt to delete a record in a read-only file causes an error. In multi-user libraries, a read-only file is not reread or locked when OMNIS executes a *Prepare for update*

command. It is good practice to make all files read only until they are required for editing. The method commands *Set read-only files* and *Set read/write files* change the file mode.

– **Memory-only files**
  builds the file as one or more slots in the CRB; the main file may not be a memory-only file

– **Closed Files**
  denies access to data held in a file to a particular level of user; all fields belonging to a closed file appear empty when displayed on the screen

Changing the mode of an existing read/write file to closed is equivalent to closing the file. OMNIS clears the CRB values for the file and releases the memory. If you reset the mode of the file back to read/write or read only, then OMNIS reopens the file and sets up a record buffer in the CRB when the file is next needed. If you edit a main file record whose parent file is closed, its connection to the parent record is not changed.

Estimate Disk Usage opens a window in which OMNIS calculates the amount of disk space required for a given number of records based on the specifications of the current file class.

If the file includes long character fields, pictures or other variable length fields, the estimated disk space is much less accurate.

Modifying a File Class

Before you store any data in a data file, you can change the field definitions in the file class as much as you want. Just reselect a line and enter any changes in the entry boxes. Once you have created a data file, you must undertake changes to the file class with care.

For example, do not attempt to tidy up any blank lines in the file class by moving fields into the gap, because data is referenced by field number not by name. The order in which the fields are entered does not matter, but OMNIS uses the field numbers to identify the data. You should not attempt to change the field numbers at any stage, even before you store any data. If you do, the names of the fields in calculations will change.

If you need to change the file class field definitions after data has been stored, the data file must be reorganized. This process is described later in this chapter.

Creating a Data File

You can create a data file from the Data File Browser.

To create a data file

- Select View>>Data File Browser from the main OMNIS menu bar
- Select Data File>>New

A standard file creation dialog appears, with the default file extension of .df1.

- Name your data file including the .DF1 extension

Reserving Space for a Data File

If a data file is allowed to grow as new records are added, new areas of the disk are allocated as they are needed. This can lead to fragmentation, where the various disk sectors making up the file do not follow one after another on the surface of the disk, and may give slower access times when searching and loading records. To prevent this, the data file should be created large enough for future use right from the beginning. To ensure this:

- For each File class, use the **Estimate Disk Usage** option in the **Modify** File class menu to estimate the disk storage for the maximum number of records
- Add all the estimates to obtain the overall file size requirement
- In the Data File Browser select **Data File>>Change Size** and supply the required information

If file fragmentation does occur and performance is suffering, you should consider running a proprietary defragmentation disk utility.

Search Classes

Searches are used with OMNIS data whenever it is necessary to restrict the number of records to be worked on; for example, to print all the client addresses in a certain area, all the salaries above a certain amount, all the video titles containing the word 'space', and so on.

Searches can also select the records read into the current record buffer when a window is used to edit data, or when a method is used to process data in the file directly. A department in a company, for instance, may only need to update certain records; access to the others may be restricted to the managers, and so on.

Searches are implemented using either the search class or the *Set search as calculation* command. You use these for searching both OMNIS data files, in conjunction with the *Find* commands, and lists.

Clear search class clears the selection of a search class so that all records are used in the print, find, next command, and so on.

Creating search classes from the Browser or Component Store is described in the *Using OMNIS Studio* manual. This section describes how you construct and use searches.

Creating Search Lines

You enter search lines into the main pane and select comparison operators from the toolbox at the bottom of the editor. Lines are added, building up a series of comparisons to test against each record in the file. If the record passes the tests, it is selected for display or printing in a report, and so on. If the record fails the test, it is passed over and the next one tested. A line of the search class can be a comparison, a calculation, or a logical AND or OR.

A comparison line consists of three elements: a comparison field, a comparison type such as equal to, and a comparison value. It takes the specified field in each record and compares it with the value entered in the comparison value box.

To enter a comparison

- Open your search class in the search editor
- Click or Tab to the Comparison field box

Enter a field name or select one from the Catalog by double-clicking on it If the Catalog is not visible, use F9/Cmnd-9 to display it.

- Click on an option in the comparison type box
- Tab to or click in the comparison value box and enter the text for the comparison

The characters typed into the comparison value box are read as a text string. Square brackets can be used to cause OMNIS to evaluate the contents of a field, for example, *[FIELD2]*. Quotation marks around text strings should not be used unless within square brackets, for example, *[con('ab',FIELD2)]*

- Press Enter when the comparison value has been entered

The search line displays the comparison line like this:

```
    FIELD   MODE    VALUE/CALCULATION
1   NAME    =       Smith
    FIELD   MODE    VALUE/CALCULATION
1   NAME    =       [#S1]
```

Comparison lines are dealt with by OMNIS as strings of characters, so you must be careful when comparing numeric, date, time and Boolean fields. OMNIS converts the strings to the same type as the field before performing the comparisons. Dates, times and Booleans can be abbreviated.

The **Edit** menu options **Cut**, **Copy**, **Paste** and **Clear** can be used on the selected line of the search class.

The **Modify** search menu has the following options:

- **Next Line**
  moves the selected line down one position and places the cursor in the field name box ready for data entry
- **Insert Line**
  inserts a blank line above the currently selected line and moves the subsequent lines down
- **Delete Line**
  deletes the highlighted line and moves the subsequent lines up
- **Clear Class**
  deletes all lines in the search class so that a new search can be designed

Numeric Fields

A numeric field in square bracket notation is always evaluated as a string, so that a comparison like this:

```
1   NUMBER_FIELD   =   [#1]
```

can fail because some of the information in #1 might be lost during the string conversion. For example, if #1=1.28, then [#1] would be evaluated as 1. If NUMBER_FIELD is a 2dp number, [#1D2] should be used to format the #1 string representation.

Boolean Fields

To test a Boolean field, the string values can be YES, NO or empty. Y, N and " can also be entered as comparison values; characters other than Y or N are converted to empty.

Searching on a Boolean field can produce unexpected results if you have not initialized the field to 'NO' or zero and you have used a check box for data entry. If the user does not click on the check box, then an empty value (not a 'NO') is stored in the data file.

Dates and Times

Date and time comparisons are carried out by converting the date/time to a string using the appropriate conversion string. Short dates are converted using the string held in #FD, short time fields use #FT, date and time fields use #FDT. For example, if the #FD date format string is 'm D CY', the short date field is converted to 'JUN 12 1994'. A search line of:

```
1   DATE_FIELD   Begins with   J
```

passes. But if a new #FD value is used such as 'D m Y', the search fails because the date converts to 12 JUN 94.

The following example shows the combined use of a comparison and calculations involving date and time functions.

	FIELD	MODE	VALUE/CALCULATION
1	DSTART	>=	11-05-94
2		CAL	DFINISH<=dat('11-06-94')
3		CAL	TFINISH<=tim('18:30')

Remember that you can obtain a complete list of all format strings for date and time fields by clicking on **Date and Time** in the **Hash** pane of the Catalog.

Calculations

A calculation line allows you to enter a calculation that will be evaluated to either zero (FALSE) or non-zero (TRUE). The outcome of the calculation decides whether a particular record passes the search condition.

To enter a calculation line:

- Click on the **Calculation** button in the **Mode** box
- Enter the calculation in the Calculation box

The two previous examples would be entered like this:

	FIELD	MODE	VALUE/CALCULATION
1		CAL	NAME=‘Smith’
	FIELD	MODE	VALUE/CALCULATION
1		CAL	NAME=#S1

For character strings such as 'Smith', quotes are necessary when they are entered in calculations but not in comparisons (see previous examples).

Calculation lines avoid the problems associated with the string conversion of comparison lines, but when comparing numbers stored to different precision, it is well worth rounding the fields with the *rnd()* function to make absolutely sure that values that look like 1.20, for example, are not really 1.200000456.

	FIELD	MODE	VALUE/CALCULATION
1		CAL	NUM_FLD=rnd(#1,2)
2		CAL	DATE_FLD=dat(#S1)
3		CAL	BOOL_FLD=1

Search calculations are optimized for query speed by choosing the best index based on the fields used in the calculation. Index optimization is discussed later in this section

Multiple Line Searches

When a search that makes more than one comparison or calculation is defined, multiple lines are used. For example, you may want to print records of your clients who live in New York or Washington and spend more than $10,000 per year on your products.

The search for this is:

	FIELD	MODE	VALUE/CALCULATION
1	TOWN	=	Washington
2		OR	
3	TOWN	=	New York
4		AND	
5	ACC_TOT	>	10000

To enter an AND or OR operator on a search line:

- Click on the blank line
- Select AND or OR from the **Mode** box

The evaluation of multi-line search logic is discussed in more detail later in this chapter.

Selecting and Using a Search

A search class is selected by the *Set search name* command. Once a search is selected (made *current)*, any reports that are printed from the standard OMNIS menus, or from methods with the *Use search* option will automatically use that search.

Using Search Classes in Methods

To use a search within a method, the search must first be made current with the *Set search name* command. In the following example, *Set search name* is included in the method before printing a report:

```
Set main file {F_COMMAND}
Set report name R_LIST1
Send to screen
Set search name S_LONDONERS
Print report (Use search)
```

Don't forget to click on the **Use search** check box when entering the *Print report* command.

In this method, the *Find first (Use search)* is used to select the first record for printing. The command *Next (Use search)* locates the next record that meets the search.

```
Set main file {F_COMMAND}
Set report name R_LIST1
Send to screen
Set search name {sMailing}
Prepare for print
Find first (Use search)
While flag true
   Print record
   Next (Use search)
End While
End print
```

Selecting Records and Using Find Tables

A *find table* is a set of records that is defined when you use the *Find*, *Find first*, *Find last*, and *Build list from file* commands. The *Find* commands define a find table and also load the first record in the table into the CRB. They must have a main file specified.

Single file find does not require a main file and does not set up a find table.

A find table always has a record order and this can be defined by specifying an indexed field, as in:

```
Set main file {CONTACTS}
Find first on NAME
```

or by using one or more sort fields:

```
Set main file {CONTACTS}
Set sort field NAME
Find first
```

A search class or calculation can be specified in order to filter the records in the find table:

```
Set search name {S_LONDONERS}
Find first on NAME (Use search)
```

This loads the first record that passes the S_LONDONERS search criteria.

Index Field and Find Table

Note that the order of the records defined by the *Find* command is defined by NAME, which is an indexed field and therefore has its own index table. The find table is in effect the NAME index table, and no further physical reordering of records is required.

Similarly, if a sort field defines the record ordering, and is an indexed field,

```
Set sort fields NAME
Set search name {S_LONDONERS}
Find first (Use search,Use sort)
```

the find table ordering is effectively the NAME index and no record loading or sorting takes place by choosing that index.

If OMNIS can find no suitable indexes, or if you specify more than one sort field, OMNIS has no choice but to read in the records to memory and physically sort them into the correct order. This takes time and memory, and always results in a working message dialog.

As a general guide, OMNIS will try to use the best index available unless you force it to use a particular index field in the *Find* or *Build list* command.

These are the rules that OMNIS uses:

– If there is a conflict between the record ordering defined by the index field and the sort field, the sort field takes precedence

– For files with more than 5000 records, OMNIS avoids physically loading and sorting the records in memory if it can find a suitable index in the correct order

– If the Main file contains less than 5000 records, if the time to sort the record set is not too great, and if there is an index field used in the search that avoids reading and filtering unnecessary records, it is used to read in the subset of records; the records are then physically sorted and stored as a physical table of records in RAM.

– If you define more than one sort field, records are read in using an index chosen by analyzing the search for the best index; the records, once read into memory, are then sorted and held in RAM as a physical table of records.

You can enter the same sort field twice to force OMNIS to use the index implied by the *Find* or *Search*.

Accessing the Find Table

Once a find table is created, the *Next* and *Previous* commands are used to load the records into the current record buffer. These commands should be used without any options if you want to use the existing find table:

```
Find first (Use search)  ;; first record is in the CRB
While flag true
   Add line to list
   Next
End While  ;; Flag cleared when no more records found
```

The flag will clear after the last record that matches the search has been read. If *Next* is used after the flag false condition, then you'll get the first record in the find table again.

If you use options such as an *Indexed field*, *Search*, or *Exact match* in a *Next* or *Previous* command and these are different from those used in the *Find*, the find table has to be rebuilt when the *Next* command is encountered. As a general rule, use the *Next* and *Previous* commands without any options if you want OMNIS to step through the current find table. The command *Clear find table* clears the table.

Note that OMNIS allows you to build one find table for each main file setting. If you have more than one open data file, each data file has its own current record buffer and find tables.

Search Calculations

The *Set search as calculation* command allows you to set up a search within a method without defining a search class:

```
Set search as calculation {mid(NAME,1,1)='B'} ;; select names
   starting with 'B'
Find first (Use search)
```

A search calculation can also be entered as part of a search class as described earlier and the following points apply to both types of calculation.

For direct access to OMNIS data files, OMNIS does not support a full query language along the lines of SQL-based systems. The concept of a find table is close to the SQL query such as this:

```
SELECT * FROM FCLIENTS WHERE FCLIENTS.TOWN = 'Washington'
```

which can be implemented in OMNIS as:

```
Set main file {FCLIENTS}
Clear main file
Find on TOWN (Exact match) {'Washington'}
```

The *Find* command builds a find table (in this case, based on the TOWN index) for which the TOWN index value is “Washington”. The first record in the table is loaded into the record buffer, and subsequent *Next* or *Previous* commands use the current table just as a *Fetch* would in SQL. When the end of the table is reached, the *Next* command sets the flag to false.

With direct access to OMNIS data, relational queries can be created using find tables. For example, the SQL query:

SELECT * FROM Authors, Publishers where Authors.city = Publishers.city

can be implemented in OMNIS using the *Enable relational finds* command and a simple search calculation:

```
Enable relational finds {PUBLISHERS,AUTHORS}
Set search as calculation {PUBCODE = AUTHCODE & AREACODE = 'X'}
Find first on PUBLISHER (Use search)
; Now print records
OK message {First record is [AUTHOR_NAME]}
```

The SQL ‘ORDER BY’ clause can be implemented using sort fields. Sorted tables are available with the *Find first* and *Find last* commands:

```
Set main file {AUTHORS}
Clear main file
Set search as calculation {ADVANCE > 5700}
Clear sort fields
Set sort field TYPE
Find first on A_CITY (Use search,Use sort)
If flag true
   Redraw (window1)
End If
```

Multi-line Search Logic

With long search classes using several lines, it is important to understand how the logical grouping is interpreted. The rules are:

1. Adjacent lines without AND or OR are treated as groups and within each group the lines are evaluated in the order in which the comparison fields appear in the file class
2. Within each group, the lines are assumed to be connected by AND unless they start with the same comparison field and a comparison type of equal to, begins or contains
3. Calculated lines are evaluated last in the group and are assumed to be connected by AND logic

The following examples show how multi-line search logic works:

```
CITY equal to `MIAMI' OR `NEW YORK'
```

	FIELD	MODE	VALUE/CALCULATION
1	CITY	=	MIAMI
2	CITY	=	NEW YORK

Because each field is the same and the comparison is Equal to, the connecting logic is OR.

```
TITLE equal to `Secretary' AND SALARY greater than $10,000
```

	FIELD	MODE	VALUE/CALCULATION
1	TITLE	=	Secretary
2	SALARY	>	10000

Because the comparison fields are different, you don't need to put in the AND between the lines.

```
CITY equal to MIAMI AND (TITLE=`MANAGER')
```

	FIELD	MODE	VALUE/CALCULATION
1	CITY	=	MIAMI
2		CAL	TITLE=`MANAGER'

The second search line is a calculation so that the connecting logic is assumed to be AND.

Indexes and Searches

When searches are used, an index may be available that makes the searching more efficient. This choice of index will override any previously selected index. Searches on non-indexed fields will force OMNIS to test every single record in the file.

Where reports and *Find* commands use a combination of sort fields and searches, OMNIS uses a simple set of rules to choose the most efficient index as described earlier.

If a *Find* or *Build list* command is given a specific indexed field, such as

```
Find on INDEX (Use search)
; or
Build list from file on INDEX (Use search)
```

OMNIS will use the index you specify to load in the records. However, if you add a sort field that implies an index that conflicts with the specified index, the sort field takes priority. By leaving the index field blank and with no sort fields, you indicate that the ordering is not important and OMNIS picks the index implied by the search.

Search Optimization

If a record fails any line of the search and the rest of the tests are linked by AND logic, the record is failed without further tests and the next record read in. For example, if most of the NAMES live in London this search is inefficient because a large number of records are tested against “Phibbs”.

```
NAME = Phibbs
TOWN = London
```

The search class is stored inside OMNIS as:

```
(NAME='Phibbs' & (TOWN='London'))
```

The innermost set of parentheses is evaluated first for each record.

The automatic sorting of fields within each group may conflict with the most efficient comparison order, so it is sometimes necessary to split the group up with a logical AND or OR. In this case, a more efficient search is:

```
TOWN = London
AND
NAME = Phibbs
```

The second ordering checks for the NAME value and then for each “Phibbs” entry, the TOWN comparison is made.

Similarly, if there is a group of lines connected by OR logic, as soon as one line passes, OMNIS passes over the rest until an AND connection is found before continuing to test the record. This intelligent interpretation is completely automatic and does not normally concern the library designer.

Choosing Indexes

When a search class is used, indexes can significantly speed up a search. Search calculations are optimized automatically but comparisons can be tuned for maximum efficiency by the developer. The first indexed field in the last group in the class is chosen as the index. In an earlier search example (which uses STATE, CITY, AREA), AREA is the indexed field. OMNIS steps through all the AREA codes of 100000 and carries out the other comparisons on each one. In this case, the search is optimized for cases where there are only a few AREA codes equal to 100000.

Search Calculations

Search calculations are optimized automatically by OMNIS provided that you do not override the optimizer by specifying an order that implies an index. The optimizer works by taking a sample of records using the index fields in the search and deciding which index is the most efficient.

In the following examples, the commands *Find first*, *Find*, *Build list from file* and *Print report* do not have any specified indexes. Thus OMNIS is able to use the built-in optimizer to choose an index.

```
Set sort field {TOWN}
Set sort field {AREACODE}
Set search as calculation {(NAME = 'Smith') & (TOWN = 'London')}
Find first (Use search,Use sort)
```

The two indexes NAME and TOWN are considered as suitable candidates for reading in the records depending on the relative numbers of records with 'Smith' and 'London'. If there were no indexes available, the sequence number index would have to be used, i.e. records are read in the order they were entered.

```
Set search as calculation {(NAME = 'Smith') & (TOWN = 'London')}
Clear main file
Find (Use search)
```

The two indexes NAME and TOWN are considered as suitable candidates for reading in the records depending on the relative numbers of records with “Smith” and “London”. The order of the records is undefined (since no sort field has been specified) and could be either TOWN or NAME order.

```
Set sort field {TOWN}
Set sort field {AREACODE}
Set search as calculation {(NAME = 'Smith') & (TOWN = 'London')}
Clear main file
Build list from file (Use search,Use sort)
```

The two indexes NAME and TOWN are considered as suitable candidates for reading in the records.

```
Clear sort fields
Set search as calculation {(NAME = 'Smith') & (TOWN = 'London')}
Clear main file
Print report (Use search)
```

The two indexes NAME and TOWN are considered as suitable candidates for reading in the records depending on the relative numbers of records with “Smith” and “London”. The order of the records printed is undefined and could be either TOWN or NAME order.

In the following examples, the optimizer is overridden by specifying an index in the Build list and Find commands:

```
Build list from file on NAME {Use search}
; NAME is used whatever the search
Find first on NAME {Use search,Use sort}
```

Compound and Case-Insensitive Indexes

Compound indexes are used by the optimizer whenever it seems most efficient. For example, if there is a compound index based on the concatenation of FIELD_A and FIELD_B, it is used in the following example to build the find table:

```
Set sort field FIELD_A (Descending)
Set sort field FIELD_B (Descending)
Build list from file (Use sort)
```

If the sorts were not both descending, clearly the index would be of no use and a sorted find table would be built in memory.

If there is a choice to be made between a case-sensitive and case-insensitive index, the case-sensitive one is used. If the only index for a field is case-insensitive and that field is chosen as the best index, any find table built will be in case-insensitive order.

Enter Data Mode

Enter data mode is the operating state used by OMNIS whenever an Insert or Edit function is selected. It places the cursor in the first entry field on the top window and allows the user to tab to or click in the entry fields and type in the data.

When a text field is selected for data entry, the cut and paste editing facilities in the **Edit** menu are available. Text can be pasted direct from a text file with the **Paste From File** option.

When the user selects a picture entry field, the cursor becomes a block and a picture can be pasted from the clipboard. Windows metafiles can be entered by selecting **Paste From File...** in the **Edit** menu.

Enter data does not terminate until either OK (the Enter key) or Cancel (Escape/Cmnd-period) is selected. If OMNIS detects an OK, the file is updated with the contents of the

current record buffer. If Cancel is detected, Enter data is terminated and the buffer cleared without affecting the data file.

Find operates in a similar way, i.e. the cursor is placed in the first indexed field, the user enters the data to be found and presses Enter or clicks OK. The data file is searched and the record which matches the entered data is displayed. If an exact match is not found, the next record in the index is displayed. If you want an 'Exact match only' Find, you must create a custom pushbutton or menu option, and use the *Prompted find (Exact match)* command.

Prepare for Update Mode

'Prepare for update mode' is a general term which encompasses *Prepare for insert*, *Prepare for edit*, and *Prepare for insert with current values*.

Prepare for update gets OMNIS ready to insert or edit/insert records in the data file. Without it, any calculations you make will never be committed to the OMNIS data file. The *Prepare for...* commands cause record locking of any read/write files in the current record buffer. Record locking for multi-user editing is dealt with later.

When a *Prepare for...* command is executed for a Read/write file, the current record may be automatically re-read to ensure that the value presented to the end-user is the most recent.

```
Calculate FIELD as 'Value'
Prepare for edit
; Do something such as Enter data
Update files if flag set
```

This may fail to store 'Value' in the data file if the file has been changed by another user. To correct this, move the *Calculate* command:

```
Prepare for edit
Calculate FIELD as 'Value'
; Call a routine for example
Update files if flag set
```

Enter Data Command

When a method is used to insert/edit data, the Enter data mode is initiated by *Enter data*. It passes control to the user so that data can be typed into fields on the window. When control is returned to the method, the flag is used to signal to the method whether OK or Cancel was used to terminate data entry. Clicking OK sets the flag, Cancel clears it.

The standard **Insert** function can be programmed using OMNIS commands, and is equivalent to:

```
Prepare for insert
Enter data
If flag false
   Clear main file
   Redraw (WindowName)
Else
   Update files
End If
```

The following example illustrates how the flag can be tested after an *Enter data* command has given control to the user for data entry. A simple window is selected in which three fields have been placed: an entry field for #S5, an OK pushbutton and a Cancel pushbutton.

The value entered into #S5 is used to search the letter file. If a match is found, a window is opened in which the new letter can be edited. The Repeat loop continues until the user selects No in the `Select another letter' Yes/No message box or clicks on the Cancel pushbutton in Enter data mode.

```
Choose letter
-------------
Set main file {F_LETTER}
Repeat
   Open window instance W_CHOOSE
   Enter data
   If flag true  ;; User has clicked OK
      Find on LETCODE {#S5}
      If flag true
         Call method 20 {edit letter}
      End If
   Else  ;; User has canceled in Enter data mode
      Close all windows
      Break to end of loop
   End If
   Yes/No message {Select another letter?}
Until flag false  ;; Cancel selected or No answer
Close all windows
```

Inserting Records

The simplest method for inserting a record is:

```
Set main file {FMAIN}
Open window instance W_FMAIN
Prepare for insert
Enter data
Update files if flag set
```

Prepare for insert clears the Main file buffer ready for the new data and prepares the system to insert data.

The action of clicking OK on the window or pressing Enter sets the flag and returns control to the method so that the *Update files* line can be executed. If the user cancels the Enter data, the Update is not executed and the record is not created.

If you want to enter a series of records, a loop could be used to reselect the insert mode automatically as each record is entered:

```
Set main file {FMAIN}
Open window instance W_FMAIN
Repeat
   Prepare for insert
   Enter data
   Update files if flag set
Until flag false
```

Editing Records

To edit records, the basic method is:

```
Set main file {FMAIN}
Open window instance W_EDIT
Prepare for edit
Enter data
If flag true
   Update files
Else
   Clear main file
   Redraw (WindowName)
End If
```

The *Enter data* command places the cursor in the first entry field in the top window and allows the displayed record to be edited. When an OK pushbutton or the Enter key is pressed, the *Update files* command writes the new data to the data file.

Modeless Data Entry

In a modeless window, as set by the window property $modelessdata, fields can be entered at all times. Editing or inserting a record can be implemented by setting a mode flag and trapping the button click event in a control method.

```
$control() method
----------
On evClick
   Switch wMode
      Case 'Editing'
         Do method Edit
      Case ‘Inserting’
         Do method Insert
   End Switch
```

Data Entry Windows

The *Window Classes* chapter in the *Using OMNIS Studio* manual describes the OMNIS forms in the Component Store. These let you generate data entry windows automatically from a file class, which contain the underlying code to view, insert, and edit your OMNIS data.

Setting Connections

Connections are used in OMNIS to relate records described by one File class to those in another. Before records in one file can be connected to records in another file, the extra indexed field must be added to the child file. This is done by setting the connection in the File class of the child file, as has already been described in this chapter.

Connecting the Records

To create the link between a child record and its connected parent record, the following must happen:

– The child record and its parent record must be read into the current record buffer.

– OMNIS must be placed in either *Prepare for Edit* or *Prepare for Insert* mode.

– The contents of the current record buffer must be saved to the disk, i.e. by selecting OK or pressing Enter when the **Commands** menu is used (or by executing the *Update files* method command).

Connections are changed only for the Main file during an edit. Consider an invoicing system where the two files are fCustomer and fInvoice: a window would be set up with

fields from both files, the customer's name field having the property $autofind**.** Once the customer details are entered, the normal method for generating an invoice is:

- Set the Main file to FINVOICE, i.e. the child file
- Select **Insert** from the **Commands** menu
- Enter the customer's name in the appropriate entry field

As the customer's name is entered, the automatic find attribute causes OMNIS to locate the appropriate customer record and read it into the current record buffer. Once the required customer details are displayed, the invoice can be generated:

- Type in the invoice amount and any other required details
- Click **OK** to save the invoice details, complete with the RSN (i.e. connection) of the appropriate customer record

In practice, the user would not be aware that files are connected; the records would appear to be all in the same file.

Once connected records have been stored, recalling, editing and printing them is very simple because of the way in which OMNIS makes use of the current record buffer:

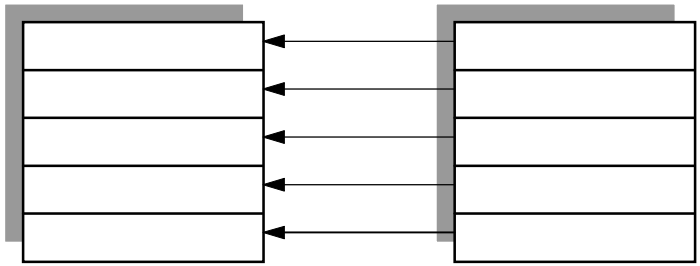
1. Whenever a Main file record is read into the current record buffer, any connected records are also read in. Consequently, to view fields in both the Main file and its connected files, you only need to open a window on which you have placed the required fields and then use **Next** or **Find** etc., to read in the records.
2. When OMNIS is placed in **Edit** mode, all the fields in the current record buffer can be edited. New records, however, can be inserted only into the Main file.
3. When a report is printed, connected records for the Main file are read into the current record buffer; fields in the connected file can be placed on the Report class just like fields from the Main file.
4. If you perform a Find using a field from a connected file, OMNIS searches for a matching record in the Main file.

Note that OMNIS does not automatically read in records from the parent of a parent file i.e. the `grandparent file'. To achieve this, the *Load connected records* command can be used for the parent file.

File Connection Schemes

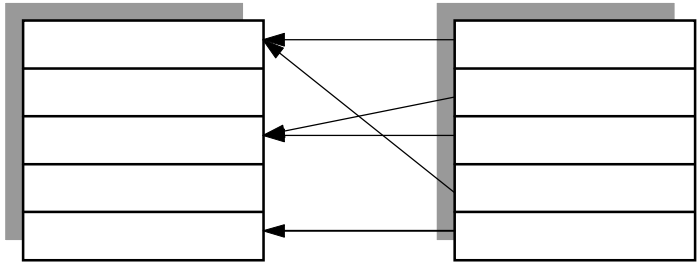
The OMNIS file connection scheme involves the addition of one invisible pointer to each record in the child file. Thus each record in this file can be linked to only one other record for each file connection. This system is ideal where one record in one file is related to just one record in another file, or where more than one record of one file relate to one record in another file. These two scenarios are known as `one-to-one' and `one-to-many' relationship respectively.

One-to-One



File A (parent) **File B** (child)

One-to-many



File A (parent) **File B** (child)

Every record in File B (the child file) is related to one record in File A (the parent file). For example, File A could be the departments in a company and File B the personnel. Clearly, a person can belong to only one department, while a department is likely to contain more than one person.

Many-to-Many Relationships

In a 'many-to-many' relationship, any record in file A can be related to more than one record in file B, and any record in file B can be related to more than one record in File A.

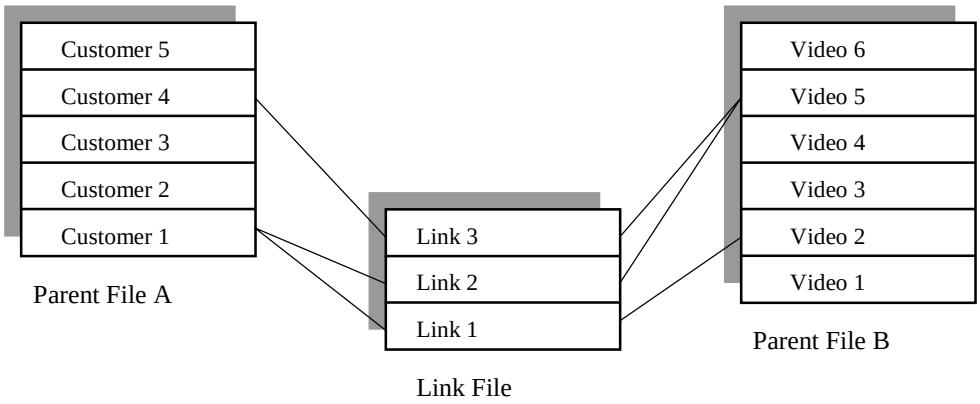
For example, consider a database system for use in a video rental outlet. There is a file of customers and a file of videos. Each customer can borrow more than one video and, over a period of time, will borrow many videos. Conversely, each video can be borrowed by any number of customers over time.

If the owner of the outlet only needs to keep track of whether a video is available for rent and to whom it has been rented, there is no need for the many-to-many relationship. However, if he wants to list, for each customer, all the videos borrowed, or for each video, all the customers to whom it has been rented, then a many-to-many relationship is necessary.

Link File

The many-to-many relationship can be implemented in OMNIS by using a third file known as a `Link' file. The link file is connected to both File A and File B, and has fields such as transaction date, number, time, etc. Each record in the link file connects one record from both parents, i.e. the two files must both be parents of the link file. They are not directly connected in any way. The link file need not have any fields at all, but usually stores the time and date for the transaction.

In the video rental outlet example, each record in the link file records one rental transaction, i.e. one customer rents one video.



Linking the Records

The Current Record Buffer is the key to understanding how the link file works. With the link file set as the Main file, reading a record in the link file automatically reads a record from each of the parent files into the current record buffer. A link file record must exist for each combination from the two other files.

To create a link between two parent records:

- Set the Main file to the link file
- Open a window containing at least three fields, one for each file. Two fields, one from each parent file, must be **auto find** fields with $autofind set
- Choose the **Insert** option from the **Commands** menu and enter values into both auto find fields to read in the two parent records in Enter data mode

- Enter the data for the Main file fields, e.g. rental date and rental transaction number
- Press **Enter** or click on an **OK** pushbutton to update the file

As the link file is updated, pointers to each parent record in the current record buffer are stored in link record.

In the video rental example, it may be desirable to print a report listing all the videos borrowed by each customer and sorted in customer name order.

Provided that the Main file is set to the link file, fields from both parents can be placed on the report. The primary and secondary sort fields can be set to the customer's surname and the video name respectively, and a report can be printed without the need for any autofind fields etc. Searches can be based on values from all three files. The *Enable Relational Finds* command can also be used to connect files, using relational joins or OMNIS connections.

Multi-user Data Access

The OMNIS database and specifically OMNIS data files are inherently multi-user. This section describes how you design a multi-user OMNIS database system.

To store an OMNIS data file you need a file server, complete with suitable network software installed and tested according to the supplier’s instructions. Depending on your requirements, you can put the OMNIS executable or program and your OMNIS library files on a server, or individual workstations. Best performance is obtained by storing the OMNIS program and library on a hard disk attached to the workstation. This means that the network is used only when data is transferred between the workstation's memory and the file server.

To maximize speed of access to the information on the server disk, you should store your data file on contiguous sectors of the disk. You can achieve this by formatting the disk immediately before installing the network software and creating the data file, or by running a proprietary defragmentation utility.

Sharing Data Files

All MacOS and networked volumes are sharable, but DOS files are sharable only if you run the SHARE command. OMNIS opens data files in shared mode by default, and there is no need to choose any special options. If you open a library or data file on a non-sharable volume, OMNIS opens the files in exclusive mode and the operating system prevents the files from being opened more than once.

While the Windows platforms use a three character file name extension to identify file types, the MacOS uses creator codes. Data and libraries created under Windows will not have the correct OMNIS Creator codes under MacOS, but the extensions .df1 and .lbs are recognized by the MacOS version of OMNIS. Similarly, if you do not use a .df1 (or .df2)

file name extension for a data file created under MacOS, it will not be recognized as a data file on a Windows platform.

So, to ensure that when you double-click on the file, you launch OMNIS, make certain you use the correct file name extensions.

You can access OMNIS data files under all platforms. If your mix of platforms includes Windows 3.1, you need to choose file names for shared data files with a maximum of eight characters plus the three character extension.

The maximum number of users is 451. As each OMNIS library opens a shared data file, its serial number is stored in a special block within the data file. Only one instance of each single-user serial number is allowed in at a time.

As each user is allowed to open the data file, a unique user number is allocated and stored in the #MU variable on that workstation. As each OMNIS workstation quits OMNIS, the #MU number is made available to the next user to open the data file.

DOS Share Configuration

OMNIS opens data files in Shared mode if they are on a networked volume or if the SHARE command is run on a DOS machine. If you have run SHARE and then open a library and data file on your local hard disk, OMNIS will log into the data file as if it were shared over a network. If a software failure causes Windows to terminate OMNIS without rolling back the locks in the data file, you may be forced to reboot before you can log back into the data file.

Record Locking and Semaphores

A locking system is used to protect records in a data file from being altered by more than one user at a time.

When a user attempts to change a record, a lock status indicator in the file is automatically checked before it can be edited. This indicator is known as a *semaphore*. If the record is not locked and the user starts changing it, the indicator is changed so that it indicates locked status, thus preventing another user from editing the same record. Semaphores are handled through standard calls to the operating system and are independent of networking software.

The normal locking system uses a page lock. This locking scheme is the most efficient but will lock every 500th record in a file. The Data File Browser menu option Slot>>Toggle Unique Locks provides a way of selecting either this page locking system or the unique row lock. The unique row lock method is slower but does not suffer from the condition where each lock affects a number of records in the file. You can change the option on a slot-by-slot basis, so that you can choose the optimum locking system for each file slot. You should use the default page locking system.

When another user attempts to edit a locked record, the mouse cursor changes into a padlock icon. Users can view locked records, but if an attempt is made to initiate a *Prepare*

for... command, the workstation is forced to wait until the lock is removed. Only the break key, Ctrl-Break or Cmnd-period releases the Prepare for edit/insert mode, clearing the flag and enabling the user to select another function. You can use the *Disable cancel test at loops* command to prevent users from canceling from a lock, but this is not recommended.

If you do not want the *Prepare for...* command to wait for the semaphore, you can first run the *Do not wait for semaphores* command. This causes the *Prepare for...* command to return a false flag to the method immediately if a record is locked. You can check the flag and present the user with an option to try again or do something else instead. For example

```
Do not wait for semaphores
Repeat
   Prepare for edit
   If flag false
      No/Yes message {The record is in use, do you want to quit?}
      If flag true
         Quit method
      End If
   End If
Until flag true
Enter data
Wait for semaphores
Update files if flag set
```

Disabling the Break/Cancel Key

The *Disable cancel test at loops* command prevents the user from canceling a *Prepare for edit/insert* command by pressing a break key. Provided that the library checks the flag after each *Prepare for* and prevents the subsequent *Update files* from being encountered, the *Working message (with Cancel)* provides an alternative way to withdraw from a lock.

File Locking

When the *Update files* command is encountered, any index tables that have been changed, data blocks added, deleted or changed are all written to the data file. While this is carried out, the header blocks in the data files are locked to prevent another workstation from doing the same. If you have a large number of users and indexes to sort, a large record length or a slow network, you will notice the lock cursor appearing for a short time if two users update the files at the same time. Normally these delays last at most a few seconds.

Index Caching

Index tables used to locate records in the OMNIS data file are held in memory on each workstation to improve performance over a network. OMNIS tracks the changes made to the index tables automatically and only re-reads the index from disk if it knows the one in

memory is out of date. This means that, as users log into the library, the index is altered more frequently and performance on the workstations is reduced.

Redrawing the Screen

If a record which is being viewed by a user is locked, you should design a method so that when the record is unlocked, the screen is redrawn and the latest version of that record is shown. If you do not redraw, even though the CRB is refreshed, the values on the screen are not brought up to date until the user tabs through the fields. The method is

```
Prepare for edit
If flag true
   Redraw (WindowName)
   Enter data
   Update files if flag set
End If
```

Constants Files and Global Data

In multi-user systems, record locking can cause serious delays if there is a record which needs to be continuously updated by a number of users, an invoice number or running total, for instance. If a user attempts to read and lock the record for the length of time it takes to enter a complete screen full of data, it could stay locked for 10 minutes or more! And if he goes for a coffee without releasing the invoice number record, the accounts department could grind to a halt.

In this circumstance, it is important for the record not to be read into the current record buffer and locked for any appreciable length of time. To avoid this, you should make the file read-only until it is ready to be updated. You can update it before the main update method is started (which can cause unused numbers to be created if a user Cancels half way through an Edit), or update it as the Enter key is pressed on completing the entry window. The command *Set read-only files* prevents OMNIS from locking a file even though it is in the current record buffer.

Set Read-only Files

You can designate a file as read-only to prevent OMNIS from locking it. Setting a file to Read/write affects the way that the local library deals with the file but changes nothing in the actual data file. You can use the *Set read/write files* command to return the file to read/write status and update the record. Note that you should not change the status of files while in *Prepare for edit/insert* mode unless you are sure that the record cannot be locked for any appreciable length of time.

In the following example, the invoice number is stored in a field cInvNum in file class fInvNumber. fInvNumber contains just one record, the current invoice number. In this method, it is initially given the read-only status. This prevents it from being locked by the

Prepare for insert command. When the details of the invoice have been entered, the invoice number is incremented and then stored in the fInvNumber file by changing its status back to Read/write. The *Update files* command terminates the *Prepare for...* mode and stores the new number.

```
Open window instance WINVOICES
Disable cancel test at loops
Set read-only files { fInvNumber }
Set main file {FINVHD}
Prepare for insert
Enter data
If flag true
   Update files
   Set read/write files { fInvNumber }
   Prepare for edit
   ; Assign the new number to the invoice
   Calculate InvNo as cInvNum+1
   Calculate cInvNum as InvNo
   ; Update both Invoice and constant file
   Update files
End If
```

An equally good approach would be to make **all** your file classes default to read-only status and then use a reversible block at the beginning of the method to set the required files to read/write status.

Using the #MU Variable

The #MU hash variable is the multi-user number, that is, it holds a number which identifies the user. Each user is given a unique number greater than zero as he selects the data file.

When users log off, their #MU numbers are reused, i.e. the next user to select the data file is assigned the first available number. In single-user mode, #MU is zero.

In situations where each user requires some scratch-pad fields in which to store temporary values, you can use the #MU hash variable to reference a corresponding local record, for example

```
; Find record on workstation number
Set main file {F_SCRATCH}
Find on USER (Exact match) {#MU}
If flag false ; No record set up for this user
   Prepare for insert
   Calculate USER as #MU
   Update files
End If
Prepare for edit
Calculate FS_TOTAL as FS_TOTAL+FS_TRAN_TOT
Update files
```

File Connections

When you are designing multi-user libraries which use connected files, you must take care to prevent excessive record locking of parent files which will be required by other users. Any file in the current record buffer will be locked, so in the sample method below, the PCODE record in the parent file P_FILE is made read-only.

Before the main record can be updated, the original parent file is made Read/write, but with the added complication that another user may have deleted that record! If this has happened, the user is forced to change the value of PCODE so that the Main file record can be connected to another parent record. For example:

```
Prepare for edit  ;; Waits for record release
Redraw windows (All windows)*** ;; ensures the changes on disk are
   displayed
Set read-only files {P_FILE}
Enter data
If flag true
   Set read/write files {P_FILE}
   ; Tests if Parent record exists before update
   Single file find on PCODE (Exact match)
   If flag true  ;; if parent record exists
      Redraw windows (All windows)
   Else  ;; parent record does not exist
      OK message {Code does not exist, please re-enter}
   End If
End If
Update files if flag set
```

Do not cancel pfu option

The *Update files* command has the option **Do not cancel pfu** which lets you write methods that maintain locks on parent records while multiple child records are inserted. Take the common example of an invoicing system: you insert an invoice record and then a number of related invoice items are added to a child file. When this option is used, the invoice record can remain locked until the whole transaction is complete. Here is an example method

```
Set main file {FINVOICES}
Set current list FLIST
Prepare for insert
Enter data
Update files if flag set (Do not cancel pfu)
; The parent is in place and remains locked. Now load items from a
   list and insert into the child file FITEMS
Set main file {FITEMS}
For each line in list
   Prepare for insert
   Load from list
   Update files (Do not cancel pfu)
End For
; At this point the parent remains locked
Update files
; Locks are released on all records
```

Note that *Prepare for...* commands cannot be nested; each one overrides the last. There is only one *Prepare for...* mode active at any time and, as each one is executed, it uses the Main file setting active at that time.

Unique Index Check

You can set the Unique index check property for a window field. In single-user mode, it prevents users from creating identical part numbers, invoice numbers, etc. It does this by checking that the value entered in the entry window does not duplicate a value in the corresponding fields on disk; the check is made as the user tabs out of the field (and before the Enter key is pressed). However, the check does not have the same effect when used in multi-user systems. In this case, two users may be entering values for that field at the same time so, although there is no duplicate on the disk as the value is entered, there will be duplication once the users have pressed Enter and the records are stored in the data file!

The Unique index property available in the file class definition is ideal for multi-user libraries and causes the check for uniqueness to be carried out on the indexed value before updating the file.

```
Load error handler $errorhand {Error handler}
; No range means all warnings and errors will be passed to
   $errorhand
Prepare for insert
Enter data
Update files if flag set

$errorhand
------------------
; Store error details before another error changes them
Calculate ERROR as #ERRTEXT
Calculate ERRNUM as #ERRCODE
If ERRNUM = kErrUnqindex
   OK message {Part number already exists, please re-enter part
   code}
   Enter data
   If flag true
      SEA repeat last command ***
   Else
      Quit all methods
   End if
End If
```

Connected Records

In some libraries you should consider the possibility that another user could alter the relationships between the records while waiting for a semaphore. For example, in a Video rental system, the record for the tape may be connected to a customer record and, while waiting for a semaphore on a video tape, another customer rents the tape and a connection or relational join is made to another customer record. If you use connections, you can add *Load connected records* to ensure that the latest record and its parent are displayed

```
Prepare for edit
If flag true
   Load connected records
   Redraw (WindowName)
   Enter data
   Update files if flag set
End If
```

One drawback of using the *Load connected records* command is the possibility of a 'deadly embrace' which is described below.

Semaphores

OMNIS makes use of additional semaphores to prevent conflicts when additional record locks are required and the system is already in *Prepare for edit/insert* mode. These semaphores are indicators set in the data file which inform other users that the record has been required for editing. Since OMNIS has already locked some records as the *Prepare for...* was issued, the additional locking requirements need to be carefully controlled by the developer, especially the potential problems associated by a user pressing a break key.

Deadly Embrace

Take the situation where two users Fred and Jim have locked some records with a *Prepare for...* command. While in *Prepare for...* mode, Jim attempts to find a record held by Fred and Fred tries to edit one of Jim's. Both users sit and wait for the padlock to go away, all day perhaps! If a user is forced to reset with the re-start button, semaphores are left on the original records and you have to wait for the file server to release them.

The simplest rule to follow is to avoid using commands in *Prepare for edit/insert* mode which read new records into the buffer for editing e.g. *Finds, Set read/write files, Load connected records* and also Automatic find fields in windows. If new records must be read in, update the files to cancel the *Prepare for...*, get the new records and issue a *Prepare for edit* to return to the former status. As each *Prepare for...* is encountered, the flag is checked and an orderly withdrawal from a lock is possible.

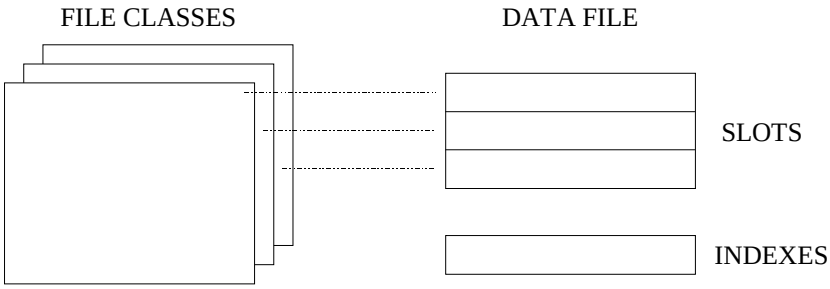
Clearing Semaphores

Semaphores are set in the data file via calls to the operating system. When the file server's operating system is restarted, they are all cleared automatically and the first OMNIS user to gain access to the file becomes #MU=1. If workstations are shut down unexpectedly, some file servers eventually clear semaphores automatically. If you are forced to break out of a lock by switching off a computer, and you have the maximum number of workstations, it may be necessary to wait for this to occur before logging back onto the data.

Data File Structure and Maintenance

In order to store and query data in the OMNIS data file, you only need to understand the general principles of database organization. If however, you are involved in database administration and management, you need a more detailed understanding of the underlying structure. This section describes the structure of OMNIS data files and the utilities available to maintain them.

OMNIS data files hold records of data whose structure you define in a file or schema class. Data associated with different file or schema classes can be held in a data file, each one occupying a *slot*. You can index some fields in a file class for faster retrieval so OMNIS also needs to keep the indexes for these fields.



You can access data in more than one data file at a time. Since the structure of the data stored in the data file is based on the framework defined in the file or schema class, it must be kept compatible by reorganizing it if structural changes are made to the class. You can also export and import data to and from OMNIS data files.

It is not recommended practice to use the same data file for both OMNIS SQL and DML access. In the remainder of this section, DML access and the use of the OMNIS file class is assumed for clarity of writing; the structure is the same, regardless of how you access your data.

Data File Structure

An OMNIS data file can store data for any number of file classes and has a maximum size of 3,840 MB. An OMNIS data file may comprise up to fifteen separate files, although OMNIS still treats these files as one continuous data file. The individual data files are called *segments*, each segment having a maximum size of 256 MB. OMNIS automatically expands the first segment of the data when it needs more space, by adding unused disk blocks to the file.

The data file is divided into blocks of 512 bytes each. There are five types of block in a data file:

– Data blocks

– Index blocks

– File header blocks

– Free blocks

– The master block

A block that stores the data for a database record is a data block. Depending on the size of an individual record, one data block may be large enough to store several records from a file, or OMNIS might need several data blocks to store just one record.

Index blocks store index values. You can index any combination of fields in the file class through the file class editor. OMNIS indexes the data values in a sorted tree structure. Every insertion, change, or deletion of a record rearranges the tree. For this reason, the more complex the index structure for a file class, the longer it takes to process insertions, deletions, or changes to data file records with that file class structure.

Free blocks are unused blocks in the data file. OMNIS keeps track of the free blocks in a free chain. Whenever OMNIS deletes a record from a specific file, it updates the free chain by marking the block or part-block that the deleted record occupied as being available for new records.

OMNIS maintains a file header block for each file class in the data file. File header blocks store information about the attributes of each field in the file. This lets OMNIS determine whether a particular data file needs reorganization. The file header block also maintains a pointer to the first block in the data chain for that file as well as a pointer to the first and last block of all indexes for that file, and other information.

Block 0 is the master block of the data file, which contains critical information about the file. It holds a total block count for its own segment of the data file and, if there is more than one segment in the data file, also holds information about the sizes of the other segments. Block 0 also stores a pointer to the first block in the data set free chain, a directory of all files in the data file, and a pointer to each of the file header blocks.

OMNIS has the concept of a *main file*; while you can open several files at the same time and edit their values, you can insert and delete records only in the main file.

Data File Reorganization

Reorganization is the process of converting the data file so that its structure reflects any changes that have been made to the data class(es) in the library. If a reorganization or reindexing operation is unsuccessful, the data file may be left in a damaged state.

When you make a change to a file class and close the design window, OMNIS automatically checks to see if reorganization is necessary and displays a prompt. You can turn off this prompt by setting $promptforreorg to kFalse in the OMNIS preferences, accessed from the Tools>>Options menu.

Reorganization is necessary if any of the following changes have been made to a file class:

– You have added or deleted indexes

– You have changed the field type of any field

– You have changed the file connections

Reorganization or reindexation is *not* necessary because of any of the following changes:

– New fields

– Changed field names

– Changed field lengths

Reorganizing the records for a file class in the data file creates a new file header block, clears the indexes and allocates all the complete blocks cleared to the free chain, converts the records to the new structure in place, transfers data to optimize it, removes the old file header, and constructs new indexes. Finally, OMNIS recreates the data dictionary entry for the file class.

You can reorganize a data file from the Data File Browser, described in the next section. You should ensure that you have adequate backups of the data file before attempting data file reorganization.

Maintaining Data Files

Data file maintenance is necessary to

– keep the structure compatible with its associated classes

– regularly check for and repair any data errors

The utilities for doing this are found in the Data File Browser.

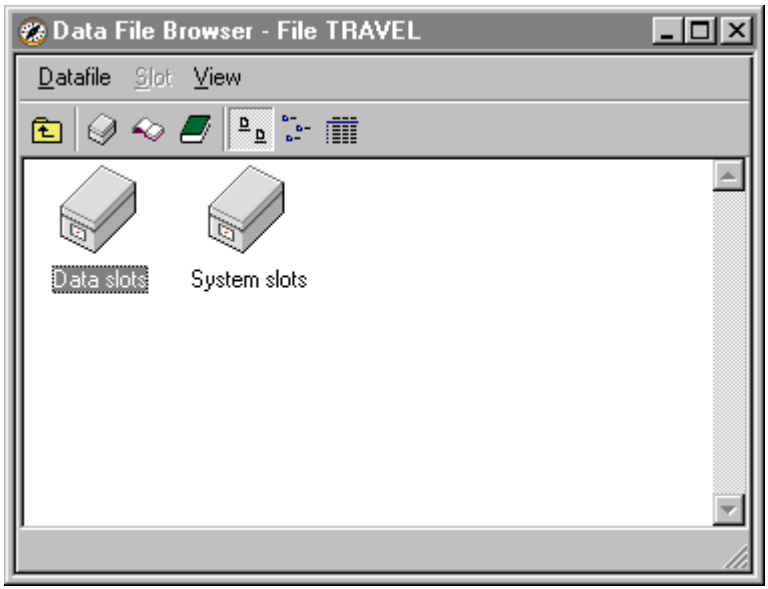
Data File Browser

You can use the Data File Browser to maintain your OMNIS data files. You open the Data File Browser from the main OMNIS menu bar.

To open the Data File Browser

- Select View>>Data File Browser from the main OMNIS menu bar

The Data File Browser lists the open data files. For a selected data file, the View>>Down One Level option shows System slots and Data slots.



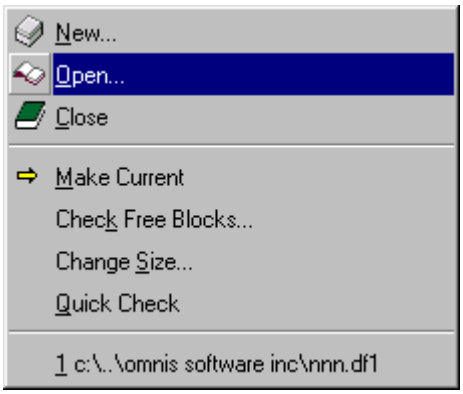
– **System slots**
  display #SLOTS and #INDEXES which are read-only and kept by the system.

– **Data slots**
  referenced internally by #SLOTS above, correspond to the file classes on which the data file is based. Viewing the next level lists all the slots in the current data file in a statistics table showing the number of fields, records, and related parameters.

File maintenance operations are carried out from the Datafile and Slot menus. OMNIS keeps a log of any damage to the data file in the **Check Data Log**, accessible from the View menu. This has options to clear the log and repair reported damage. You can select one or more slots to check by Ctrl/Cmnd-clicking in the list.

Datafile menu

The Datafile menu lets you open and close data files, and create new ones. The Datafile menu also contains



– **Make Current**
specifies the selected data file as the current data file

– **Check Free Blocks**
steps through the chain of free blocks and reports the number of free blocks: any corruption causes OMNIS to report an error

– **Change Size**
displays the Change Data File dialog, which lists the segments in the current data file; only the first segment of the data file expands automatically as you add records; you must size the other segments individually

– **Quick Check**
opens the Check Data Log that lists any damage in the data file

In the Change Data File dialog you increase the size of a segment by clicking in the **Segment size** field and typing in a new number of 512-byte blocks. You can add a new segment with **Add segment**. This displays the Select directory for new segment dialog, which lets you choose a directory for the segment. On Windows, the allowed directories are the directory of the first segment and any directories in the OMNIS environment string, a list of directories similar to the DOS path. OMNIS looks in all these directories for secondary data file segment files (DF2 and so on). Under MacOS, you can put the new segment in the OMNIS folder or in the root of any mounted volume.

You delete a segment other than the first one by selecting it in the list and clicking on **Delete segment**. This action is irreversible.

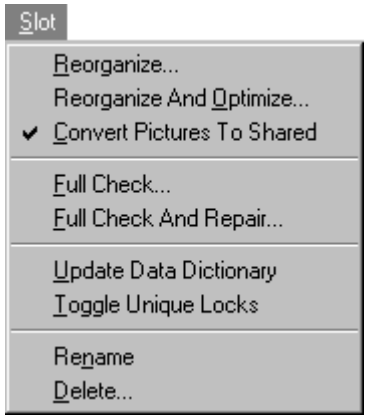
As the first data segment grows to accommodate additional data blocks, it can become *fragmented*: the sectors of the disk become non-contiguous because other files on the disk use sectors adjacent to the data file. This can cause slower disk access times because the read/write head must jump from one track to another to find records. This is particularly a problem in multi-user installations. The solutions are either to run a proprietary program to move all the data blocks onto contiguous sectors of the disk or to start with a recently formatted disk and create the data file at its maximum size before you delete any other files from the disk.

The Check Data Log that lists any damage in the data file. This reflects only the damage found as the data file was used since the last time you repaired the data, and unless your library has “Nexted” through each record in all the files, this may not reflect the true state of the data. OMNIS does not carry out any actual checking of the data file itself, and if

OMNIS finds no problems in the log, it displays the message “No damage found by quick check”. The log has three options, **Print**, **Clear**, and **Repair**. Choosing the **Clear** option results in a warning if there are any damaged blocks reported in the log. The **Repair** option attempts to correct the damaged blocks and enters a report of found or lost data in the log window.

Slot menu

The **Slot** menu has the options



– **Reorganize**
initiates a standard reorganization of the selected data file slots

– **Reorganize And Optimize**
reorganizes the selected data file elements and attempts to store the records that optimize the access times; unused space distributes among the data and data files can get larger during this process. Note that Reorganization is not a data recovery option and should be carried out only on good data

– **Convert Pictures to Shared Format**
converts any pictures stored in the data file to shared picture format

– **Full Check**
forces OMNIS to examine the internal consistency of the areas shared by all slots, such as the free space; any problems are reported in the Check Data log and can be repaired by selecting the Repair option in the repair log window

When you launch OMNIS, it clears the log. When you choose a full check, OMNIS writes out all reports of damage and messages about any repairs carried out to the log. The **Clear** option results in a warning if there are any damaged areas reported in the log, and you should normally back up the data and carry out the repairs before using the data file. Selecting **Repair** causes OMNIS to repair all the damage listed in the log and rebuild the indexes. Once begun, you cannot interrupt the process and a software crash or power failure can leave the data file in an unusable state so it is essential to back up before you start.

Signs of a damaged index include looping of records during report printing, missing records on a report, or missing records in Find, Auto find, Next, or Previous operations.

– **Full Check and Repair**
displays the Check Data Log as for the previous item and repairs any damage as part of the operation

– **Update Data Dictionary**
copies the data dictionary as stored in the current design library file class to the system slot held in the current data file

If the slot for the file class requires reorganization, OMNIS leaves unchanged the field types of the fields needing reorganization. If you make minor changes to a file class, such as altering field lengths and so on, OMNIS uses the field lengths from the file class when determining the number of characters that you can type into the field. However, OMNIS reads attributes that affect the way data is stored (such as field type) from the data dictionary.

– **Toggle Unique Locks**
  modifies the way in which multi-user locking is handled for that slot in the data file

The default, non-unique locking, is faster but has the disadvantage that each time you lock a record, you lock approximately one in every 500 records in the same slot. This is the most reliable locking mode. If you find it essential to use *row locking*, where the locks apply to only one record at a time, you can toggle this mode on a slot-by-slot basis. The locking mode for each slot is given in the Unique Locks column in the main browser pane.

– **Rename**
  renames the selected slot and creates an archive of the data. You can also use this to rename the file class in the data file when you rename it in the library, which lets you continue to use the data file without needing to recreate the data

– **Delete**
  deletes the selected slot

Shared Data Files

Some data file operations require that only one user is using the data file, and cannot be started if there are other current users; they are:

– Check free blocks, Reorganize, Reindexing

– Repairing data, Deleting or renaming a slot

Chapter 11—OMNIS SQL

This chapter describes how to use OMNIS SQL to access an OMNIS database, and assumes a working knowledge of SQL. A formal definition of the OMNIS SQL language is included.

The OMNIS Data Access Module (DAM), which is in the EXTERNAL folder, lets you connect to an OMNIS database using OMNIS SQL. No additional software is required to use OMNIS SQL.

Connecting to the Database

For an application to logon to a database in runtime requires a set of commands in a method. For OMNIS SQL all that is required is the command *Set hostname* to identify the database that contains the tables you want to access.

```
Set hostname { c:\omnis\df\my-dfile}
```

Sending SQL to the Database

Before a client application can get any data from a server, it must set up a corresponding place in OMNIS to hold the data. This involves mapping the structure of the data, including column names and data types. Typically, you do this using OMNIS schema classes. You can define a schema to include all columns of the server table or any subset of the columns.

The schema serves as a framework for defining a table class which, in turn, is the reference for defining lists and rows. OMNIS uses list and row variables for handling client/server data. Creating schema and table classes is described earlier in this manual, and using lists and rows is described in the *List Programming* chapter.

To send SQL to the database, you can either write your own methods, or use the table instance methods that handle both single row and bulk SQL transactions. This section covers custom methods; OMNIS table classes were described earlier.

OMNIS provides two different ways of building a SQL statement and sending it to the database: the *Perform SQL* command and SQL scripts.

Perform SQL

The *Perform SQL* command sends a single-line SQL statement to the current SQL session:

```
Perform SQL { SELECT name FROM Agent }
```

You can substitute text into the SQL statement using square bracket notation:

```
Perform SQL { SELECT name from [TABLENAME] }
```

where TABLENAME is an OMNIS variable.

SQL Scripts and the SQL Buffer

For longer statements that you may want to enter on more than one line, OMNIS provides the SQL script. OMNIS has a SQL buffer, an area of memory that contains a single SQL statement that you build up with a series of commands. *Begin SQL script* clears the buffer; *SQL:* enters a line of SQL text, *End SQL script* closes the buffer and *Execute SQL script* sends the contents of the buffer to the database. For example,

```
Begin SQL script
SQL: INSERT INTO Agent
SQL: (name,number)
SQL: VALUES ('FRED',123)
End SQL script
Execute SQL script
```

is equivalent to the *Perform SQL* command

```
Perform SQL {INSERT INTO Agent (name, number) VALUES ('FRED', 123)}
```

You can also use *Get SQL script {field name}* to copy the contents of the SQL buffer into a variable and *Set SQL script {field name}* or to copy the contents of a variable into the current SQL buffer.

Error Handling

Both *Perform SQL* and *Execute SQL script* clear the flag when the operation is not successful and the functions *sys(131)* and *sys(132)* report the error code and error text respectively.

```
Perform SQL { SELECT name FROM Agent }
If flag false
   OK message {SQL error [sys(131)] [sys(132)] }
End If
```

The Name Functions

There are several functions you can use to create part of a SQL statement. Each of these functions takes a file class name or field name list as a parameter and evaluates to a string of text that OMNIS inserts automatically into the SQL buffer. With these functions, you can write general-purpose methods that will work with any server without typing long SQL statements. For more information, see the descriptions in the OMNIS Help. The functions are:

– createnames()

– insertnames()

– selectnames()

– updatenames()

– wherenames()

Data Mapping

There are several ways to map OMNIS data into SQL statements.

Square Bracket Notation

SQL statements can contain square bracket notation, which OMNIS evaluates. If you use it you must supply quoted literals. For example, to update an Agent table by setting the name to a string from a variable called FIELD, you must quote the square bracket notation expression:

```
Perform SQL { UPDATE Agent SET name ='[FIELD]' }
```

You cannot use the string '[][]' (two sets of empty square braces) in your SQL statements because the DAMs use this string to mark the variables passed as @[]

Bind Variables

A *bind variable* is an OMNIS variable to which you want to refer in a SQL statement. Instead of expanding the expression, OMNIS binds, or associates the variable value with a SQL variable. If you place an @ before the opening square bracket, OMNIS evaluates the expression and passes the value to the server directly rather than substituting it into the SQL statement as text. You can also use this syntax to bind large fields such as pictures into a SQL statement:

```
Perform SQL { INSERT INTO Agent (agentPortrait) values (@[P_FIELD])
```

Never quote bind variables, and use them only to represent complete literals or values; otherwise you will get an error from the server.

Generally, using bind variables performs better than square bracket notation and is more flexible with respect to data representation. You should use square bracket notation only when the notation expression evaluates to a part of a SQL statement broader than just a value reference (such as an entire WHERE clause, for example) or where you know that simple value substitution is all you need. This works best for numeric data; strings tend to cause problems because of the issues with quoting. You must include quotes when using square bracket notation, but you don't need to when using bind variables. Also, if you are inserting NULL data into the database, you should use bind variables, since square bracket notation tends to insert empty strings into the SQL statement, not SQL nulls. This also applies to pictures, binary data, and very long text.

Select Tables and Cursors

A *select table* is a table of results that belongs to a session. When you send a SQL SELECT statement to the server and there is no error, the results of the SELECT become available to OMNIS as the select table for the current session. The select table can be empty; in this

case, the flag is true after the execution of the select and is only set to false when you attempt to fetch the first row after the end of the select table.

You can map the data in the select table into OMNIS data in three ways:

– *Declare cursor* and *Fetch next row*

– *Build list from select table*

– *Retrieve rows to file*

When you fetch data from the server, OMNIS converts data types between the native SQL server and the OMNIS data type if possible, including numeric precision. If there is a total mismatch between OMNIS field types and SQL column types, you can lose information or get a SQL error.

Declare cursor and Fetch next row

The *Declare cursor* and *Fetch next row* commands let you map each row in the select table into the CRB on a row-by-row basis.

Declare cursor defines a *SQL cursor*, a named pointer to a row in the select table, and associates a SQL select statement with the cursor. *Open cursor* opens the cursor, parses the SQL statement, binds input data, and executes the SQL statement. *Set current cursor* switches OMNIS to use the named cursor.

When you execute a SQL SELECT statement, the *current cursor* points to the first row in the resulting select table. When you *Fetch next row*, you fetch the row pointed to by the current cursor and move the cursor to the next row. You can have more than one cursor active at a time, letting you select rows based on values retrieved from a completely separate select table. You use the *Set current cursor* command to use a particular cursor as the current cursor with the *Fetch* commands.

Unless you are using multiple cursors, you don't need to explicitly open a cursor; OMNIS automatically opens one for you.

The *Fetch next row* command loads the column values for a single row of the select table into the OMNIS CRB fields.

If the list of fields does not match the columns in the select table, OMNIS tries to map the data as best it can. If there are more columns than fields, then OMNIS doesn't copy the extra column values into OMNIS variables. On the other hand, if there are more fields than columns, then OMNIS leaves the extra field values unchanged.

The usual retrieval process is to fetch the rows in a loop, one at a time, until there are no more rows in the select table. To do this, you use *Fetch next row* which fetches the row pointed to by the current cursor, then moves the cursor to the next row. After successfully fetching a row, the flag is true. After you fetch the last row, the next fetch returns a false flag and does nothing to the mapped fields. You then can use the *Close cursor* command to close a cursor explicitly, freeing the memory it uses.

Using the *Repeat* and *Until* commands with the flag lets you fetch until the flag turns false, though you must save the value of the flag in a separate variable for the test, since other commands may reset the flag before reaching the end of the loop. You can also use a *While* command, fetching the first row before entering the loop.

There are several variations on the *Fetch* command.

– *Fetch first row* fetches the first row in the select table and points the cursor at the second row

– *Fetch current row* fetches the current row and leaves the cursor pointing to that row

– *Fetch next row* fetches the current row and points the cursor to the following row

Build list from select table

The *Build list from select table* command fetches all the rows of the select table into a list that has been defined with the appropriate fields. The command appends the values rather than overwriting any values in the list so you can use this feature to put multiple select tables into a list, but there is a *Clear list* option to clear the list first. If you have defined the list with other fields or variables, the *Add CRB fields* option inserts these values to the list as well.

You can use #LM or $linemax to limit the size of the list regardless of the number of rows in the select table. There are also commands provided by most servers to limit number of rows returned; do not confuse these with the #LM value, which just affects the list.

The following method selects a table called **Contacts** with columns **name** and **number** directly into a list.

```
; Local variable lvContacts (list)
Set current list lvContacts
Define list { fContacts }
Perform SQL { SELECT name, number FROM Contacts }
; Creates the select table of all rows and columns
If flag true
   Build list from select table
End If
```

Retrieve Rows to File

This command copies the select table on a row by row basis into the current client import file, where the data is appended in tab-delimited format.

```
Set client import file name {my_file}
Open client import file
Perform SQL {Select * from my_table}
Retrieve rows to file
Close client import file
```

OMNIS SQL Language Definition

The following sections show the grammar of OMNIS SQL using BNF (Backus-Naur Form) diagrams, using the conventions from the ANSI standard.

Each statement includes a note specifying what parts, if any, of the statement depart from the ANSI 1989 standard for SQL.

SQL Statement

```
SQL_statement ::=
        create_table_statement
       | create_index_statement
       | delete_statement_searched
       | drop_index_statement
       | drop_table_statement
       | insert_statement
       | select_statement
       | update_statement_searched
       | update_statement_positioned
       | alter_table_statement
```

The SQL statement is the text that goes in the *Perform SQL* command or in a SQL script starting with *Begin SQL script*. The rest of the grammar depends on this main element.

ANSI SQL has the following statements that OMNIS does not implement. Most statement involve cursors, and OMNIS implements these as commands rather than as SQL statements.

- **close_statement**
  closes a cursor (see the *Close cursor*, *Quit cursor*, and *Reset cursors* commands)
- **commit_statement**
  commits a transaction (see the *Commit current session* command)
- **declare_cursor**
  declares a cursor (see the *Declare cursor* command)
- **delete_statement_positioned**
  deletes a row based on current cursor position
- **fetch_statement**
  fetches a row using the current cursor (see the *Fetch* commands)
- **open_statement**
  opens a cursor (see the *Open cursor* command)
- **create_schema_statement**
  creates a schema containing tables and views; OMNIS SQL does not support schemas
- **create_view_statement**
  creates a view; OMNIS SQL does not support views

– **grant_privilege**
grants an access privilege on an object to a user; OMNIS SQL does not implement any SQL security

CREATE TABLE

```
create_table_statement ::=
       CREATE TABLE table ( table_element_comma_list )
       CONNECTIONS ( table_comma_list )
```

The CONNECTIONS clause is an OMNIS extension to the ANSI standard that lets you specify a list of file classes to which to connect a file class. Connections are parent-child relationships between file classes. See the *OMNIS Data Files* chapter for information on connections

```
table_element ::= column_definition | UNIQUE ( column_comma_list )
```

You can define a file class using the SQL CREATE TABLE statement. The fields in the format come from the list of column definitions. You can also specify that the values for a group of columns are unique, taken together, with the UNIQUE constraint. You can have more than one UNIQUE constraint. All the columns in a UNIQUE constraint must be defined with the NOT NULL qualifier (see below).

The ANSI standard contains several other table constraints, namely PRIMARY KEY, FOREIGN KEY and CHECK that OMNIS SQL does not implement.

```
column_definition ::= column_data [ [ NOT ] NULL ]
```

The NOT NULL constraint specifies that when you insert a row, the value for this column must not be NULL.

The ANSI standard specifies a default clause that lets you define a default value for the column. It also lets you specify that the column is UNIQUE, REFERENCES a primary key in another table, or satisfies a CHECK constraint. OMNIS SQL does not implement any of these features.

```
column_data ::=
       column_name data_type
```

```
data_type ::=
       [ LONG ] VARBINARY
       | BIT
       | VARCHAR ( NUMBER )
       | CHAR ( NUMBER )
       | NATIONAL CHAR[ACTER] VARYING (NUMBER)
       | NCHAR VARYING ( NUMBER )
       | SEQUENCE_TYPE
       | DATE [ ( { 1900..1999 | 1980..2079
       | 2000..2099 } ) ]
       | TIME
       | TIMESTAMP
       | TINYINT
       | SMALLINT
       | INTEGER
       | NUMERIC ( number, integer)
       | DEC[IMAL] ( number, integer)
       | FLOAT_TYPE [ ( integer ) ]
       | REAL
       | LIST
       | PICTURE
```

ANSI data types include CHARACTER, NUMERIC, DECIMAL, INTEGER, INT, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION. OMNIS does not implement FLOAT and DOUBLE PRECISION directly, though FLOAT_TYPE is similar to FLOAT. The other data types are OMNIS specific. The integer value in the NUMERIC, DECIMAL, and FLOAT_TYPE types corresponds to the OMNIS subtypes for numbers; 0-8, 10, 12, and 14 are the possible values.

ALTER TABLE

```
alter_table_statement ::=
       ALTER TABLE table ADD
       { column_data | ( column_data_comma_list ) }
```

The ALTER TABLE statement lets you add a column to an already existing table using the same syntax as in CREATE TABLE.

The ALTER TABLE statement does not exist in the 1989 ANSI standard.

DROP TABLE

```
drop table statement ::=
       DROP TABLE table_name
```

The DROP TABLE statement removes a file slot and any data for that slot from an Omnis datafile.

The DROP TABLE statement does not exist in the 1989 ANSI standard.

CREATE INDEX

```
create_index_statement ::=
       CREATE [CASE SENSITIVE] [UNIQUE] INDEX index
       ON table ( index_column_comma_list )
```

```
index_column ::=
       column_reference [ ASC ]
```

The CREATE INDEX statement lets you create an index on an OMNIS database column. You can make the index UNIQUE, asserting that no two rows of the database have the same value for this combination of columns. You can also make the index CASE SENSITIVE, this will usually result in more efficient queries. The index column list contains columns from the table, and the table must already exist. You can also specify ASC on an individual column to sort it in ascending, as opposed to descending, order.

The CREATE INDEX statement does not exist in the 1989 ANSI standard.

DROP INDEX

```
drop_index_statement ::= DROP INDEX index
```

The DROP INDEX statement removes the named index, which must already exist.

The DROP INDEX statement does not exist in the 1989 ANSI standard.

SELECT

```
select_statement ::=
       SELECT [ ALL | DISTINCT ] { value_expression_comma_list | * }
       from_clause
       [ where_clause ]
       [ group_by_clause ]
       [order_by_clause ]
       [FOR UPDATE ]
```

The SELECT statement is the basic query statement in OMNIS SQL. It largely matches the ANSI standard, one exception being the having clause, which in OMNIS SQL is part of the group by clause instead of being a separate clause in the select statement. That is, in OMNIS SQL you cannot have a HAVING clause separate from the GROUP BY clause.

The FOR UPDATE clause initiates special locking for the records in the query. When you fetch a row from a cursor containing a SELECT statement with a FOR UPDATE clause, OMNIS locks the row for update. One of three things can then happen:

– You update the record with an UPDATE ... WHERE CURRENT OF cursor_name (see below), which on completion unlocks the row

– You fetch another row, which releases the lock on the previous row and locks the current one

– You terminate the transaction, which releases all locks

The order_by clause is separated out in ANSI SQL so that there is only one ordering for a query. Since OMNIS SQL does not have any set operators, such as UNION, there is no need to separate out the ordering clause.
The ANSI 1989 standard has no for_update clause. This comes from embedded SQL, the syntax there is FOR UPDATE OF column_name_list.

Value Expression

```
value_expression ::=
        term
       | value_expression { + | - } term

term ::=
       factor
       | term { * | / } factor

factor ::=
       [ { + | - } ] primary

primary ::=
        literal
       | column_reference
       | function_reference
       | ( value_expression )
```

A value expression is a key element of SQL that lets you calculate a value using an arithmetic expression language. You build an expression out of literal numbers and strings, references to columns, or parenthesized, nested expressions. You can combine expressions with any of the four arithmetic operators. The grammar above expresses the precedence relationships between the operators: unary + and - take precedence over * and /, all of which take precedence over binary + and -.

Column and Table References

```
column_reference ::=
        [ table . ] column_name

       | [ alias . ] column_name
```

The column name corresponds to a field in a file class.

```
table ::=
       [ library_name . ] table_name
```

The table name corresponds to a file class or to a table alias in the same SELECT statement, and the library name corresponds to a library. The table must belong to the library.

OMNIS SQL does not support the ANSI standard syntax alias.*, meaning all the columns from the table to which the alias refers. Also, if you use something other than a library name, or a name that OMNIS cannot recognize as a library name, you will get a syntax error.

Function Reference

```
function_reference ::=
        scalar_function
       | aggregate_function
```

A function reference is either a scalar function or an aggregate function. *Scalar functions* operate on each row of data in the select; *aggregate functions* operate on groups of rows.

The ANSI SQL standard has no scalar functions.

```
scalar_function ::=
      scalar_function_name ( value_expression_comma_list )
```

There are a number of scalar functions, summarized below.

Function	Purpose	Parameters
ABS	absolute value of a number	number
ACOS	angle in radians, the cosine of which is a specified number	number
ASCII	ASCII character corresponding to an integer between 0 and 255, inclusive	integer
ASIN	angle in radians whose sine is the specified number	number
ATAN	the angle in radians whose tangent is the specified number	number
ATAN2	the angle in radians whose tangent is one number divided by another number	number 1, number 2
CHARINDEX	the starting character position of one string in a second string	index string, source string
CHR	ASCII character corresponding to an integer between 0 and 255, inclusive	integer
COS	cosine of a number	number
TODAT	converts a date string or number to a date value using a format string	date string/number, format string
DIM	increments a date string by some number of months	date string, months
DTCY	a string containing the year and century of a date string	date string
DTD	a string containing the day part of a date string or a number representing the day of the month, depending on context	date string
DTM	a string containing the month part of a date string or a number representing the month of the year, depending on context	date string
DTW	a string containing the day of the week part of a date string or a number representing the day of the week, depending on context	date string
DTY	a string containing the year part of a date string or a number representing the year, depending on context	date string

Function	Purpose	Parameters
EXP	exponential value of a number	number
INITCAP	transforms string by capitalizing the initial letter of each word in the string and lower-casing every other letter	string
LENGTH	number of characters in a string	string
LOG	natural logarithm of a number	number
LOG10	base 10 logarithm of number	number
LOWER	transforms string by lower-casing all letters	string
MOD	modulus of a number given another number	number, modulo number
POWER	the value of a number raised to the power of another number	number, power
ROUND	rounds a number to an integer number of significant digits	number, significant digits
SIN	sine of a number	number
SQRT	square root of a number	number
STRING	concatenates some number of strings into a string	string[, string, ...]
SUBSTRING	extracts part of a string starting at a given index and moving a certain number of characters	string, start index, length
TAN	tangent of a number	number
UPPER	transforms a string by upper-casing all letters	string

```
aggregate function ::=
        COUNT(*)
       | aggregate function name ( DISTINCT column reference )
       | aggregate function name ( [ ALL ] value expression

aggregate_function_name ::=
       AVG | MAX | MIN | SUM | COUNT
```

There are some departures from the ANSI standard for DISTINCT aggregates: you can use only one such function in a given SQL statement, and you cannot use aggregate functions in expressions in a GROUP BY clause or WHERE clause.

FROM Clause

```
from_clause ::=
        FROM table_reference_comma_list

table_reference ::=
       table_name [ AS ] [ alias ]
```

The FROM clause lets you specify the table to input into the SQL statement. Multiple tables in the list indicate a join, and the WHERE clause specifies the join condition. Each table reference can have an optional alias that lets you refer to the table in other parts of the SQL statement by the alias. You can use this to abbreviate references to the table in the other clauses.

The ANSI standard does not have the optional AS keyword.

WHERE Clause

```
where_clause ::=
       WHERE search_condition

search_condition ::=
       boolean_term | search_condition OR boolean_term

boolean_term ::=
       boolean_factor | boolean_term AND boolean_factor

boolean_factor ::=
       [ NOT ] boolean_primary

boolean_primary ::=
       predicate | ( search_condition )
```

The WHERE clause lets you select a subset of the input rows using a logical predicate. The above grammar defines the precedence of the logical operators AND, OR, and NOT.

```
predicate ::=
        comparison_predicate
       | between_predicate
       | in_predicate
       | like_predicate
       | relation_predicate
       | null_predicate
```

The ANSI standard has, in addition to the above predicates, the quantified and exists predicates (nested selects), which OMNIS does not support. The relation_predicate is an OMNIS extension to the standard that lets you use OMNIS connections; see below.

```
comparison_predicate ::=
       value_expression comparison_operator value_expression

comparison_operator ::=
       < | > | = | <> | != | >= | <= | *= | =*
```

The standard comparison predicate involves one of the relational operators (greater than, less than, and so on).

ANSI SQL also allows you to use a nested select statement in place of the right-hand value_expression; OMNIS SQL does not support that. OMNIS adds the !=, *=, and =* operators (not equal, left outer join, and right outer join, respectively) to the ANSI standard operators.

An *outer join* is a join that includes all the rows in the tables regardless of the matching of the rows. The *= operator includes all rows from the table on the left that satisfy the rest of the WHERE clause. The =* operator includes all rows from the table on the right that satisfy the WHERE clause. Rows from the other table (right and left, respectively, contribute values if there is a match and NULLs if not. This syntax is similar to the SYBASE outer join syntax.

```
between_predicate ::=
       value_expression [ NOT ] BETWEEN value_expression AND
       value_expression

in_predicate ::=
       value_expression [ NOT ] IN ( literal_comma_list )
```

The ANSI standard lets you use a subquery (a nested select) as well as a literal list; OMNIS does not.

```
like_predicate ::=
       column_reference [ NOT ] LIKE literal
```

The ANSI standard adds an ESCAPE clause to the like_predicate to let you specify an escape character so you can match a % or _; OMNIS does not implement this.

```
null_predicate ::=
       column_reference IS [ NOT ] NULL

relation_predicate ::=
       { CHILD | PARENT } OF table
```

The relation_predicate lets you test the current row as being either a child or a parent of rows in the specified table. . See the *OMNIS Data Files* chapter for information on parent-child connection relationships

GROUP BY Clause

```
group_by_clause ::=
       GROUP BY column_reference_comma_list [ HAVING
       search_condition ]
```

The group_by_clause lets you group the input rows into groups according to a set of columns. The HAVING clause lets you select the groups, as opposed to the WHERE clause, which selects the rows going into the groups.

ANSI SQL has no ordering dependency between GROUP BY and HAVING, and you can have a HAVING clause without an accompanying GROUP BY. OMNIS does not allow this.

OMNIS SQL does not support the use of functions in a GROUP BY clause.

ORDER BY Clause

```
order_by_clause ::=
       ORDER BY order_column_comma_list

order_column ::=
       column_reference [ ASC | DESC ]
```

The order_by_clause lets you sort the output rows of the SQL statement using columns from the input tables.

The ANSI standard lets you sort by value_expressions in the select list by specifying the number of the expression; OMNIS does not.

INSERT

```
insert statement ::=
       INSERT INTO table [ ( column_reference_comma_list ) ]
       { VALUES ( insert_value_comma_list ) | select_statement }
```

The INSERT statement inserts rows into an OMNIS table. The first list of columns names the columns you are creating; this exists to let you reorder the list to match your list of values or select statement.

There are two alternative ways to supply values to the INSERT statement. You can supply actual values through a VALUES clause that contains a list of values, or you can give a SELECT statement that creates a table of data matching the insert list. See the *SELECT* statement section above for details on SELECT.

```
insert_value ::=
       literal | NULL
```

An insert value is a literal value or the NULL value specified by the string “NULL”.

UPDATE

```
update_statement_searched ::=
       UPDATE table SET assignment_comma_list [ where_clause ]

assignment ::=
       column_reference = { value_expression | NULL }
```

The searched update statement updates all rows that satisfy the predicate in the WHERE clause by assigning the indicated value or NULL to the column.

OMNIS SQL will let you preface the column name in the assignment with the library and table names, which extends the ANSI standard. There is no need to specify the additional names, but you can do so for clarity if you wish. Specifying a table other than the table in the UPDATE table clause, generates an error.

```
update_statement_positioned ::=
       UPDATE table SET assignment_comma_list
       WHERE CURRENT OF cursor
```

The positioned update statement updates the current row, the row to which the current cursor points. See the description of the *Declare cursor* command in the OMNIS Help. The WHERE CURRENT OF cursor clause works with the SELECT ... FOR UPDATE statement to update rows locked for update.

DELETE

```
delete_statement_searched ::=
       DELETE FROM table [ where_clause ]
```

The DELETE statement deletes rows from the OMNIS database based on the predicate in the WHERE clause. OMNIS deletes all rows that satisfy the predicate.

Chapter 12—SQL Browser

This chapter describes how you setup the OMNIS DAMs and create and modify sessions using the SQL Browser. It assumes you have the correct access rights to log on to and create tables on your server database.

Setting up the DAMs

OMNIS Studio installs the DAMs into your EXTERNAL folder under the main OMNIS folder. Each DAM provides the interface between OMNIS and a particular server database, sending SQL instructions and getting back data. You must make sure that no other copies of a DAM are located in any OMNIS folder or subdirectory on your local machine, regardless of the filename.

Direct DAMs

The vendor *application programming interface* (API) provides a direct connection to a particular type of server database. DAMs that use these interfaces are called *direct* DAMs since they access the DBMS directly through the API.

The OMNIS Data Access Manager provides the following direct DAMs

– **Oracle**
connects to ORACLE with full use of PL/SQL under Windows, and MacOS

– **DB2**
connects to IBM’s DB2 Universal Database (UDB) under Windows, and MacOS

– **Sybase**
connects to Sybase under Windows, and MacOS

– **Informix**
connects to INFORMIX under Windows, and MacOS

Middleware DAMs

Middleware is software that provides a common API for several different relational database managers, as well as non-relational file systems. The middleware accepts standard SQL and translates it into or passes it through to the underlying data access mechanism.

The OMNIS Data Access Manager provides the following middleware DAMs:

– **ODBC**
Microsoft’s ODBC (Open Database Connectivity) lets you write database- or server-

independent client applications; it connects to many different databases under Windows, and MacOS; the ODBC middleware is available from many different vendors including Intersolv and Visigenic Software

– **EDA/SQL**
Information Builders Inc's EDA (Enterprise Data Access) connects to a variety of servers under Windows only

Client and Network Software

Every DAM requires specific client and networking software to run. The specific version of the client software supported for each DAM is documented on the OMNIS website.

ORACLE

The Oracle DAM connects to both local and remote Oracle databases. You will need either local Oracle or the SQL*Net driver software for your network configuration.

DB2

The DB2 DAM connects to IBM's DB2 Universal Database (UDB). Your System or Database Administrator needs to install the DB2 Universal Database on a suitable server, and all clients need to install the DB2 Client Application Enabler appropriate to their operating system, all available from IBM and documented in IBM's *Quick Beginnings* manual. You can use the Client Configuration Assistant to enable access to your database.

INFORMIX

To use the Informix DAM you will need a version of Informix-Net. Under MacOS, check that your services file contains an entry for the Informix Server, since this does not happen automatically.

SYBASE

The Sybase DAM connects to the Sybase server database and requires the OpenClient software.

ODBC

The ODBC DAM connects to many different databases and file systems using the appropriate client ODBC drivers. You must use an ODBC driver with an API that conforms to the core or minimal ODBC API conformance level. The driver must also support the following level 1 ODBC API extensions:

– SQLColumns

– SQLDriverConnect

– SQLGetData

– SQLGetInfo

– SQLGetTypeInfo
– SQLStatistics
– SQLTables
– SQLSetConnectOption
– SQLGetFunctions
– SQLParamData
– SQLPutData

and the Level 2 functions, if you wish to make use of the BatchSize functionality.

– SQLExtended Fetch
– SQLSetScrollOptions

The ODBC driver should also conform to the core SQL language. You can use drivers that conform to the more limited minimal SQL language conformance level, but this limits the functionality available to you.

EDA

The EDA DAM connects to a variety of databases using Enterprise Data Access client and server software from Information Builders, Inc. EDA/SQL is a gateway product connecting to over 70 different databases and file systems, supporting many different network protocols and operating systems.

Platform Specific Issues

Some additional information is given here in case of problems on specific platforms.

Windows NT

If your connect protocol is Named Pipes or IPX/SPX you may have problems retrieving multi-column data that includes a binary picture field, if connecting to Microsoft SQL Server 6.0. Data corruption can occur when all characters of the server column are filled, that is, the last character of the text is replaced with a garbage character. For example, "Dumbo" would come back as "Dumb|", the last character being "|", the vertical bar character. The Picture field is retrieved intact and displays correctly when viewed. If the binary picture field is not included in the data retrieval, this type of corruption does not occur.

This problem only occurs if the connect protocol is Named Pipes or IPX/SPX, which are not supported or tested. TCP/IP will not cause this problem, so ask your Database Administrator to modify the MS-SQL 6.0 Server to allow connections via TCP/IP. To do this:

- Open the SQL-Server 6.0 tools on the client machine

- Open the Client Configuration Utility and choose **Advanced**
- Select the server name and change the DLL to TCP/IP; add the IP address and port number of the server in the connect string field
- Click on **Add Modify**, then **Done**

MacOS

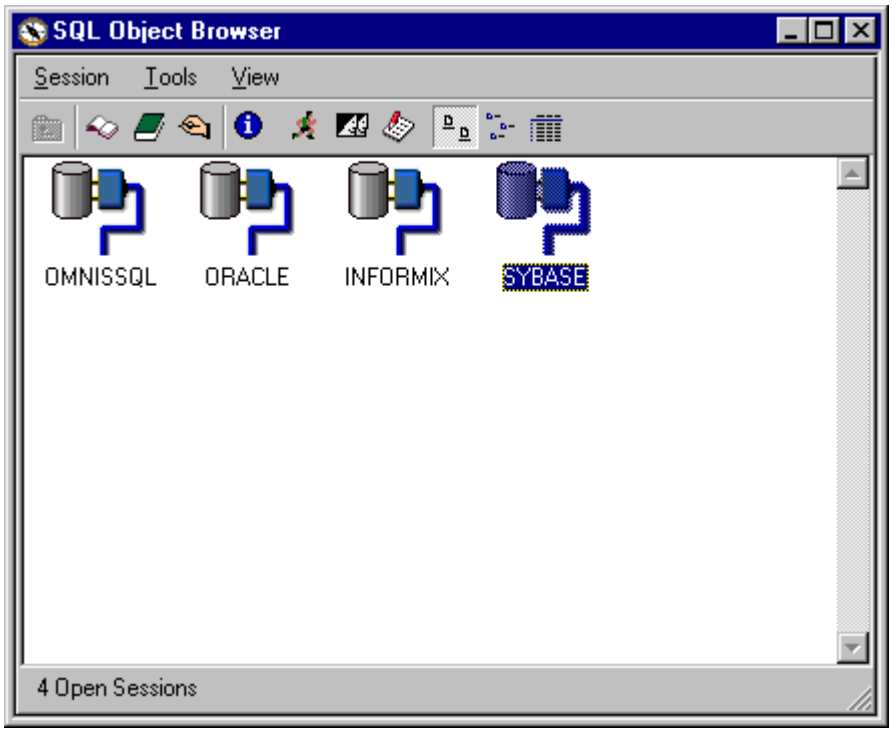
If you are using a 68K Macintosh, for ODBC and Sybase you need to be running the Apple Shared Library Manager (ASLM). The Code Fragment Manager must be on the PowerMac for all DAMs.

Sessions

The process of connecting and logging on to your server database creates a *session* in OMNIS. You can log on and create sessions using the *SQL Browser*, available under the Tools menu on the main OMNIS menu bar.

To open the SQL Browser

- Select the Tools>>SQL Browser option on the main OMNIS menu bar



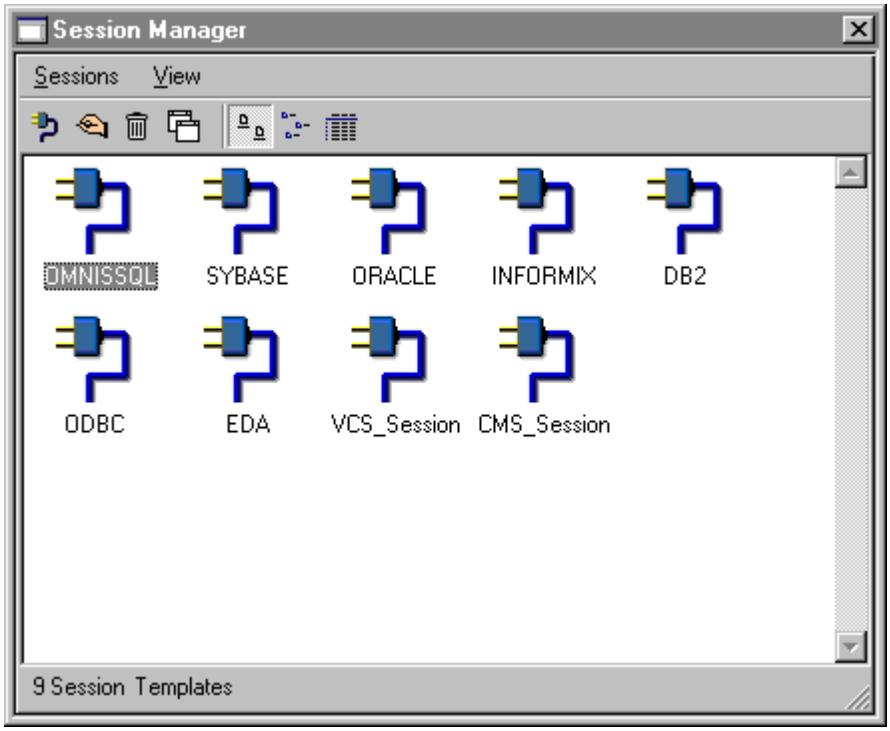
The SQL Browser shows all the databases or sessions you have open, so it is empty when you first open it. Each icon at the top-level of the SQL Browser represents an open database or session. The Tools menu on the SQL Browser menubar contains various tools that let you manage your database. Most of the functionality in the SQL Browser is grayed out until you open a session.

You can either create a new session and enter your details from scratch, or you can modify one of the template sessions in the SQL Browser. A template is provided for each supported server database and middleware connection, along with one for OMNIS SQL. The templates contain some default parameters, but you will need to provide further details, such

as hostname, username, and password, before you can logon to your database. To create or modify a session you need to open the Session Manager.

To open the Session Manager

- Select Session>>Modify Sessions from the SQL Browser menu bar, or click on the Modify Sessions button on the toolbar



Modifying a Session Template

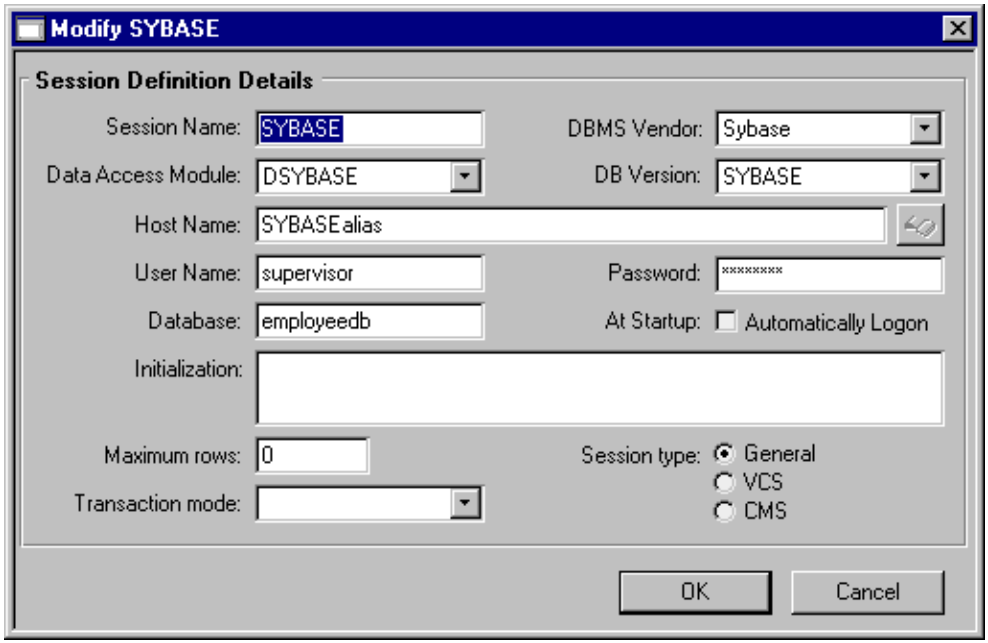
The quickest and simplest way to logon to your database is to modify one of the session templates provided.

To modify a session template

- Double-click on the session in the Session Manager

or

- Click on the session and select Sessions>>Modify



The Modify Session window lets you change the details for a session. Not all the items in the window are required for every DBMS, as described below.

– **Session Name**
the name of the session or logon, limited to a maximum length of 15 characters

– **DBMS Vendor**
the DBMS software

– **Data Access Module**
the name of the DAM; for some databases you can connect using a direct DAM or via ODBC, for example you can connect to Oracle either directly or using ODBC. DAM names all start with the letter D followed by the database or middleware name

– **DB Version**
the database version, as follows

	DB Version
Oracle	ORACLE6, ORACLE7, or ORACLE8
Informix	INFORMIX, INFORMIX-SE or INFORMIX-ONLINE
Sybase	SQLSERVER, or use one of the SYBASE or SQLSERVE synonyms
DB2	not used, leave empty
ODBC	not used, leave empty
EDA	not used, leave empty

– **Host Name**
the server connection information, as follows

	Host Name
Oracle	the Oracle database alias (note this must be prefixed with an @)
Informix	the Informix database name
Sybase	the name of the Sybase alias
DB2	the DB2 database name
ODBC	the Data Source Name defined for the connection via the ODBC Administrator
EDA	the name in the ENTITY field in the EDALINK.CFG file

– **Username and Password**
sets the database username and password strings

	Username	**Password**
Oracle	your username	your password
Informix	not used, leave empty	Optional; use for EXCLUSIVE or SHARED to control access
Sybase	your username	your password
DB2	your username	your password
ODBC	target database username	target database password
EDA	EDA/SQL host username	EDA/SQL host password

– **Database**
sets the SQL Server database for Sybase logons

– **Initialization**
defines an initialization string that is sent to the database when you log on

– **At Startup**
executes this logon automatically when you start up OMNIS

– **Maximum Rows**
sets the maximum number of rows to return for queries in the Interactive SQL window

– **Transaction Mode**
sets the transaction mode for the session: Automatic (the default), Generic, or Server

– **Session Type**
either *General* for a standard database session (the default), *VCS* if the session is to be used with the OMNIS VCS, or *CMS* for the OMNIS CMS

– **DB2 Extenders**
allows system administrators to enable DB2 extender data types

When you have provided all your session information

- Click on OK to save the new or modified session, and close the Modify Session window

Creating a New Session Template

To create a new session template

- In the Session Manager, select Sessions>>New

A new session definition appears in the Session Manager, with the default name New Session. To use this session, you need to rename it, and double-click on it to add your database details, as described in the previous section.

Duplicating a Session Template

To duplicate a session template

- In the Session Manager, click on the session you want to copy and select Sessions>>Duplicate

A copy of the session template appears in the Session Manager. You can select it and modify it as you wish.

Having created or modified your session, you need to open it to establish the connection to your database.

Deleting a Session Template

To delete a session template

- In the Session Manager, click on the template and select Session>>Delete

This operation is irreversible, so only delete a session if absolutely necessary.

Opening a Session

To open a session you need to return to the top level of the SQL Browser.

To open a session

- Make sure you close the Modify Session window and the Session Manager
- In the SQL Browser, select the Session>>Open option from the menubar
- Select the appropriate session from the Open submenu
- or you can
- Click on the Open Session button and select a session from the popup menu

The Session>>Open submenu includes only those sessions that you have modified, that is, ones that have complete logon details.

Once you open a session, most of the SQL Browser functionality is enabled, and you can browse and manipulate your database in the current session. Some features may remain disabled, depending on the database you are connected to.

Closing a Session

To close the current session

- Select the session and click on the Close Session button on the SQL Browser toolbar

or

- Select the session and select Session>>Close from the SQL Browser menubar

or

- Right-click on the session and select Close from its context menu

Session Information

To display information about the current session

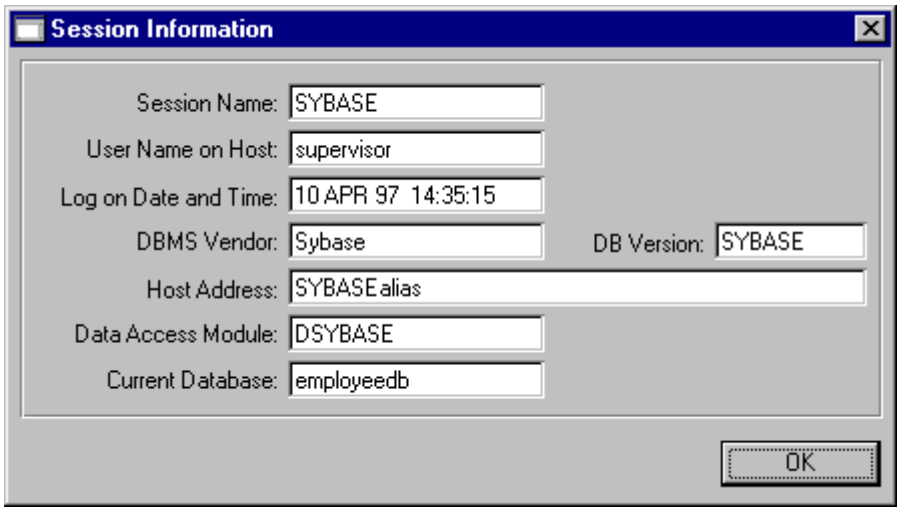
- Select the session and click on the Session Info button in the SQL Browser toolbar

or

- Click on the session and select Session>>Session Info in the SQL Browser menu bar

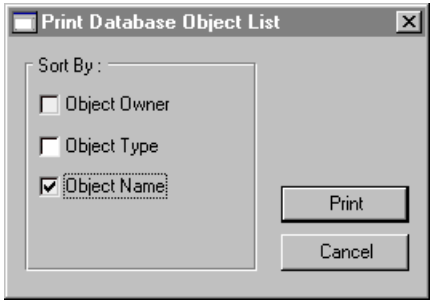
or

- Right-click on the session and select Session Info from its context menu



Database Information

You can print a list of objects in your database including the object name, type, owner, and schema (not applicable for some DBs). The report is sent to the current print destination.



To print an object list for your database

- Click on the database and select Session>>Print Database in the SQL Browser menu bar

or

- Right-click on the database and select Print Database from its context menu

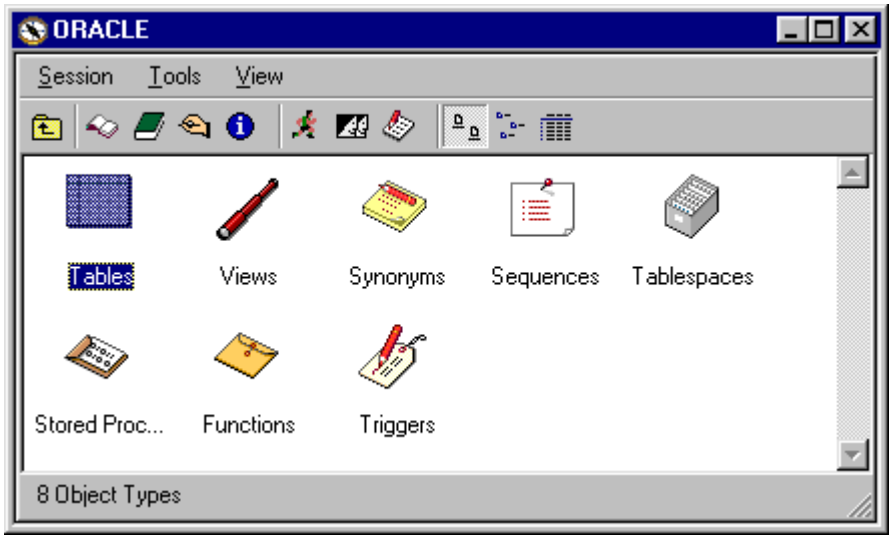
The criteria for the sort are in Owner, Type, Name, Schema order (the order in the dialog), for example, if you click on Type and Name the object list is sorted on Type then by Name.

DB2 Extenders

The final option on the Session menu lets you enable and disable DB2 extenders for a DB2 database. These are described in the *Server-Specific Programming* chapter.

Managing SQL Objects

Once you open a database or session in the SQL Browser, you can view and modify the objects in your database. All databases contain Tables, as well as other objects such as Views, Synonyms, Functions, Stored Procedures, and so on, depending on your database. For example, if you access an Oracle database you will see something like the following.



You can print a list of the objects in a group by right-clicking on the group and selecting Print Objects from the context menu. You can view the contents of a group, such as the Tables group, by double-clicking on the group in the SQL Browser. Once you can see the objects in your database, the SQL Browser provides a number of ways to manipulate them, assuming you have the correct privileges.

Copying Tables between Sessions

You can copy a table from one database or session to another using drag and drop in the SQL Browser.

To copy a table from one database to another

- Open both databases in the SQL Browser using a session for each
- View the tables in each session by double-clicking on the Tables group
- Drag the required table from one database to the other

A series of dialogs will appear to enable the table to be created in the target database. In particular, note that if a table of the same name already exists, you are prompted for an alternative; you cannot overwrite an existing table.

Object Menu

The Object menu lets you create, modify, rename, and delete SQL objects, in addition to providing access to the User Privileges dialog. This menu also allows you to view and insert data for the current table using the Interactive SQL tool. Furthermore for DB2 only, this menu lets you enable and disable the DB2 extenders for the current table. The menu is also available as a context menu by right-clicking on an object.

Creating a new Object

As well as viewing existing objects in your database, the SQL Browser lets you create new objects, such as tables and views and other objects for other types of database. To do this you must have the correct access privileges.

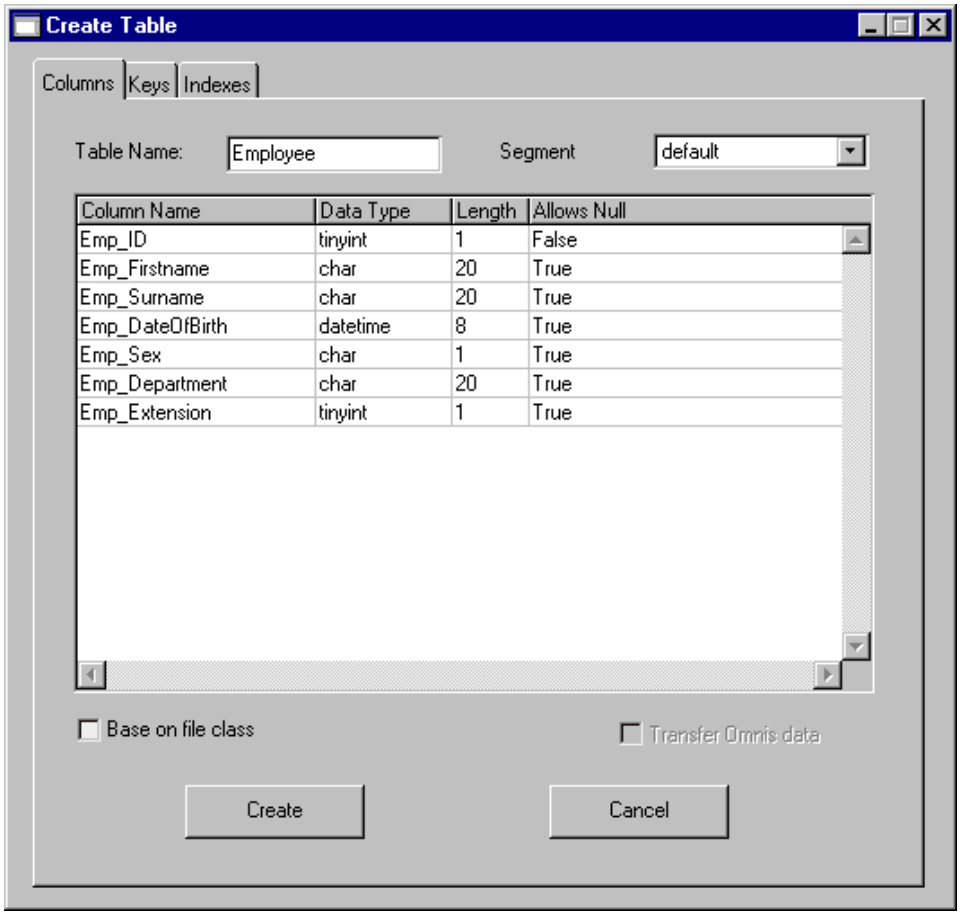
To create a new database object

- In the SQL Browser, display the group of objects in which you want to create a new one, for example, open the Tables group to create a new table
- Select New from the Object menu

or you can

- Right-click on any object and select New from the context menu

The contents of the dialog that appears depends on the object you are creating. For example, the Create Table dialog contains tabs for Columns, Keys, and Indexes.



Modifying an existing object

To modify an object

- Double-click on the object

or

- Click on the object and select Modify from the Object menu

or

- Right-click on the object and select Modify from the context menu

A dialog appears as for creating an object, with the current object's data displayed. As with creating an object, the dialog that appears depends on the type of object.

Renaming an object

The Rename option (not available for ODBC or EDA sessions) renames the current object, subject to any limitations imposed by the specific database. For example, Informix table and column names are limited to a maximum length of 15 characters.

Deleting an object

The Delete option deletes the selected object or objects; since this operation is irreversible, you are asked to confirm the deletion before it occurs.

User Privileges

The Privileges option is only available for direct DAMs and lets you set the user privileges for the current object. For example, you can allow select, insert, and update for the various categories of users and groups, assuming you have the right to do this.

Object Information

You can print a report showing the columns in the current table or view.

To print object information

- Click on the table or view and select Object>>Print in the SQL Browser menu bar

or

- Right-click on the table or view and select Print from its context menu

Viewing and Inserting Data for a Table

The Object menu in the SQL Browser lets you view and insert data into the current table. The Show and Insert options do not require any SQL data classes in your library, in this case, the SQL Browser interacts directly with your database.

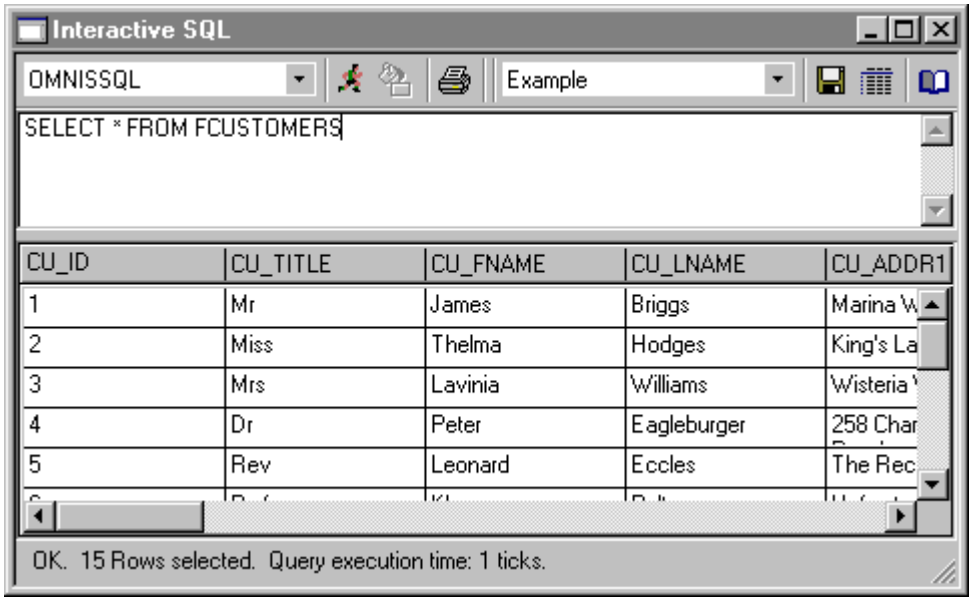
To view the data for a table

- Click on the table and select Show Data from the Object menu

or

- Right-click on the table and select Show Data from the context menu

The Show Data option creates a Select statement for the current table and sends it to your server automatically via the Interactive SQL tool. The results of the Select are displayed in the lower pane of the ISQL tool.

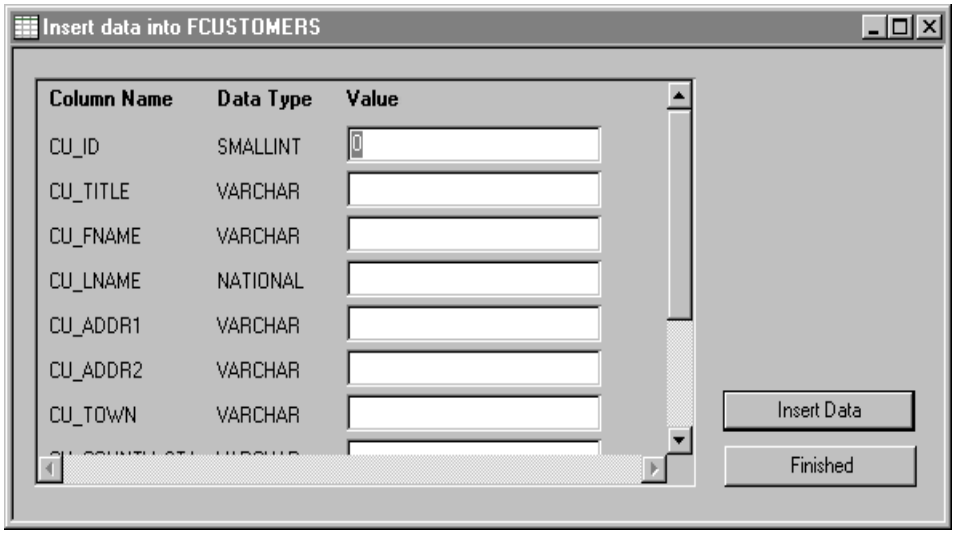


To insert data for a table

- Click on the table and select Insert Data from the Object menu

or

- Right-click on the table and select Insert Data from the context menu



The SQL Browser creates a window containing an entry field for each column in your table. The window lets you insert a single row of data into your database.

DB2 Extenders

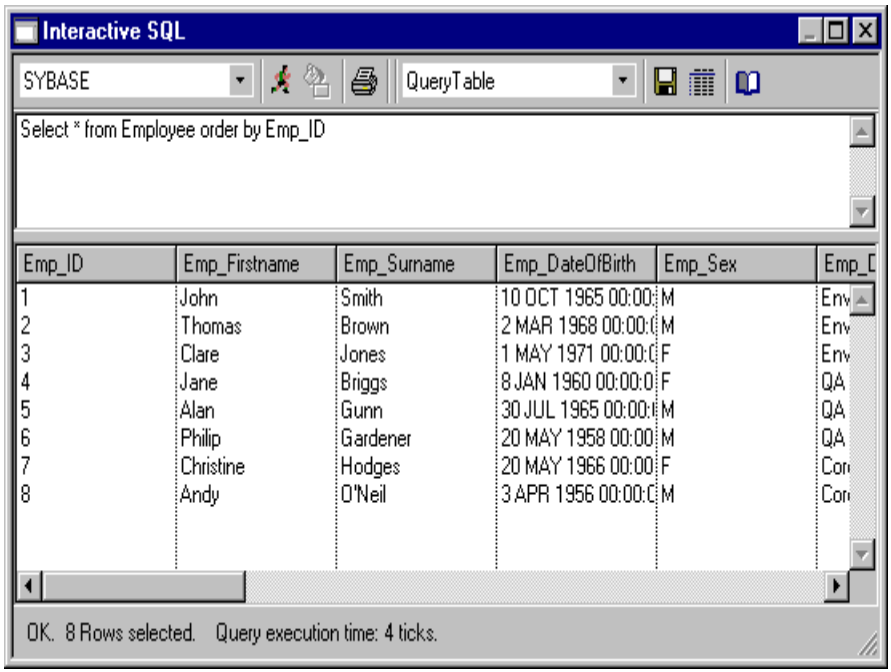
The final option on the Object menu lets you enable and disable DB2 extenders for a table in a DB2 database. This is described in the *Server-Specific Programming* chapter.

Interactive SQL

The *Interactive SQL* tool lets you execute SQL statements in any active session, so you must first logon to your server by opening a session in the SQL Browser. Using Interactive SQL, you can maintain your database interactively or you can test SQL statements before using them in a method. See the *SQL History* window below for a way of copying previously executed SQL statements into a method.

To open the Interactive SQL tool

- In the SQL Browser, select the Tools>>Interactive SQL menu item or click on the Interactive SQL button in the toolbar



The Interactive SQL tool lets you enter any SQL statement valid for your server, including multiple SQL statements if the server supports them. The results pane displays any results for a query with column names taken from the database columns.

You can send any SQL statement to the server, not just a Select query statement, but a Select is the only command that returns results.

The Interactive SQL toolbar has the following buttons:

- **Session Dropdown List**
  drops down a list of the active sessions, letting you switch between sessions
- **Run SQL**
  runs the SQL statement or statements currently in the SQL pane. See below.
- **More Results**
  fills the result table with additional result batches for a SQL Server query
- **Print Results**
  prints a report using the data in the results pane
- **Stored SQL Dropdown List**
  drops down a list of the SQL statements you have saved
- **Save Query**
  stores a SQL statement for later use; a dialog prompts for a name and optional description for the SQL statement.
- **Stored Query Manager**
  lists the SQL statements you have saved and gives you some additional options for each of them. See below for more details.
- **View SQL History**
  lets you view the most recent queries you have run. See below for more details.

To run a SQL statement

- Type your SQL statement into the SQL pane and press Return, or click on the **Run SQL** button

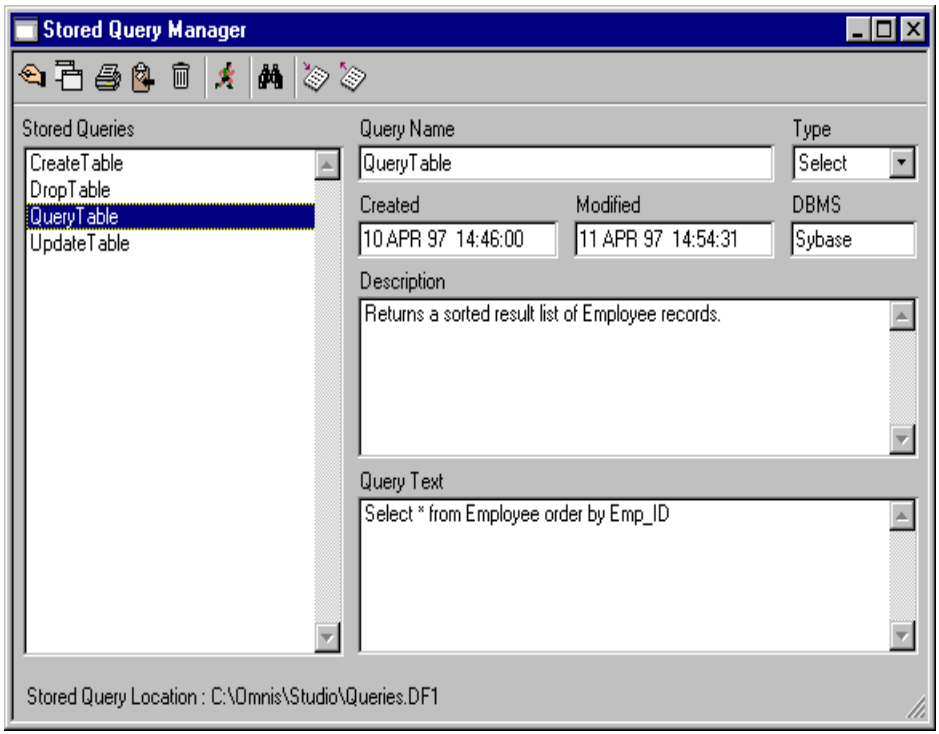
If the SQL statement has a syntax error or some other problem, you will see an error message in the status bar at the bottom of the window. Do not use a semicolon or other delimiter at the end unless you are typing in multiple SQL statements in a format the server accepts.

If the SQL statement is acceptable, the status bar tells you so, the SQL statement is executed and the results are displayed in the results pane. The status bar displays how many rows were returned in the results and how long the statement took in ticks, which is a measure of CPU time relative to your type of computer; see your microprocessor hardware manual or other documentation for details.

You can change the size of the columns in the report pane by dragging the line between the columns to the left or right in the header area, and you can sort rows by clicking on the column headings.

Stored Query Manager

The Stored Query Manager lets you manage the SQL statements you have saved using Interactive SQL.



It has a toolbar, a scrolling list of named SQL statements, and a pane with fields for each statement, including the text of the statement. The list of named SQL statements contains the names of all the SQL statements you have saved. When you select a line in the list, the statement pane displays the details for that statement.

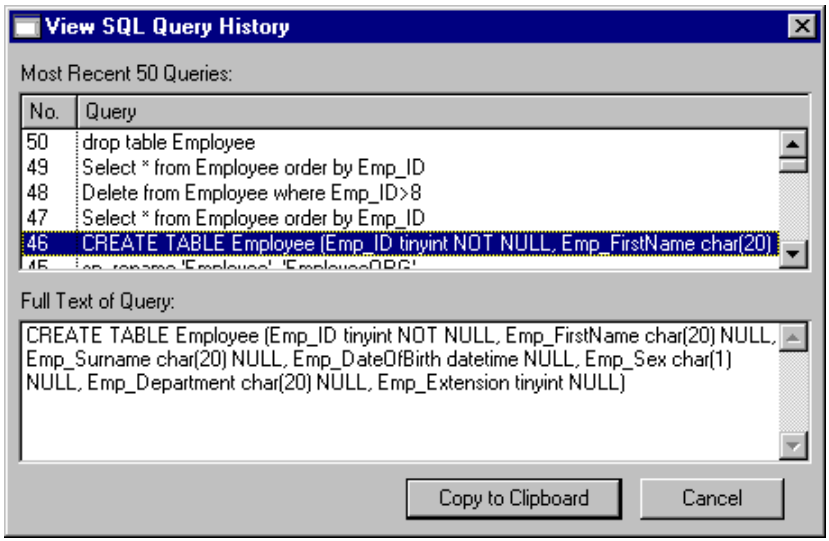
The statement pane gives you the name and type of statement such as Select, the date and time it was created and last modified, the database for which the statement was constructed, a description, and the SQL text of the statement. You can change the name, description, and query text by clicking on the field and editing the text in it.

The toolbar has several buttons for managing the statements:

- **Modify Query**
  saves any changes to disk
- **Duplicate Query**
  alter the query name in the right hand pane before clicking on this button. A copy of the query with the new name is made, which you can edit as you wish
- **Print Query**
  prints the query as a report to a report destination
- **Copy Query to Clipboard**
  copies the query text to the clipboard
- **Delete Query**
  removes the query from the list. Since this is irreversible, you are asked to confirm the deletion.
- **Run Query**
  displays the Interactive SQL window with the selected statement in the SQL pane and runs the query.
- **Find**
  attempts to locates a query; you specify the text you are looking for and which field in the stored query is to be searched
- **Export SQL**
  exports the query text of your currently selected query to a text file.
- **Import SQL**
  imports a query from a text file so you can add it to your stored queries.

SQL History

The View SQL History button lets you inspect the most recent queries you have submitted in your SQL sessions. The number of queries stored can be changed from Tools>>Options in the SQL Browser.



The View SQL Query History window has two panes, the query list and the query text.

You select a query from the query list; the full text of the query is displayed in the query text pane. Clicking on the Copy button copies the query to the clipboard, closes the History window and displays the Interactive SQL window, where you can paste the query. Alternatively, you can paste the query into a method by clicking on the class in the Browser that is to contain the SQL and opening the method editor.

The Cancel button closes the window without copying the SQL text.

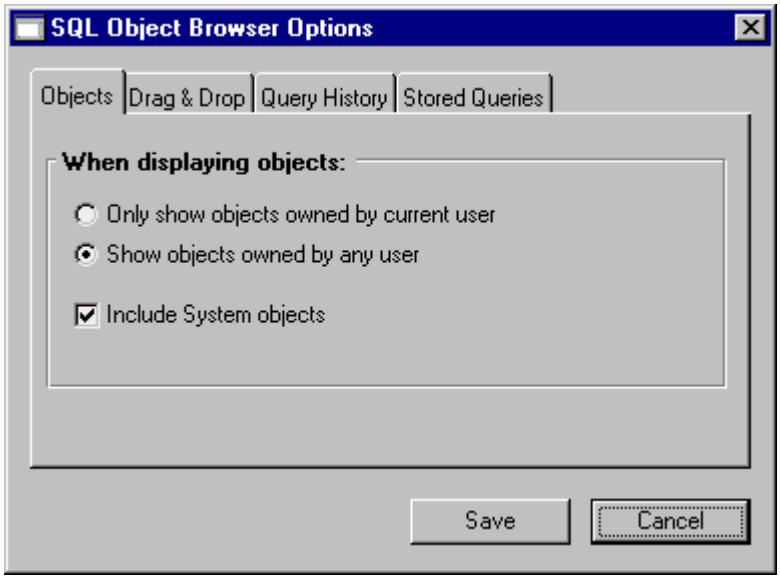
User Administration

The User Administration option lets you alter user information; it is not available in an ODBC or EDA session. If your session provides a direct connection to Informix, Oracle, or Sybase, a dialog specific to the user and group structure of that database appears that lets you alter the user information.

Options

To open the SQL Browser options

- Select Tools>>Options in the SQL Browser toolbar



The SQL Browser options lets you change:

– **Objects**
the range of objects displayed

– **Drag & Drop**
the types of class created when you drag a server table from the SQL Browser onto your library; the default option creates a schema class for every server table, but you can specify that OMNIS creates a schema and table class for every server table

– **Query History**
the number and type of queries retained in the history list

– **Stored Queries**
the location of the OMNIS data file that holds stored queries

Chapter 13—Client/Server Programming

The SQL Browser lets you connect to your server database quickly and easily, and the SQL Form wizards in the Component Store let you build the interface to your server database. However you may want to modify the SQL forms created automatically or build your own from scratch to complete your client/server application. To do this, you use the OMNIS client/server commands.

This chapter covers information and features specific to each of the supported proprietary databases and middleware components. Information about the precise level of client and, to a lesser extent, server software is not presented here since it changes so frequently. Please see the OMNIS Studio web site for the software versions supported.

Connecting to your Database

The session templates in the SQL Browser contain all the necessary code to connect to your server database automatically. However to connect and log on to your database using a method you need to use the following OMNIS commands:

```
Set current session { session name } ;; optional for single session
Start session { DAM name }
Set database version { database version }
Set hostname { host name }
Set username { user name }
Set password { password }
Logon to host
```

Starting a Session

The *Set current session* command creates a session with the specified *session name* which becomes the current session. If you are using a single session, you don't need to use this command. If the specified session already exists it becomes the current session. SQL error messages use the name of the current session. There is no limit to the number of sessions you can have open, except the usual limit imposed by memory.

The *Start session* command loads the specified DAM and initializes communication between the current session and the server database. The following table lists the parameters you can use in the *Start session* command for the different DAMs. Note that some DAM names contain the letter ‘O’, not zero.

Connection	Start Session Keyword
Informix	dINFORMX
Oracle	dORACLE
Sybase	dSYBASE
DB2	dDB2
ODBC	dODBC
EDA	dEDA

Setting the Database Version

The *Set database version* command has a set of options that let you identify the version of the database to use, but if you use the default database version or a DAM with no database version parameters, you don’t need to use this command.

For direct DAMs, the database version can be the version of the software, such as Oracle, but for ODBC it can be the database manager to which you want to connect via the middleware. The following table shows the various database version names and what they mean.

Note that many DAMs have a default database version; the table marks these as (default). If you don't use the *Set database version* command, you get this version by default. Generally you should supply the database version to avoid problems with mismatching server access and server type.

DAM	Database Version	Description
Informix	INFORMIX	INFORMIX
	INFORMIX-ONLINE	Same as INFORMIX (default)
Oracle	ORACLE	ORACLE (default)
Sybase	SQLSERVER	System 10 / 11
	SQLSERVE	Synonym for SQLSERVER
	SYBASE	Synonym for SQLSERVER
	MSSQLSERVER	Microsoft SQL Server
	MSSQLSERVE	Synonym for MSSQLSERVER
	MICROSOFT	Synonym for MSSQLSERVER
	DB2	Gateways to DB2
	MDIDB2	DB2 via MDI Gateway
	GATEWAY	Generic gateways
DB2		Not required; do not use the Set database version command
ODBC		No database version; do not use the Set database version command
EDA		No database version; do not use the Set database version command

Setting the Hostname, Username, and Password

Informix

The *Set hostname* command takes the Informix database name as its argument. *Set username* does nothing. *Set password* is optional, but you can use the term “EXCLUSIVE” if you want exclusive access to the database or “SHARED” (the default) if you want shared access.

You should set the properties **$uniquefieldnames** and **$sensitivefieldnames** to kFalse under the Tools>>Options/Preferences menu option on the main OMNIS menu bar. This combination works best since Informix converts all table and column names to lower case and is case-insensitive.

Oracle

The Oracle DAM does not use the *Set hostname* command. You need to use *Set username* to specify the entire Oracle logon string:

```
Set username { username/password@T:server:SID }
```

with SQL Net1, or specify the user name and password separately in the *Set username* and *Set password* commands. The T refers to the TCP/IP protocol; use AT for AppleTalk. The server is the name of the server. The SID is the name of the database on the server, which Netware servers don't require.

With SQLNet 2.x, the interfaces file TNSNAMES.ORA holds symbolic names for calling a defined connection.

If you have trouble logging on to Oracle through Windows, make sure your WIN.INI file contains an [Oracle] section that has a line like

```
ORA_CONFIG=C:\WINDOWS\ORACLE.INI
```

the ORACLE.INI file must exist in that location

Sybase

The host name is the name of the SQL Server alias. Under MacOS, you will find this name in the **interfaces** file. Look for a line starting with the word 'query'. The name above this line is the SQL Server alias. On Windows, look in the **WIN.INI** file for the [SQLSERVER] section. Each server alias has a line starting with the alias followed by an equal sign and the server information. If you need to set up or change the alias and server information, see the SQL Server documentation or call Sybase technical support. The user name and password are the SQL Server logon user name and password.

The DAM automatically configures itself on encountering a DBMS version of 10.0 or later.

DB2

The host name is the DB2 database name specified during configuration. You do not need to specify the database version using the *Set database version* command. The user name and password are the user name and password for the specified host.

ODBC

The host name is the Data Source Name you defined for the ODBC connection using the ODBC Administrator. The user name and password are the user name and password for the target database system.

If you specify the host name, user name, and password, the ODBC DAM tries the connection using those values. A dialog box prompts you for any missing logon item. You can also supply these values in the **ODBC.INI** file as default values for the fields in the dialog box, which you can then override when the box appears.

Microsoft recommends that you connect to their SQLServer 6.x via ODBC.

EDA

The host name is in the ENTITY field in the **EDALINK.CFG** file. The user name and password correspond to the user name and password of the host, as supplied through the vendor: you should refer to the EDA/SQL documentation for details.

Logging on and off

Each kind of server database has a distinct way of logging on. The following sections describe the specific format of the command that you need to use to logon to your chosen server.

– *Logon to host*
  logs on to the server database specified in the current session

– *Logoff from host*
  logs off the current session from the server but does not free the resources (such as cursors) associated with the session. You can logon to another server with another *Set hostname*, *Set username*, *Set password,* and *Logon to host* sequence

– *Start session*
  logs off then back on if the session had already started

– *Quit cursor(s)*
  logs you off the server and also frees the memory associated with the session, such as cursors and select tables

– *Quit cursor(s) (Current)*
  quits the current cursor

– *Quit cursor(s) (All)*
  quits all open cursors; automatic when OMNIS quits

SQL Separators

The *Set SQL Separators* command lets you define the characters that are used to specify the thousand and decimal separators during a session. Not all DAMs make use of this command, and you need to be sure the separators are compatible with those in OMNIS.

Interacting with your Server

Once a user is logged into a server, they can make use of all the tables and views to which they have been granted access.

Mapping to the Data

Before a client application can get any data from a server, it must set up a corresponding place in OMNIS to hold the data. This involves mapping the structure of the data, including column names and data types. Typically, you do this using OMNIS schema classes. You can define a schema to include all columns of the server table or view, or any subset of the columns. In addition you can create query classes that use columns from one or more schema classes.

You can use schema and query classes to define list and row variables to handle your server data. Information on creating schema, query, and table classes will be found earlier in this manual, as will details on using list and row variables..

Sending SQL to the Server

To send SQL to the server, you can either write your own methods, or use the table instance methods that generate SQL automatically and handle both single row and bulk SQL transactions. OMNIS provides two different ways of sending SQL to the server: the *Perform SQL* command and SQL scripts.

Perform SQL

The *Perform SQL* command sends a single-line SQL statement to the current SQL session. For example

```
Perform SQL { SELECT name FROM Agent }
```

OMNIS sends to the server whatever you pass as a parameter to the command. It can be standard SQL or any other command statement the server can understand.

SQL Scripts and the SQL Buffer

For longer statements that require more than one line, you can use the *SQL:* command. OMNIS has a SQL buffer, an area of memory that contains a single SQL statement that you build up with a series of commands. The command *Begin SQL script* clears this buffer; the command *End SQL script* closes the buffer; the command *Execute SQL script* sends the contents of the buffer to the server. The *SQL:* command enters a line of SQL text into the buffer and appends a space to the end of each line. For example

```
Begin SQL script
SQL: INSERT INTO Agent
SQL: (name,number)
SQL: VALUES ('FRED',123)
End SQL script
Execute SQL script
```

is equivalent to the *Perform SQL* command

```
Perform SQL {INSERT INTO Agent (name, number) VALUES ('FRED', 123)}
```

You cannot split a string literal over more than one line. You can however put more than one statement on a single line, separated by semicolons.

OMNIS limits the size of the SQL buffer to 32K bytes. You should not try to build a SQL statement or series of SQL statements that exceeds this. In particular, do not use the square bracket notation to substitute in very large literal values or BLOBs; you can easily exceed the 32K limit.

The SQL statement buffer holds all SQL statements that have been entered but not yet executed since the last *Begin SQL script* or *Reset cursor* command.

SQL Errors

If *Perform SQL* or *Execute SQL script* result in an error, as well as the flag being cleared, the error code and error text are held by *sys(131)* and *sys(132)* respectively. In addition the $sqlerror() method in the table instance is called. If smart lists are used, the list row properties $error and $errortext contain the same results as sys(131) and sys(132).

Square Bracket Notation

You can substitute text into the SQL statement using square bracket notation. For example

```
Perform SQL { SELECT name from [TABLENAME] }
```

where TABLENAME is an OMNIS variable, substitutes the value of that variable into the SQL statement to use as a table name. When the SQL statement is read, OMNIS evaluates the expression and inserts the result as text into the statement. You must supply quoted literals according to the rules imposed by the version of SQL your server uses. For example, to update an Agent table by setting the name to a string using a variable called FIELD, you must quote the square bracket notation expression:

```
Perform SQL { UPDATE Agent SET name ='[FIELD]' }
```

Bind Variables

A *bind variable* is an OMNIS variable to which you want to refer in a SQL statement. Instead of expanding the expression, OMNIS associates, or binds, the variable value with a SQL variable. To specify a bind variable you place an @ before the opening square bracket. OMNIS evaluates the expression and passes the value to the server directly rather than

substituting it into the SQL statement as text. You can also use this syntax to bind large fields such as pictures into a SQL statement:

```
Perform SQL {INSERT INTO Agent (agentPortrait) values (@[FIELD])}
```

Do not quote bind variables, and use them only to represent complete literals or values, otherwise you will get an error from the server.

Not all database systems allow bind variables; in these cases OMNIS will behave as though they do, but will instead perform literal expansion as though you had entered square bracket notation instead of bind variables

Generally, using bind variables performs better than square bracket notation and is more flexible with respect to data representation. You should use square bracket notation only when the notation expression evaluates to a part of a SQL statement broader than just a value reference such as an entire WHERE clause, or where you know that simple value substitution is all you need.

Bind variables work best for numeric data; strings tend to cause problems because of the issues with quoting. You must include quotes when using square bracket notation, but you don’t need to when using bind variables. If you are inserting NULL data into the database, you should use bind variables, to ensure that SQL nulls, rather than empty strings are inserted.

You can pass an item reference via @[] notation to a DAM if the item reference points to a variable, but this is not recommended. If the item reference points to something else, a class for example, the value is not defined.

Editing the SQL Script

You can use the commands *Get SQL script {field name}* and *Set SQL script {field name}* to copy the contents of the SQL buffer for the current session into a field or variable or to copy the contents of a field or variable into the current SQL buffer, respectively. It can also be useful to view the final SQL script before it is executed.

For example, to manipulate the contents of the SQL buffer directly, the following script displays the buffer and lets you change it.

```
Begin SQL script
SQL: SELECT * from [TableName]
SQL: WHERE [Key] >= [Val2]
End SQL script
Get SQL script { my_string } ;; this puts the SQL into my_string
```

Since this command bypasses all the normal checks carried out on the SQL buffer, you must take care not to introduce errors when using *Set SQL script*.

Preparing the Cursor

When sending a SQL statement using *Perform SQL* and *Execute SQL script* the SQL script is parsed and interpreted by the server. Should you wish to send the same statement again, you can bypass this preparation stage using *Prepare current cursor*. For example:

```
Begin SQL script
SQL: INSERT INTO Sales(col1,col2,..) VALUES (@[col1],@[col2], .. )
End SQL script
Prepare current cursor
Execute SQL script
```

Subsequent use of *Execute SQL script* on the same cursor will execute the same statement without having to set up the SQL buffer each time since the indirection and bind variables are already prepared. This can greatly speed up the process of, say, inserting many rows into a server table within a loop.

```
Set current cursor { Cursor1 }
Repeat
   ; get next row
   Execute SQL script ;; insert the row
Until .. ;; no more rows to insert
```

Data Type Mapping

OMNIS converts the data in an OMNIS field or variable into the corresponding SQL data type. Since each DAM maps to a wide variety of data types on servers, each DAM determines the correct conversion for the current server. See the next chapter for details on how each DAM maps SQL data types to OMNIS datatypes and vice versa.

Select Tables and Cursors

A *select table* is a table of results that belongs to an OMNIS cursor after a select statement has been issued. When you send a SQL SELECT statement to the server and there is no error, the results of the SELECT become available to OMNIS as the select table for the current cursor. A valid SELECT can result in the select table being empty. The flag is true after the execution of a valid select and is only set to false when you attempt to fetch beyond the end of the select table.

An OMNIS cursor should not be confused with a SQL cursor. The ANSI standard select cursor on a remote database is the name of a pointer into a result set on a DBMS. An OMNIS cursor on the other hand contains the context of the connection to a remote database, the associated OMNIS context and possibly a reference to a SQL cursor..

Unless you are using multiple cursors, you don't need to explicitly open a cursor; OMNIS automatically opens a cursor for you.

The *Declare cursor* command defines an OMNIS cursor, and associates a SQL statement with the cursor. The *Open cursor* command opens the cursor, parses the SQL statement,

binds input data, and executes the SQL statement. The *Set current cursor* command switches OMNIS to use the named cursor. The *Prepare current cursor* command opens the current cursor and parses the SQL statement but does not bind data nor execute the statement. You can later execute the statement with *Open cursor*, which you can call multiple times with some time savings over not having prepared the cursor with *Prepare current cursor*.

Not all servers or DAMs use SQL cursors. This means you can have only one SQL statement active at a time in a session. The following cursor command discussion applies only to those DAMs that have cursor operations.

When you execute a SQL SELECT statement through an OMNIS cursor, the current cursor acquires a reference to the first row in the resulting select table. When you execute $fetch() you fetch the row pointed to by the current cursor and move the reference to the next row. You can have more than one cursor active at a time, letting you select rows based on values retrieved from a completely separate select table. You use the *Set current cursor* command to use a particular cursor as the current cursor with the $fetch().

Fetching rows is described in detail in the *List Programming* chapter.

Building a List from a Select Table

The *Build list from select table* command fetches all the rows of the select table into a list. You first need to define the list with the appropriate field-to-column mappings; in a table instance these will automatically be correct. Normally, the command appends the values rather than overwriting any values in the list so you can use this feature to put multiple select tables into a list, but there is a *Clear list* option to clear the list first. If you have defined the list with other fields or variables, the *Add CRB fields* option inserts these values to the list as well.

You can use #LM or $linemax to limit the size of the list regardless of the number of rows in the select table. There are also commands provided by most servers to limit number of rows returned; do not confuse these with the #LM value, which just affects the list.

The following method selects a table called Contacts with columns name and number directly into a list.

```
; Declare local variable lvContacts of type List
Set current list lvContacts
Define list from table { fContacts }
Perform SQL { SELECT name, number FROM fContacts }
; Creates the select table of all rows and columns
If flag true
   Build list from select table
End If
```

You can generally improve performance in the *Build list from select table* command by setting the batch size with *Set batch size*. you will need to experiment to determine the best setting for your environment.

Retrieving Rows to File

Retrieve rows to file lets you retrieve a select table directly into a file on disk. You should make sure that there are no intervening commands that might move the current cursor if you want to retrieve all of the select table rows into the file.

To retrieve a select table into a file, you first set the client import file, delete any existing file if you don't want to append records, open the file, retrieve the records, then close the file. You must close the file in order to import it into another tool.

When you read a select table into an import file with *Retrieve rows to file*, you must have the import file open and ready to receive it. If the file already exists, OMNIS opens it and moves to the end of the file, where it will append the incoming data. However, if the file does not already exist, OMNIS creates and opens it. By preceding the *Open client import file* with a *Delete client import file*, you are guaranteed an empty file for the next SQL transaction. For example

```
Set client import file name { XprImportFile }
Open client import file
Begin SQL script
SQL: SELECT cust_name, cust_city, credit_line FROM customer;
End SQL script
Execute SQL script
Retrieve rows to file
Close client import file
```

Describing Your Database

OMNIS provides a set of commands that give you access to data dictionary information about any database to which you can connect using a DAM. Using these commands, you can code database-independent methods to describe your server database, no matter what its type. These commands work by creating select tables as though you had queried the information from the database. You use standard commands to fetch the data into OMNIS.

Describe database (Tables)

The *Describe database (Tables)* command creates a select table with one row for each server table accessible to the current session. The single column contains the table name.

The following method builds a list of tables available for the specified session.

```
Set current list [MyList]
Define list {Table_Name}
Describe database (Tables)
Build list from select table
```

Describe server table (Columns)

The *Describe server table (Columns) { tablename }* command creates a select table with one row for each column of the specified server table. Each row contains the following columns:

Col	Meaning
1	the column name
2	the equivalent standard SQL data type (CHARACTER, NUMBER, DATETIME, ...)
3	the column width (for character columns)
4	the number of decimal places (for numeric cols), empty for floating numbers
5	NULL or NOT NULL
6	indicates whether the column is an index or a primary key (see below)
7	the remarks or description for the column where available.

Column 6 indicates whether the column is an index or a primary key using a 2-character string.

1st char	2nd char	Either char	Denotes
‘Y’			index
	’Y’		primary key
		‘N’	neither
		‘U’	indeterminate

Composite Primary keys or indexes cannot be detected and will show as “N”s.

In addition, the table instance property $colsinset returns the number of columns in the current result set for the session used by the table.

To build a schema class from a table called Agent, you would execute the commands

```
Describe server table (Columns) {Agent}
Make schema from server table {AgentSchema}
```

This command uses the select table of columns names created by the *Describe server table (Columns)* command and builds a schema class called AgentSchema. The *Make schema from server table* command sets the primary key property of a schema column based on the information returned in column 6 of the *Describe server table (columns)* result set.

Describe database (Views)

The *Describe database (Views)* command creates a select table with one row for each view available to the current session. The single column contains the view name.

Describe results

The *Describe results* command builds an OMNIS list of information similar to *Describe server table (Columns)* for the current select table. It builds the list directly rather than requiring you to fetch the rows, since this is describing an internal data structure, not a part of the server data dictionary.

Describe server table (Indexes)

The *Describe server table (Indexes) { tablename }* command creates a select table that lists the unique indexes for the specified server table. Each row has the following columns:

Col	Meaning
1	the name of the indexed column
2	the name of the index used for the column (in column 1)
3	the numeric position of the column within a composite index (defaults to 1 for non-composite indexes)

You can obtain the list of non-unique indexes by adding the /N switch to the command:

```
Describe server table (Indexes) {MyTable /N}
```

The /A switch gives a list of all indexes. /U is the default; unique indexes only. This command lets you write general purpose data handling methods, such as:

```
Set current list MY_LIST
Define list {KEY_NAME}
Describe server table (Indexes) {[TABLE]}
Build list from select table
```

Building User Views

Using the *Describe server table (Columns)* command, you can get column information about a table in a server database. OMNIS can use this information to build classes that correspond to the server tables with the *Make schema from server table* command. This gives you a basis for creating user views; it can also let you develop dynamic user views based on the structure of a server database.

Using the data dictionary commands, you can write methods that attach to the server, get a list of tables, get a list of columns for each table, then build a schema class for each server table.

Transactions

There are several commands in OMINS to manage transactions in addition to the native SQL facilities for transaction management.

Transaction Modes

Transaction processes vary from one database to another. OMNIS provides some options for transactions that let you choose how to control transactions for your particular database.

The *Set transaction mode* command lets you set transaction processing in the current session to one of these values:

- **Automatic**
  commit or roll back each SQL statement automatically as soon as the next SQL command starts (*Begin SQL script*, *Perform SQL*, *Reset cursor*) or if the statement did not generate a result set immediately after it was executed, or the statement fails
- **Generic**
  implement basic transaction processing using the *Commit current session* and *Rollback current session* commands
- **Server**
  rely on server transaction processing commands only; see the specific DBMS documentation for the default transaction mode

With INFORMIX, you should not use Automatic transaction mode since INFORMIX closes all cursors at the end of each transaction. If you automatically end the transaction after each SQL statement and you are using multiple cursors, any SQL statement with one cursor will close all the other cursors. The Server transaction mode under INFORMIX turns statement level checking on. All specified constraints are checked at the end of each INSERT, UPDATE or DELETE statement. If a constraint violation occurs, the statement is not executed.

The Sybase DAM defines automatic transaction mode as committing after the execution of each SQL statement: they use Sybase transaction management so the commit happens as part of the Sybase processing rather than on the return to the OMNIS client.

With generic transaction mode, you can commit the current session and transaction with the command *Commit current session*. You can roll back the session and transaction with the command *Rollback current session*. With server transaction mode, you must use the server-specific SQL statements or functions to implement transactions. You can also commit and roll back with the standard SQL commands COMMIT WORK and ROLLBACK WORK, if your database manager provides that command, in server mode.

The server may commit or roll back the transaction under certain circumstances such as errors of a certain type, server shutdown, methods with embedded commits and rollbacks,

or data definition (DDL) statements such as CREATE TABLE. See your DBMS documentation for more information about transaction management in your type of server.

Multiple Sessions

OMNIS makes it possible to connect not only to multiple servers but to multiple servers running different database managers, using the OMNIS session. Setting a session using the *Set session { session_name }* command makes that session the *current session*. If you use a session name already set, the named session becomes the current session.

INFORMIX does not support multiple sessions on different servers but does support multiple sessions on the same server. To start a session on a different server you must logoff and logon to the different server.

All the session commands (*Open cursor*, *Prepare current cursor*, *Commit session*, and so on) apply to the current session. Each session has a set of objects (select tables, import file, error status, and so on) that let it manage the individual server connection. You can have any number of sessions going on at once, subject to memory limits, so you can connect to many different types of server at the same time.

If you issue a SQL command but you haven't set up a session, OMNIS creates a default session called CHANNEL_*n* as the current session, where *n* is a sequential number.

It is important to understand how your servers process transactions and that you must coordinate all transactions among multiple servers, since the individual database managers on the servers don't communicate with one another.

Unlike a single database session, a multiple-session client application has multiple server transactions progressing in parallel. At any given moment in time, each session has a single transaction going and transactions can interact: you can select data from one server and update another while a third waits for an insert done after successful completion of the update. In this case, you can consider all three sessions part of a single, large transaction. All the transactions start at the same point and commit or roll back at the same point. Remember that in OMNIS is that you must commit or roll back the individual sessions with three separate *Commit session* commands.

For example, here is a method that updates a row in a lookup table in a database based on a value retrieved from another database:

```
; Declare class variable CHAR255 of Character 255 type

; The following methods create a session and logon to the database
; servers, setting the individual session names.

Do method M_DB/d1 ( ‘Session1’ ) ;; Logon to database 1
Do method M_DB/d2 ( ‘Session2’ ) ;; Logon to database 2

; Read the rows from the server
Perform SQL { set connect default }
Perform SQL { select char255 from testtypes }
.. error check
;  Fetch the row.
Fetch next row
.. error check

; Change to the other session before executing the SET CONNECT
; statement. This preserves the cursor and select table
; in the first session.
Set current session { Session2 }
;  Update the ORACLE row.
Perform SQL
   {update testtypes set char255 = "X" where char255 = @@[CHAR255]}
.. error check
;  Commit both sessions.
Commit current session
.. error check
Set current session { Session1 }
Commit current session
.. error check
```

In practice you would improve this code by encapsulating many of the common elements into separate methods, then calling the methods to execute the separate, individual transactions, testing for results in the main transaction method. You can also use error handlers to simplify error checking..

Server Status and Error Handling

This section shows how you can obtain status and error information for your database. It also contains a section for each DAM giving hints on solving specific problems you may encounter.

Status and Error Functions

The *sys()* function returns error and status information from the current session. The most important functions are *sys(131)* and *sys(132)*, which return server errors.

Function	Returns
sys(130)	the name and version of the DAM; empty if you have not issued the *Start session* command
sys(131)	the error code of the last command sent to the server, zero for no error; codes are specific to the server DBMS, see the DBMS documentation
sys(132)	the error text that corresponds to the error code
sys(133)	the number of columns for the current select table
sys(134)	the number of rows processed by the previous SQL statement; set after INSERT, UPDATE, and DELETE statements, zero after most other statements
sys(135)	the number of rows you have fetched from the select table; after selecting all rows with *Retrieve rows to file*, *Build list from select table*, or the final *Fetch next row*, corresponds to the number of rows in the select table
sys(136)	name of the current cursor
sys(137)	name of the current session
sys(138)	1 if there are more rows to fetch, 0 if there are none; this helps deal with multiple select tables in Sybase

All server errors set the flag to false. If you get errors during an OMNIS command and the error is not a fatal OMNIS error, it does not prevent execution of further commands. You should always test the flag after SQL commands and take the appropriate action.

```
Set session { CHAN_TUT }
Start session { ORACLE }
If flag false
   OK Message { Start failed }
   Reset cursor (Session)
   Quit all methods
End If
Set username { your_name }
Set password { your_password }
Logon to host
If flag false
   OK Message { Logon failed: [sys(131)]--[sys(132)] }
   Reset cursor (Session)
   Quit all methods
End If
```

Reset cursor

The *Reset cursor* command resets communication with the server to a stable state. It clears or empties the SQL buffer, the error status, and the select table for one or more cursors.

– *Reset cursor (Session)*
resets the cursors in the session containing the current cursor

– *Reset cursor (Current)*
resets the current cursor

– *Reset cursor (All)*
resets all the open cursors

Character Mapping

When reading data from a server database, OMNIS expects the character set to be the same as that used in an OMNIS data file. The OMNIS character set is based on the MacOS extended character set, but is standard ASCII up to character code 127. Beyond this value, the data could be in any number of different formats depending on the client software that was used to enter the data.

The *Set character mapping* command lets you translate the character codes for data read into and sent out of OMNIS client/server libraries. You use this command to select a conversion table for the data flowing in and out of OMNIS. For example, suppose you are working with a database that stores EBCDIC characters. In order to accommodate this database, you could create an '.IN map file' which translates EBCDIC characters to ASCII characters when the server is sending OMNIS data. One could also create a '.OUT file'

which reverses the process by converting ASCII to EBCDIC characters when OMNIS is sending data to the server.

Under Windows, OMNIS uses the same character set as under MacOS, and a mixed platform OMNIS installation should have no need of *Set character mapping*. However, if the data in a server table was entered by another software package, running under Windows or DOS for example, the characters past ASCII code 127 would be incorrect when read with either OMNIS platform. In this situation the *Set character mapping* command could be used to map the character set.

When you use the *Set character mapping* command, performance will be adversely affected because each character processed must invoke the OMNIS conversion routine. If possible, data should be converted once and written to a new server table in the OMNIS format. However, when sharing server tables with non-OMNIS clients this is not possible.

There are two kinds of character maps: IN and OUT files. IN files are used to translate characters coming from a server database into OMNIS. OUT files are used to translate characters that travel from OMNIS back to a server database. The pair of files must have the same name but with the extensions .IN and .OUT, and the *Set character mapping* command loads .IN and .OUT files at the same time. Charmap files are restricted to eight characters plus an extension of .IN or .OUT as required, and must be placed in a folder called CHARMAPS under the OMNIS\EXTERNAL folder.

The Charmap Utility

The Charmap utility lets you create character mapping files and is located in the OMNIS\STUDIO folder. You can change a given character to another character by entering a numeric code for a new character. The column for the Server Character for both .IN and .OUT files may not actually represent what the character is on the server. This column is only provided as a guide. The Numeric value is the true representation in all cases.

To change a character select a line in the list box, and change the numeric code in the Server Code edit box. Once the change has been recorded, press the Update button to update the character map. You can increase/decrease the value in the Server Code edit box by pressing the button with the left and right arrows. Pressing the left arrow decreases the value, pressing the right arrow increases the value.

The File menu lets you create new character map files, save, save as, and so on. The Make Inverse Map option creates the inverse of the current map, that is, it creates an .in file if the current file is an .out character map, and vice versa.

Using the Map File

Make sure the .IN and .OUT files are located in the OMNIS\EXTERNAL\CHARMAPS folder. Enter the *Set character mapping* command, followed by the CharMap translation file name pair, where the file name pair is both an .IN and .OUT pair of translation tables. For example:

```
Set character mapping { TransTbl }
; for the files TRANSTBL.IN file and TRANSTBL.OUT
```

You can disable translation by specifying the *Set character mapping* command with no parameter.

Chapter 14—SQL Classes and Notation

OMNIS has three *SQL classes* that provide the interface to your server database: they are the *schema*, *query*, and *table* class. Schema and query classes map the structure of your server database. They do not contain methods, and you cannot create instances of schema or query classes. You can however use a schema or query class as the definition for an OMNIS list using the $definefromsqlclass() method, which lets you process your server data using the SQL methods against your list. When you create a list based on a schema or query class a table instance is created which contains the default SQL methods.

Table classes provide the interface to the data modeled by a schema or query class, and exist primarily to allow you to override the default methods in the table instance. Like schema and query classes, you can use a table class as the definition for an OMNIS list and use the same SQL methods against your list.

The SQL list methods and notation are described in this chapter. Creating SQL classes in the Browser or Component Store is described in the *Using OMNIS Studio* manual.

Schema Classes

A *schema class* maps the structure or *data dictionary* of a server table or view within your library. A schema class contains the name of the server table or view, a list of column names and data types, and some additional information about each column. The data types are the equivalent OMNIS data types, and the names must conform to the conventions used by the particular server. Schema classes do not contain methods, and you cannot create instances of a schema class. You can define a list based on a schema class using the *Define list from SQL class* command or the $definefromsqlclass() method. You can create a schema class either from the Browser or Component Store.

Schema Class Notation

Each library has a $schemas group containing all the schema classes in the library. A schema class has the type kSchema.

In addition to the standard class properties, such as $moddate and $createdate, a schema class has the following properties

– **$objs**
the group of columns in the schema class

– **$servertablename**
the name of the server table or view to which the schema corresponds

The $objs group containing the columns in the schema class supports the group methods including $first(), $add(), $addafter(), $addbefore(), and $remove(). The $add... methods require the following parameters

– **Name**
the name of the column

– **Type**
constant representing the OMNIS data type of the column

– **Subtype**
constant representing the data subtype of the column

– **Description** (optional)
a text string describing the column

– **Primary-key** (optional)
a boolean set to kTrue if this column is a primary key. If omitted it defaults to kFalse

– **Maximum-Length** (optional)
for character and national columns, the maximum length; for other types, OMNIS ignores the value of this parameter. If omitted for character and national columns, it defaults to 10000000.

– **No-nulls** (optional)
a boolean set to kTrue if this column cannot have NULL values. If omitted it defaults to kFalse

You can identify a particular column in the $objs group using its column name, order, or ident, a unique number within the scope of the schema class assigned to the column when you add it. A schema column has the following properties (all are assignable except $ident)

– **$name**
the name of the column

– **$coltype**
the OMNIS data type of the column

– **$colsubtype**
the OMNIS subtype for the data type of the column

– **$colsublen**
the maximum length for Character and National columns

– **$desc**
a text string describing the column

– **$primarykey**
if kTrue the column is a primary key

– **$nonull**
if kTrue the column does not allow null values

– **$order**
the position of the column in the list of columns in the schema class

– **$ident**
a unique number within the scope of the schema class, identifying the column

Make Schema From Server Table

You can also create a schema using the *Make schema from server table* command. For example

```
Describe server table (Columns) {ServerTable}
Make schema from server table {MySchema}
```

will create the schema MySchema, referencing server table ServerTable, using the current OMNIS session.

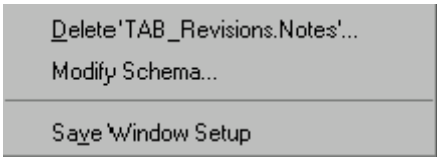
Query Classes

Query classes let you combine one or more schema classes or individual columns from one or more schemas, to give you an application view of your server database. A query class contains references to schema classes or individual schema columns. Like schema classes, query classes do not contain methods, and you cannot create instances of a query class. You can define a list based on a query class using the *Define list from SQL class* command or the $definefromsqlclass() method.

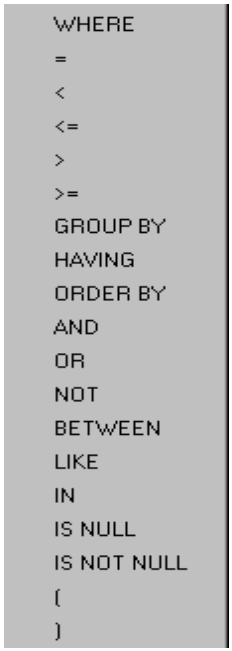
You can create a query class either from the Browser or Component Store. The Catalog pops up when you open the query class editor, which lets you double-click on schema class or column names to enter them into the query editor. Alternatively, you can drag schema class or column names into the query editor. Furthermore, you can reorder columns by dragging and dropping in the fixed left column of the query editor, and you can drag columns from one query class onto another. You can also drag a column from the schema editor to the query editor.

You can drag from the query list, the schema editor, and the Catalog, and drop onto the extra query text field labeled ‘Text appended to queries’. Dragging a query column from the right-hand list of the catalog query tab inserts a bind variable reference in the form @[$cinst.name].

The column entries have a context menu, which allows you to delete a column, and to open the schema editor for the schema containing the column.



The additional query text edit field has a context menu which allows you to insert text commonly used in SQL queries.



The query class editor does not validate schema class or column names, nor does OMNIS automatically update query classes when you edit a schema class. You need to update your SQL classes manually using the Find and Replace tool.

The alias allows you to eliminate duplicate column names when defining a list from the query class. By default, each list column name is the same as the schema column name. You can override this with the alias. If the column name is empty, meaning use all columns in the schema, OMNIS inserts the alias at the start of each column name in the schema, to create the list column name; otherwise, OMNIS uses a non-empty alias as the list column name.

Calculated Columns

Query classes can also contain calculated columns. A calculated column is an entry in a query class which has:

– A schema name, which determines the table to be used in the SQL statement.

– A column name. This is the calculation. OMNIS treats a column name as a calculation if it contains at least one open parenthesis and one close parenthesis. This rule helps to distinguish a calculated column from a badly named schema column. OMNIS performs no validation on the calculation, and it simply inserts it into queries generated by $select or $selectdistinct, and into the result of $selectnames.

– An alias, used as the list column name.

A calculated column is represented, in the list or row variable defined from a SQL class, as a character column of maximum length. Its column properties $excludefrominsert and $excludefromupdate (see the Table Instance section) are automatically set to kTrue. If you include strings in the form “<schema name>.” or “<library>.<schema name>.” in the calculation, then OMNIS replaces them with “<server table name>.” when it adds the calculation to a query. The “<server table name>” comes from the schema class.

Query Class Notation

Each library has the group $queries containing all the query classes in the library. A query class has the type kQuery.

A query class has the standard properties of a class together with $extraquerytext, a text string which in some cases OMNIS appends to automatically generated SQL, and for example may contain a where clause. The extra query text string can be empty. Before OMNIS adds $extraquerytext to a SQL query, it replaces strings in the form “<schema name>. “and “<library>.<schema name>. “ with “<server table name>.”. The “<server table name>” comes from the schema class. This allows you to design query classes which are independent of the table names actually used on the server, since the only place storing the table name is the schema.

A query class has a $objs group containing a list of references to schema columns, or schema classes. $objs supports the same group methods as $objs for the schema class, with the exception of $findname. The $add… methods require the following parameters:

– **Schema name**
  the name of the schema, which can be qualified by a library name

– **Column name** (optional)
  the name of the column in the schema

– **Alias** (optional)
  the alias used to eliminate duplicate list column names

Each query class object has the following properties.

– **$schema**
  the name of the schema, which can be qualified by a library name

– **$colname**
  the name of the column in the schema; if empty, all columns from the schema class specified in the $schema property are included

– **$alias**
  lets you eliminate duplicate column names from a list defined from a query or a table class referencing the query; if $colname is empty, this is a prefix which OMNIS inserts

at the start of each column name in the schema named in $schema; otherwise, OMNIS uses a non-empty alias in the place of the column name

– **$order**
the position of the object in the class

– **$ident**
a unique numeric identifier for the object

A list defined from a query class using the $definefromsqlclass() method has columns which correspond to the objects in the query class. The order of the columns in the list corresponds to the order of the columns in the query class. When an object includes a complete schema, the columns have the order of the columns in the schema class. You can eliminate duplicate list column names using the $alias property.

Queries Tab in the Catalog

The Catalog has a queries tab which lists the query classes in the current library. For each query class, the right hand list shows the list column names which would result from defining a list from the query class.

Creating Server Tables from Schema or Query Classes

You can create a table or view in your server database by dragging a schema or query class from the IDE Browser and dropping it onto an open session in the SQL Browser.

To create a server table or view from a schema or query class

- Create the schema or query class in the IDE Browser
- Define the columns in the schema or query class
- Use the SQL Browser to open the SQL session for your database
- Drag the schema or query class from the IDE Browser on to your Session

If you drag a schema class onto an open session, OMNIS creates a SQL table with the table name defined in your schema class. If you drag a query class, OMNIS creates a SQL view with the name of the query class.

Table Classes

An instance of a table class provides the interface to the data modeled by a schema or query class. You only need to create a table class if you wish to override some of the default processing provided by the built-in table instance methods.

You can create a table class from the Component Store or from the Browser. You can edit the methods for a table class or add your own custom methods in the method editor.

Table Class Notation

Each library has a $tables group containing all the table classes in your library. A table class has all the basic properties of a class plus $sqlclassname, which holds the name of the schema or query class associated with the table class. To create a table class using a method, you can use the $add() method.

```
Do $clib.$tables.$add('MyTable') Returns TabRef
; returns a reference to the new table
Do TabRef.$sqlclassname.$assign('AgentSchema') Returns MyFlag
```

Table Instances

You create a table instance in OMNIS when you define a list or row variable from a schema, query, or table class, using the *Define list from SQL class* command, or the $definefromsqlclass() method. Table instances created from schema or query classes have all the default methods of a table instance. Table instances created from a table class have all the default methods of the table class in addition to any custom methods you have added, perhaps to override the default methods.

When you use *Define list from SQL class* or $definefromsqlclass(), OMNIS defines your list to have either one column for each column in the schema class, or one column for each column referenced by the query class. In the case where you use a table class, OMNIS uses the $sqlclassname property of the table class to determine the schema or query from which to define the list.

A list variable defined in this way has all of the methods and properties of a normal list variable, together with all of the methods and properties of the table instance. You never access the table instance directly; you can think of it as being contained in the list variable.

For example, if you want to display a grid containing your data in a SQL form you can use the following code in the $construct() method of the form to create a list based on a schema class

```
; Declare instance variable iv_SQLData of type List
Set current session {session-name}
Do iv_SQLData.$definefromsqlclass('SCHEMACLASSNAME')
Do iv_SQLData.$select()
Do iv_SQLData.$fetch(1000) ;; Fetch up to 1000 rows
```

Once you have defined and built your list you can use the table instance methods to manipulate the data. Equally you could declare a row variable, define it from a table, schema or query class, and manipulate your data on a row-by-row basis using many of the same methods.

You can add columns to a list which has a table instance using the $add() method. For example, the following method defines a list from a query class and adds a column with the specified definition to the right of the list.

```
Do LIST.$definefromsqlclass($clib.$queries.My_Query)
Do LIST.$cols.$add('MyCol',kCharacter,kSimplechar,1000)
```

Columns added in this way are excluded from the SQL queries generated by the SQL methods described in this section. You can only add columns to the right of the schema or query related columns in the list.

The *Define list from SQL class* command and $definefromsqlclass() method both reset the $linemax property of the list to its largest possible value.

Table Instance Notation

Table instances have methods and properties which allow you to invoke SQL queries and related functionality via the list containing the table instance. Some methods apply to list variables only and some to row variables only. Some of these methods execute SQL, which by default executes in the context of the current OMNIS session. The methods do not manage transactions; that is your responsibility.

The table instance methods are summarized in this section, with a more detailed description of each method in the next section.

– **$select()**
  generates a Select statement and issues it to the server

– **$selectdistinct()**
  generates a Select DISTINCT statement and issues it to the server

– **$fetch()**
  for a list, fetches the next group of rows from the server, for a row, fetches the next row

The following methods apply to row variables only.

- **$insert()**
  inserts a row into the server database
- **$update()**
  updates a row (or rows if the where clause applies to several rows) in the server database
- **$delete()**
  deletes a row (or rows if the where clause applies to several rows) from the server database

The following methods apply to smart lists only, updating the server database from the list.

- **$doinserts()**
  inserts all rows in the list with the row status kRowInserted
- **$dodeletes()**
  deletes all rows in the list with the row status kRowDeleted
- **$doupdates()**
  updates all rows in the list with the row status kRowUpdated
- **$dowork()**
  executes the three $do... methods above, in the order delete, update, insert

When you call $doinserts(), $dodeletes(), $doupdates() or $dowork(), the table instance calls the appropriate method(s) from the following list, to invoke each individual insert, delete or update. This allows you to use table class methods to override the default processing. As a consequence these methods only apply to smart lists.

- **$doinsert()**
  inserts a single row with row status kRowInserted
- **$dodelete()**
  deletes a single row with row status kRowDeleted
- **$doupdate()**
  updates a single row with row status kRowUpdated

The following methods apply to smart lists only, reverting the state of the list, that is, they do not affect the server database.

- **$undoinserts()**
  removes any inserted rows from the list
- **$undodeletes()**
  restores any deleted rows to the list, and resets their status to kRowUnchanged

– **$undoupdates()**
  restores any updated rows to their original value, and resets their status to kRowUnchanged

– **$undowork()**
  executes the three $undo... methods above, one after the other, in the order insert, update, delete

You can use the following methods to create text strings suitable for using in SQL statements. You are most likely to use these if you override default table instance methods using a table class.

– **$selectnames()**
  returns a comma-separated list of column names in the list or row variable, suitable for inclusion in a SELECT statement

– **$createname()**
  returns a comma-separated list of column names, data types, and the NULL or NOT NULL status, for each column in the list or row variable, suitable for inclusion in a CREATE TABLE statement

– **$updatenames()**
  returns a text string containing a SET clause, suitable for inclusion in an UPDATE statement

– **$insertnames()**
  returns a text string containing a list of columns and values for a row variable, suitable for inclusion in an INSERT statement

– **$wherenames()**
  returns a text string containing a Where clause, suitable for inclusion in a SQL statement that requires a constraining clause

You can use the following method in a table class.

– **$sqlerror()**
  a means of reporting errors. The table instance default methods call this method when a problem occurs while executing SQL

Table instances have the properties of list or row variables as well as the following.

– **$sqlclassname**
  the name of the associated schema or query class used to define the columns of the list; this property is NOT assignable

– **$useprimarykeys**
  if true, only those schema columns that have their $primarykey property set to true are used in Where clauses for automatically generated Update and Delete statements. OMNIS automatically sets $useprimarykeys to kTrue when defining the list, if and only if there is at least one primary key column in the list

– **$extraquerytext**
a text string appended to automatically generated SQL; used by the $select(), $selectdistinct(), $update(), $delete(), $doupdates() and $dodeletes() methods; for example, it can contain a Where clause. When the table instance is defined either directly or indirectly via a query class, OMNIS sets the initial value of this property from the query class; otherwise, this property is initially empty

– **$servertablenames**
a comma-separated list of the names of the server tables or views referenced by the schemas associated with the table instance. If the table instance uses a schema class to define its columns, there is only one name in $servertablenames. If the table instance uses a query class, there can be more than one name, corresponding to the schemas referenced by the query class, and in the order that the schemas are first encountered in the query class

– **$sessionname**
the name of the OMNIS session to the server, on which the table instance methods execute their SQL; if empty, OMNIS issues the SQL on the current session

– **$colsinset**
the number of columns in the current result set for the session used by the table instance; this property is NOT assignable

– **$rowsaffected**
the number of rows affected by the last call to $insert(), $update(), $delete(), $doinserts(), $doupdates(), or $dodeletes()

– **$rowsfetched**
the number of rows fetched so far, using the $fetch() method, in the current result set for the session used by the table instance

– **$allrowsfetched**
set to kTrue when all rows in the current result set for the current table instance have been fetched, otherwise kFalse at other times

List columns in a list containing a table instance have two table instance related properties: $excludefromupdate and $excludefrominsert. When true, the column is omitted from the result of $updatenames or $insertnames, and from the list of columns in the SQL statements generated by $update or $insert. Note that the column is not omitted from the where clause generated by $update and $updatenames, therefore allowing you to have a column which is purely a key and not updated. For example:

```
Do MyList.$cols.MyKey.$excludefromupdate.$assign(kTrue)
```

The default setting of this property is kFalse. If you define a list from a SQL class and use $add to add additional columns, you cannot set these properties for the additional columns.

Table Instance Methods

The following methods use the list variable MyList or row variable MyRow which can be based on a schema, query, or table class.

$select()

```
Do MyList.$select([parameter-list]) Returns STATUS
```

$select() generates a Select statement and issues it to the server. You can optionally pass any number of parameters which OMNIS concatenates into one text string. For example, *parameter-list* could be a Where or Order By clause. The method returns kTrue if the table instance successfully issued the Select.

The $select() method executes the SQL statement equivalent to

```
Select [$cinst.$selectnames()] from [$cinst.$servertablenames]
   [$extraquerytext] [parameter-list]
```

The following $construct() method for a SQL form defines a row variable and builds a select table. The form contains an instance variable called iv_SQLData with type Row.

```
Set current session {session-name}
Do iv_SQLData.$definefromsqlclass('schema-name')
Do iv_SQLData.$select()
```

$selectdistinct()

```
Do MyList.$selectdistinct([parameter-list]) Returns STATUS
```

$selectdistinct() is identical in every way to $select(), except that it generates a Select Distinct query.

$fetch()

```
Do MyList.$fetch(n[,append]) Returns STATUS
```

$fetch() fetches up to *n* rows of data from the server into the list, or for row variables fetches the next row. If there are more rows available, a subsequent call to fetch will bring them back, and so on. The $fetch() method returns a constant as follows

kFetchOk	OMNIS fetched n rows into the list or row variable
kFetchFinished	OMNIS fetched fewer than n rows into the variable; this means that there are no more rows to fetch
kFetchError	an error occurred during the fetch; in this case, OMNIS calls $sqlerror() before returning from $fetch(), and the list contains any rows fetched before the error occurred

When fetching into a list, if the Boolean append parameter is kTrue, OMNIS appends the fetched rows to those already in the list; otherwise, if append is kFalse, OMNIS clears the list before fetching the rows. If you omit the append parameter, it defaults to kFalse.

The following method implements a Next button on a SQL form using the $fetch() method to fetch the next row of data. The form contains the instance variables iv_SQLData and iv_OldRow both with type Row.

```
; declare local variable lv_Status of Long integer type
On evClick
   Do iv_SQLData.$fetch() Returns lv_Status
   If lv_Status=kFetchFinished|lv_Status=kFetchError
      Do iv_SQLData.$select()
      Do iv_SQLData.$fetch() Returns lv_Status
   End If
   Calculate iv_OldRow as iv_SQLData
   Do $cwind.$redraw()
```

$insert()

```
Do MyRow.$insert() Returns STATUS
```

$insert() inserts the current data held in a row variable into the server database. It returns kTrue if the table instance successfully issued the Insert. The $insert() method executes the SQL statement equivalent to

```
Insert into [$cinst.$servertablenames] [$cinst.$insertnames()]
```

The following method implements an Insert button on a SQL form using the $insert() method to insert the current value of the row variable. The form contains the instance variable iv_SQLData with type Row.

```
On evClick
   Do iv_SQLData.$insert()  ;; inserts the current values
```

$update()

```
Do MyRow.$update(old_row[,disable_where]) Returns STATUS
```

$update() updates a row in a server table from the current data held in a row variable. It returns kTrue if the table instance successfully issued the Update. Note that if the SQL statement identifies more than one row, each row is updated.

The *old_row* parameter is a row variable containing the previous value of the row, prior to the update.

The optional *disable_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then OMNIS excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

The $update() method executes the SQL statement equivalent to

```
Update [$cinst.$servertablenames][$cinst.$updatenames('old_row')]
   [$extraquerytext]
```

The following method implements an Update button on a SQL form using the $update() method. The form contains the instance variables iv_SQLData and iv_OldRow both with type Row.

```
On evClick
   Do iv_SQLData.$update(iv_OldRow)
```

$delete()

```
Do MyRow.$delete([disable_where]) Returns STATUS
```

$delete() deletes a row from a server table, matching that held in the row variable. It returns kTrue if the table instance successfully issued the Delete. Note that if the SQL statement identifies more than one row, each row is deleted. The optional *disable_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then OMNIS excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

The $delete() method executes the SQL statement equivalent to

```
Delete from [$cinst.$servertablenames] [$cinst.$wherenames()]
   [$extraquerytext]
```

Note that [$cinst.$wherenames()] is omitted by setting disable_where to kTrue.

The following method implements a Delete button on a SQL form using the $delete() method. The form contains the instance variable iv_SQLData with type Row.

```
On evClick
   Do iv_SQLData.$delete()
   Do iv_SQLData.$clear()
   Do $cwind.$redraw()
```

$doinserts()

```
Do MyList.$doinserts()Returns MyFlag
```

This method only works for smart lists. $doinserts() inserts rows with status kRowInserted in the history list, into the server table, and returns kTrue if the table instance successfully issued the Inserts. $doinserts() calls $doinsert() once for each row to be inserted. $doinserts() then accepts the changes to the smart list, unless an error occurred when doing one of the Inserts.

$dodeletes()

```
Do MyList.$dodeletes([disable_where])Returns MyFlag
```

This method only works for smart lists. $dodeletes() deletes rows with status kRowDeleted in the history list, from the server table, and returns kTrue if the table instance successfully issued the Deletes. $dodeletes() calls $dodelete() once for each row to be deleted. $dodeletes() then accepts the changes to the smart list, unless an error occurred when doing one of the Deletes. The optional *disable_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then OMNIS excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

$doupdates()

```
Do MyList.$doupdates([disable_where]) Returns MyFlag
```

This method only works for smart lists. $doupdates() updates rows with status kRowUpdated in the history list, in the server table, and returns kTrue if the table instance successfully issued the Updates. $doupdates() calls $doupdate() once for each row to be updated. $doupdates() then accepts the changes to the smart list, unless an error occurred when doing one of the Updates. The optional *disable_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then OMNIS excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

$dowork()

```
Do MyList.$dowork([disable_where]) Returns MyFlag
```

This method only works for smart lists. $dowork() is a shorthand way to execute $doupdates(), $dodeletes() and $doinserts(), and returns kTrue if the table instance successfully completed the three operations. The optional *disable_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then OMNIS excludes the where clause from the automatically generated SQL for $dodeletes() and $doupdates(). This may be useful if you want to pass your own where clause using $extraquerytext.

$doinsert()

```
$doinsert(row)
```

$doinsert inserts the row into the server database. The default processing is equivalent to

```
row.$insert()
```

$dodelete()

```
$dodelete(row)
```

$dodelete deletes the row from the server database. The default processing is equivalent to

```
row.$delete()
```

$doupdate()

```
$doupdate(row,old_row)
```

$doupdate updates the row in the server database, using the old_row to locate the row. The default processing is equivalent to

```
row.$update(old_row)
```

$undoinserts()

```
Do MyList.$undoinserts() Returns MyFlag
```

This method only works for smart lists. $undoinserts() undoes the Inserts to the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistinserts().

$undodeletes()

```
Do MyList.$undodeletes() Returns MyFlag
```

This method only works for smart lists. $ undodeletes() undoes the Deletes from the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistdeletes().

$undoupdates()

```
Do MyList.$undoupdates() Returns MyFlag
```

This method only works for smart lists. $undoupdates() undoes the Updates to the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistupdates().

$undowork()

```
Do MyList.$undowork() Returns MyFlag
```

This method only works for smart lists. $undowork() undoes the changes to the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistwork().

$sqlerror()

```
Do MyList.$sqlerror(ERROR_TYPE, ERROR_CODE, ERROR_TEXT)
```

OMNIS calls $sqlerror() when an error occurs while a default table instance method is executing SQL. The default $sqlerror() method performs no processing, but you can override it to provide your own SQL error handling. It passes the parameters:

ERROR_TYPE	indicates the operation where the error occurred: kTableGeneralError, kTableSelectError, kTableFetchError, kTableUpdateError, kTableDeleteError or kTableInsertError.
ERROR_CODE	contains the SQL error code, as returned by *sys(131)*.
ERROR_TEXT	contains the SQL error text, as returned by *sys(132)*.

$selectnames()

```
Do MyList.$selectnames() Returns SELECTTEXT
```

Returns a text string containing a comma-separated list of column names in the list variable in the format:

```
TABLE.col1,TABLE.col2,TABLE.col3,...,TABLE.colN
```

The returned column names are the server column names of the list columns in the order that they appear in the list, suitable for inclusion in a SELECT statement; also works for row variables. Each column name is qualified with the name of the server table.

$createnames()

```
Do MyList.$createnames() Returns CREATETEXT
```

Returns a text string containing a comma-separated list of server column names and data types for each column in the list variable, suitable for inclusion in a CREATE TABLE statement; also works for row variables. The returned string is in the format:

```
col1 COLTYPE NULL/NOT NULL,col2 COLTYPE NULL/NOT NULL,
   col3 COLTYPE NULL/NOT NULL,...,colN COLTYPE NULL/NOT NULL
```

The NULL or NOT NULL status of each column is derived from the $nonull property in the underlying schema class defining the column.

$updatenames()

```
Do MyRow.$updatenames() Returns UPDATETEXT
```

Returns a text string in the format:

```
SET TABLE.col1=@[$cinst.col1],TABLE.col2=@[$cinst.col2],
   TABLE.col3=@[$cinst.col3],...,TABLE.colN=@[$cinst.colN]
```

where col1…coln are the server column names of the columns in the row variable. Each column name is qualified with the name of the server table.

```
Do MyRow.$updatenames([old_name]) Returns UPDATETEXT
```

The optional parameter *old_name* is the name of a row variable to be used to generate a ‘where’ clause. If you include *old_name*, a ‘where’ clause is concatenated to the returned string in the following format:

```
WHERE col1=@[old_name.col1] AND ... AND colN=@[old_name.colN]
```

The columns in the where clause depend on the setting of $useprimarykeys. If $useprimarykeys is kTrue, then the columns in the where clause are those columns marked as primary keys in their schema class. Otherwise, the columns in the where clause are all non-calculated columns except those with data type picture, list, row, binary or object.

You can replace $cinst in the returned string using:

```
Do MyRow.$updatenames([old_name][,row_name]) Returns UPDATETEXT
```

where *row_name* is the name of row variable which OMNIS uses in the bind variables. This may be useful if you override $doupdate() for a smart list.

$insertnames()

```
Do MyRow.$insertnames() Returns INSERTTEXT
```

Returns a text string with the format:

```
(TABLE.col1,TABLE.col2,TABLE.col3,...,TABLE.colN) VALUES
   (@[$cinst.col1],@[$cinst.col2],@[$cinst.col3],...,@[$cinst.colN])
```

where col1...colN are the server column names of the columns in the row variable. The initial column names in parentheses are qualified with the server table name. You can replace $cinst in the returned string using:

```
Do MyRow.$insertnames([row_name]) Returns INSERTTEXT
```

where *row_name* is the name of row variable which OMNIS uses in the bind variables. This may be useful if you override $doinsert() for a smart list.

$wherenames()

```
Do MyRow.$wherenames() Returns WHERETEXT
```

Returns a text string containing a Where clause in the format:

```
WHERE TABLE.col1=@[$cinst.col1] AND TABLE.col2=@[$cinst.col2] AND
   TABLE.col3=@[$cinst.col3] AND ... TABLE.colN=@[$cinst.colN]
```

where col1...colN are the server column names of the columns in the row variable. Each column name is qualified with the server table name.

The columns in the where clause depend on the setting of $useprimarykeys. If True, then the columns in the where clause are those columns marked as primary keys in their schema class. Otherwise, the columns in the where clause are all non-calculated columns except those with data type picture, list, row, binary or object.

The = operator in the returned string is the default, but you can replace it with other comparisons, such as < or >=, by passing them in the *operator* parameter.

```
Do MyRow.$wherenames([operator]) Returns WHERETEXT
```

You can replace $cinst in the returned string using:

```
Do MyRow.$wherenames([operator][,row_name]) Returns WHERETEXT
```

where *row_name* is the name of row variable which OMNIS uses in the bind variables. This may be useful if you override $dodelete() for a smart list.

If you want to see the SQL generated by the table instance SQL methods, you can use the command *Get SQL script* to return the SQL to a character variable after you have executed the SQL method. Note that the returned SQL will contain bind variable references which do not contain $cinst. This is because *Get SQL script* does not execute in the same context as the table instance. However, you will be able to see the SQL generated, which should help you to debug problems.

Chapter 15—Server-Specific Programming

This chapter contains server-specific information for each of the proprietary databases and middleware configurations that OMNIS Studio supports.

Oracle

This section contains the additional information you need to access an Oracle database, including server-specific programming and PL/SQL, troubleshooting and data type mapping to and from Oracle. For general information about logging onto Oracle and managing your database using the SQL Browser, refer to the earlier parts of this manual.

Server-specific Programming

Almost every DBMS has its own specific, extra features that are not part of the SQL standard. You can take advantage of many of these through *server-specific programming*. OMNIS provides special keywords and the *Server specific keyword* command to assist you in this. Server-specific keywords are single-word commands enclosed in angle braces, such as <NULLASEMPTY>.

You use the *Server specific keyword* command to send a keyword command to your server.

```
Server specific keyword { <DESCRIBETABLES> ALL}
```

This command, after evaluating square bracket notation, sends the string to the DAM, which translates the command into the appropriate server instructions. The string often includes parameters.

The Oracle DAM supports the following keywords.

– <DESCRIBETABLES> ALL|USER
  If ALL is specified the Describe tables command will return all tables in the database. If USER is specified the Describe tables command will only return the names of the tables owned by the current user.

– <NULLASEMPTY> ON|OFF
  If ON is specified any NULLs retrieved from Oracle will be treated as empty instead of NULL. If OFF is specified NULLs are retrieved as NULLs.

– <TRAILINGSPACES> ON|OFF
If OFF is specified any trailing spaces on data being inserted are stripped. If ON is specified trailing spaces are kept. By default trailing spaces are stripped (OFF).

Updating and Deleting Specific Rows

You can use positioned updates and deletes to update and delete specific rows from the select table you are fetching. To enable positioned updates and deletes, declare and open a cursor for your SELECT statement:

```
Declare cursor EMP_CURSOR for SELECT * FROM EMP FOR UPDATE
Open cursor { EMP_CURSOR }
```

As you fetch individual rows using the cursor, you can decide whether or not to update or delete the current row with the "WHERE CURRENT OF" syntax. This language takes the session name of the session in which you executed the FOR UPDATE clause. For example, if you declared the above cursor and opened it, then started fetching rows, you could update the last row fetched using these statements:

```
; fetch the next row
Set current cursor { EMP_UPDATE_CURSOR }
Perform SQL{UPDATE EMP SET DEPTNO = 5 WHERE CURRENT OF EMP_CURSOR }
```

You need to change the current cursor in order to use the original cursor (EMP_CURSOR) to do the update. You can delete the row using this statement:

```
; fetch the next row
Set current cursor { EMP_DELETE_CURSOR }
Perform SQL { DELETE FROM EMP WHERE CURRENT OF EMP_CURSOR }
```

You must fetch at least one row before using WHERE CURRENT OF in a positioned update or delete statement. You cannot use this syntax after issuing a *Build list from select table* command, since you must fetch the rows one at a time to set the cursor to a current row. You must be in the same session in which you declared the cursor in order to use the WHERE CURRENT OF clause; if you have changed sessions, you must change back using the *Set current session* command.

You need to *Set transaction mode (Generic)* before using these commands. Do not use the OMNIS Autocommit feature *Autocommit (On)* or *Set transaction mode (Automatic)*.

ORACLE lets you select multiple tables for update:

```
Declare cursor EMP_DEPT_CURSOR for SELECT * FROM EMP, DEPT FOR
   UPDATE
Open cursor { EMP_DEPT_CURSOR }
```

You can then update each table separately:

```
Perform SQL {UPDATE EMP SET DEPTNO = 5 WHERE CURRENT OF
   EMP_DEPT_CURSOR}
Perform SQL {UPDATE DEPT SET DNAME = 'Quality Assurance' WHERE
   CURRENT OF EMP_DEPT_CURSOR}
```

PL/SQL

The ORACLE DAM fully supports ORACLE PL/SQL, which is a procedural language that the server executes. You create a PL/SQL script, send it to ORACLE in the same way as any SQL statement, and the server executes it.

The DAM improves PL/SQL performance by deferring binding for parsing SQL and PL/SQL statements. It also reduces numeric conversions between OMNIS and ORACLE variables.

With PL/SQL support, you can establish updateable bind variables. That is, you can include OMNIS variables in a PL/SQL statement, and when the PL/SQL statement executes, it updates the OMNIS variables with a new value. For example:

```
BEGIN
SELECT id INTO :XINT FROM Emp WHERE name = ‘Jones’;
END;
```

In this example, ':XINT' is a PL/SQL bind variable associated with the OMNIS variable 'XINT'. After the PL/SQL executes, the OMNIS variable XINT has the value associated with the column id from the row with the name ‘Jones’.

Since ORACLE PL/SQL variable names and OMNIS variable names use a different set of characters, the DAM needs to remap some OMNIS variables to be able to refer to them in PL/SQL. This is automatic, but there are some mappings that you should be aware of. The DAM translates all occurrences of '.' in OMNIS variable names to '_'. In addition, for hash variables, the DAM translates '#' to a 'z'.

When you create a PL/SQL program and pass OMNIS variables to the ORACLE Server, the input and output sizes of the variables may be different for character, picture, and binary fields; the initial size may even be zero. OMNIS must allocate a buffer for each updateable bind variable. The size of the buffer is set as the greater of the current size of the variable and the default minimum size (256 bytes for Character; 1,024 bytes for Picture and Binary). In some cases, you may need a variable that is much larger than either the default minimum or the current size of the variable. In this case, you can put the size of the buffer after the bind variable, like this:

```
:myvar:2048
```

The DAM cannot return OMNIS picture fields from PL/SQL through updateable bind variables. You can bring picture fields into OMNIS by means of a normal SQL select statement.

The Oracle DAM supports select tables returned through PL/SQL procedures. However, Oracle can only return single column tables. To bind a column of a list to the select table being returned you must use the following:

:list_name(num_rows).column_name

where num_rows is the maximum number of rows to bind. Specifying the number of rows is optional. If the number of rows is not specified then the maximum number that available memory permits is returned. To let the DAM calculate the maximum number of rows simply omit (num_rows) from the above. Note that if the procedure returns more rows than there are allocated, an error results.

You cannot execute a stored procedure on Oracle directly. For example, the following will not work:

```
Perform SQL: EXECUTE METHOD(PAR)
```

To execute such a procedure you must use the full SQL that PL/SQL requires. The statement would be:

```
Perform SQL: BEGIN EXECUTE METHOD(PAR); END;
```

The ORACLE DAM lets you send and receive null data, both in SQL statements and in PL/SQL programs. This works for both input and output variables. You can set the null display feature for the OMNIS fields to see the results.

The ORACLE DAM concatenates multiple error messages to give you more information when possible.

Server Information

The *server()* function takes a parameter in which you request information from the DAM about the server. You can use the *Calculate* command to place the result in an OMNIS variable:

```
Calculate RESULT as server('Version') ;; Returns the version number
   of the active DAM

Calculate PATH as server('Path') ;; Directory path of the DAM

Calculate API as server('vendorAPI') ;; Directory path of server API
   if available
```

Every DAM, including the Oracle DAM, supports the following *server()* parameters.

– **Version**
the version string, same as sys(130)

– **Vendorapi**
the version string of the client API with which the DAM compiled

– **Path**
the file path to the DAM

– **File**
the name of the DAM file

– **DAM**
the name of the DAM

In addition, the Oracle DAM supports the following *server()* parameters:

– **GETDESCRIBETABLES**
whether Describe table returns ALL tables or just the USER's tables

– **GETTRANSACTIONMODE**
the current transaction mode - AUTOMATIC, GENERIC or SERVER

Troubleshooting

The following points may help in resolving issues in programming OMNIS applications that use an Oracle database.

– If you connect to DB2 or SQL/DS through an ORACLE Gateway using the ORACLE DAM, there is a 70-character limit for error strings

– If you bind an OMNIS Character or National variable to a CHAR, you should note the ORACLE rules for comparing CHAR data. ORACLE will pad out the strings to the same size with blanks, which might lead to unexpected results. See the ORACLE *SQL Reference Manual* for details.

– Attempting to set batchsize higher than 255 causes method execution to halt in the debugger.

– OMNIS supports blob fields of up to 100MB (for binary, or picture fields). However, in practice you may have memory problems under MacOS which puts a real limit of about 4 to 8MB on blob fields.

Platform Specific Issues

– For characters to be read between MacOS and Windows on Oracle, the Mac character set has to be used in the **Oracle.ini** file under Windows and in the registry under WINDOWS 95 and Windows NT. However, if the Mac character set and the server character set are different not all characters may be successfully converted and so will be represented by a default character chosen by the server.

– The Oracle DAM on NT can send and receive BLOBs of up to 100MB. However, in practice the real limit may be less depending on the available memory.

Data Type Mapping

The following tables describe the data type mapping for OMNIS and Oracle.

OMNIS to ORACLE

OMNIS Data Type	ORACLE Data Type
CHARACTER	
Character/National <= 255	VARCHAR(n)
Character/National > 255	VARCHAR(n)
Character/National 2100	LONG
DATE/TIME	
Short date (all subtypes)	DATE
Short time	DATE
Date time (#FDT)	DATE
NUMBER	
Short integer (0 to 255)	NUMBER(3, 0)
Long integer	NUMBER(11, 0)
Short number 0dp	NUMBER(10, 0)
Short number 2dp	NUMBER(10, 2)
Number floating dp	FLOAT
Number 0..14dp	NUMBER(16, 0..14)
OTHER	
Boolean	VARCHAR2(3)
Sequence	NUMBER(11, 0)
Picture	LONG RAW
Binary	LONG RAW
List	LONG RAW
Row	LONG RAW
Object	LONG RAW
Item reference	LONG RAW

ORACLE to OMNIS

Server Data Type	Describe Data Type	OMNIS Data Type
CHARACTER		
CHAR	Char	Character
VARCHAR2	Character	Character
LONG	Long	Character
RAW	Binary	Binary
LONG RAW	Picture	Picture
DATE/TIME		
DATE	Datetime	Date time (#FDT)
NUMBER		
NUMBER	Number(10,14)	Number floating dp
NUMBER(10)	Number(10,14)	Number floating dp
NUMBER(10, 8)	Number(10,14)	Number floating dp
NUMBER(18)	Number(10,14)	Number floating dp
NUMBER(16, 15)	Number(10,14)	Number floating dp
NUMBER(18, 15)	Number(10,14)	Number floating dp
FLOAT	Number(10,14)	Number floating dp
FLOAT(30)	Number(10,14)	Number floating dp

Sybase

This section contains the additional information you need to access a Sybase database, including server-specific programming, troubleshooting, and data type mapping to and from Sybase. For general information about logging on to Sybase and managing your database using the SQL Browser, refer to the earlier parts of this manual.

Server-specific Programming

Almost every DBMS has some special features you can take advantage of through *server-specific programming*. OMNIS provides special keywords and the *Server specific keyword* command to assist you in this. Server-specific keywords are single-word commands enclosed in angle braces, such as <WRITEBLOB>.

You use the *Server specific keyword* command to send a keyword command to your server.

```
Server specific keyword { <WRITEBLOB> }
```

This command, after evaluating square bracket notation, sends the string to the DAM, which translates the command into the appropriate server instructions. The string often includes parameters. For example, the following command, issued to a Sybase session, sets the W_Tours/Error method as the error message handler for the session.

```
Server specific keyword { <SQLMESSAGE> W_Tours/Error }
```

The Sybase DAM supports the following keywords.

– <**CALLERRORHANDLER**>
  calls the current error handler for the session or the global error handler if there is no session handler

– <**CALLMESSAGEHANDLER**>
  calls the current message handler for the session or the global message handler if there is no session handler

– <**DBCANCEL**>
  discards all pending results in select tables; faster than Reset session; see section below for warning on use

– <**DBCANQUERY>**
  discards a single pending result table

– <**RPC**>
  defines and executes a remote procedure call: see section below

– <**RPCPASSWORD**>
  sets a password for a remote connection; see below

– <**RPCRESULTS**>
processes output from a remote procedure call; see below

– <**SETERRORHANDLER>**
sets an OMNIS method as session error handler

– <**SETMESSAGEHANDLER>**
sets an OMNIS method as session message handler

– <**SETPROGRAMNAME**>
sets the Sybase program name column in the sysprocess table to the parameter

– <**SETTIMEOUT**>
sets the number of seconds that Open client will wait for Sybase to respond to a query; 0 sets the timeout to infinity

– <**SKIPEMPTYSETS**>
ignores empty select tables in a multiple-table sequence; see the Sybase section below for details

– <**SQLERROR**>
sets an OMNIS method as global error handler

– <**SQLMESSAGE**>
sets an OMNIS method as global message handler

– <**SETENCRYPT_ON**>
sets password encryption on logon

– <**SETENCRYPT_OFF**>
disables password encryption on logon (Default)

– <**WRITEBLOB**>
updates a binary large object (blob) in a Sybase database; see the section below for details.

Multiple Select Tables

The Sybase DAM lets you run more than one SQL SELECT statement at once and return multiple select tables, one after the other. You need to do some special processing in order to process all the tables. Ordinarily, when the flag turns false on a *Fetch next row*, it means there is no more data. With multiple select tables, there may be more data. There is a special system function, *sys(138),* which is true if there are more rows (regardless if the flag is false) and false if there are no more rows. As you process several select tables, the flag turns false when you retrieve the last row from a table.

The *Build list from select table* command retrieves data into a list up to the limit #LM, the maximum number of items in list. You can thus get a list full, flag true, and *sys(138)* true, which means there are more rows in the select table to fetch because you filled up the list. If

the flag is false, you have exhausted the current select table, but *sys(138)* can still indicate more rows, meaning there are more select tables.

One way to do multiple SELECTs is to code a procedure containing several SELECT statements that is to execute on the server. The example below executes the following method (assuming the number of lines in your list is adequate to contain all the rows returned):

```
create proc multi_select as
SELECT firstName, lastName FROM Agents
SELECT id, name FROM Customers
```

To execute this from an OMNIS method and to fetch the results,

```
Perform SQL { exec multi_select }
Set current list { My_list1 }
Define list { aFirstName, aLastName }
Build list from select table
If #F = kFalse & sys(138)  = 1  ;; flag false and sys(138) = 1
   Set current list { My_list2 }
   Define list { anID, aName }
   Build list from select table
End if
```

You need to be careful about handling errors from the server for each SQL statement, or you may lose the error before seeing it. Sometimes a server will return multiple errors and OMNIS will not be able to tell you because only the last one gets passed through to you through the error handling mechanism (see below). You should use multiple SQL statements sparingly for this reason.

The sysprocess table (in the master database) has a program_name column that stores a separate name for each connection to the server. The <SETPROGRAMNAME> server-specific keyword lets you put a name into this column for the current session that you can then use to distinguish multiple sessions.

The default name is the name of the library that first loaded the DAM. If you have already set this name with the keyword, you can restore the default value by calling the keyword with no parameter.

You must use this keyword after starting a session but before logging on to the server, because the value gets set at logon: using the keyword after logon results in an error. The value persists across logons and logoffs, and *Reset current cursor* does not reset it.

If the string passed to the keyword is too long, the DAM truncates it without reporting an error. The DAM stores it internally as a thirty-character string but the program_name field on the SQL Server is shorter than this. Finally, the DAM allows spaces and eight-bit characters in the string and ignores spaces between the keyword and the first non-space character of the name.

The server-specific command <SKIPEMPTYSETS> lets you process multiple-select-table queries more effectively by letting you skip over any empty tables in the sequence of multiple tables.

The default setting for the parameter is NO, which means that the Sybase DAM reports the select tables exactly as Open Client reports them. When this is called with the “YES” parameter, this causes the DAM to skip over all select tables that have no rows in them. The effect, for the developer, is that the empty tables do not exist.

The NOCOLUMNS parameter tells the DAM to skip only those select tables that have no columns and no rows of data. The examples above generate select tables that have no columns or rows. This query,

```
SELECT * from sysobjects where 1 = 2
```

generates a select table with no rows and seventeen columns. There are times where being able to distinguish between sets without columns and with columns is useful.

This setting affects only a single session, and you can toggle it on and off at any time after starting the DAM. Toggling it while fetching data sets may produce unexpected results due to the ordering of the operations within the DAM itself. When the session is logged off, this parameter is reset to NO. The *Reset session* command does not affect this setting.

Error Handling

Because one OMNIS command’s action can generate multiple errors or messages, there is an issue relating to which of these gets stored in *sys(131)* and *sys(132)*, which can report only one code and string per action. Whenever the server reports an error or a message to the DAM, it uses the following rules:

– Every error and message has a severity code, so the DAM stores an error or a message only if its severity is greater than or equal to the one already stored (if any)

Errors and messages use slightly different severity scales (1-11, and 0,11-24, respectively), so the DAM treats all messages as if they were between severity 5 and 6 on the error handling scale. The ranking, from least to most important, is therefore:

1. Severity 1–5 errors
2. Severity 11–24 messages
3. Severity 6–11 errors

– The *sys()* functions never report messages with a severity of 0 (informational messages, including ones produced with the Transact-SQL *print* command); these functions also never report the Open Client generated error of severity 5 (20018 on MacOS, 10007 on Windows) that indicates a message just arrived

– *sys(131)* contains the ErrorNumber, OSErrorNumber, or MessageNumber code, depending on the rules above.

– *sys(132)* contains the severity value, followed by a colon, followed by one of three things:
    the MessageString for a message
    the ErrorString for an error
    the OSErrorString followed by a colon then the ErrorString for an OS error.

Each time you call the DAM, it clears the information in sys*(131)* and sys*(132)* so that it doesn't cause errors from one action to be associated with a later one.

You should be aware that the errors reported by the error handler have codes and messages that sometimes differ between the MacOS and Windows Sybase clients.

Instead of relying on this limited capability, you can declare error or message handlers in OMNIS to capture all the relevant information directly from the DAM. The DAM invokes these handlers as soon as it receives notification of the error or message from the Sybase Open Client. Hence the DAM usually calls them while you are waiting for a SQL command to return in OMNIS. For example, you might issue a *Logon to host* command, and the DAM might call your message handler multiple times before returning from the command.

An exception is severity 0 messages; sometimes the server does not report these until you have finished fetching rows from the select table. Here is a handler that reports all messages directly to the user. Thus:

```
; Declare parameter vars SessionName, MessageString,
; ServerName, MethodName of Character type
; Declare parameter vars MessageNumber, MessageState,
; Severity, LineNumber of Long Integer type
OK message {con(Message: ",MessageString," (Code=", MessageNumber,")
   (Severity=",Severity,")")}
```

You can then set this method (W_Tours/ErrHand, say) as the error handler for your session:

```
Set hostname {myserver}
Set username {me)
Set password {secret}
Server specific keyword {<SETMESSAGEHANDLER> W_Tours/ ErrHand}
Logon to host
```

When you log on, your application greets you with an OK box declaring:

```
Message: Changed database context to 'mydatabase'. (Code=5701)
   (Severity=0)
```

You can implement a wide variety of things in an error or message handler. You might choose to report severe errors to the user, or to set flags to cause your library to behave differently based on certain codes from the server. Other things you might want to do would be to store all incoming messages in an error log that you can review at any time in your application. Since you can install and remove handlers at any time, you might want to turn on your handlers in particularly tricky parts of your code. Alternatively, you can set up

separate handlers for each session and have them report their messages in different ways appropriate for each session's role in the library.

There is a limit of 250 bytes when the DAM calls the handler. If the combination of parameters is too long, then the DAM may truncate the message or oserror and dberror text. If truncation occurs, then the text ends with an asterisk (*).

The Sybase error handler is unusual in that you can return a code in one particular instance. If and only if the error is a timeout error (20003 under MacOS, 10024 under Windows) for a statement, then you can return either 2 to continue waiting or 1 to cancel the operation.

WARNING The effect of returning a 1 varies between the Windows and MacOS Open Client. You will probably *never* want to return a value of 1. Under MacOS, this causes Open Client to kill the dbprocess, thus terminating your connection to the SQL Server. On Windows, this merely cancels the action and discards the results.

The DAM uses the hash variable #1 when it retrieves this return code error from OMNIS., You must therefore refrain from using #1 while in the error handler.

Establishing a timeout with the server-specific keyword <SETTIMEOUT> can be useful, therefore, because you can use the errors to let the user know that a long query is, in fact, doing something; but using it to kill the query is more problematic.

If you place breakpoints of any sort in a handler method, then when OMNIS encounters the breakpoint, it will clear the stack and clear the method's variables.

You should not do too much in a handler. If you have work that involves adding scores of things to lists or calling dozens of other methods, it is usually best to set a flag while in the handler then do this work after you return to the main flow of your library. OMNIS can get confused if the handler gets too complex due to the complexity of handling a call back from the DAM while it is, itself, waiting for the DAM to return.

Blobs

The Sybase DAM can send and retrieve text and image fields that are larger than 32K (*BLOBs*, or "binary large objects"). When dealing with BLOBs, you need to take some limitations into account.

Transferring BLOBs is very memory-intensive, since each layer of software has a copy of at least part of the blob. Thus, sending a simple 40K picture can demand several times that amount of RAM before it gets passed over to the DBMS.

The Sybase DAM provides different ways to send and fetch BLOBs. You can send or fetch the fields using the standard OMNIS commands, with certain limitations; or you can use the <WRITEBLOB> keyword to update a text or image field on the DBMS, with the implicit functionality to retrieve a large text or image field.

The standard SQL commands (*Perform SQL*, *Fetch next row*, and so on) let you pass large character, picture, list or binary fields by binding them into SQL statements or mapping

SQL results into OMNIS fields. There are, however, platform-specific limits on how large the values can be.

Under Windows, you cannot have values larger than 32,767 bytes. Sending a larger value generates an error message (31,955 for OMNIS-to-Sybase, 31,966 for Sybase-to-OMNIS) meaning that it was unable to send the value, after which the current transaction is aborted. This means that the DAM discards any text sent since the last *Execute SQL script* or *Perform SQL*. For data being retrieved, the DAM returns a NULL value to the mapped OMNIS field.

Under MacOS, the limit is much less well defined. It is a combination of the memory you allocate to OMNIS and the DAM and of the limits on the size of a query that the SQL Server can process. You should be careful, therefore, to allocate sufficient memory for the DAM and OMNIS if you want to use BLOBs.

There are no limitations, aside from memory concerns, on sending multiple BLOBs in one SQL script.

To fetch a blob larger than these limits, you can select the particular column of values by itself, and then use normal retrieval operations such as *Fetch next row* or *Build list from select table*. This special fetching mode is otherwise completely invisible. For example, if you have an image column named myPhoto in a table called myTable:

```
Perform SQL {select myPhoto from myTable} ;; gets the whole blob
```

If, however, photoNum is an integer column in the same table, the command:

```
Perform SQL {select photoNum, myPhoto from myTable}
```

will return a NULL image column, since there are *two* columns in this select table.

This blob-fetching feature lets you retrieve BLOBs under MacOS in a way that uses less memory than the standard method. Retrieving BLOBs as single select-list elements is thus always a good idea for optimization.

No matter how you fetch the BLOBs, you should not forget to set the textsize parameter on the Server. This parameter tells the server not to truncate all outgoing values to this setting (see your Sybase documentation for more details). Therefore, if you set the textsize parameter to the default of 32,767 and select a 500K image, you get a 32,767 byte value in OMNIS.

```
Perform SQL { set textsize 123456 }
```

This sets the textsize parameter, for this session, to about 123K.

This setting is for fetching values only. When fetching BLOBs under MacOS using the standard commands, you should not set this parameter to its largest value. Increasing this also causes Open Client to allocate more memory to deal with the larger BLOBs. Therefore, setting it too small will truncate your fetched data while setting it too large may cause Open Client to kill your connection (resulting in a dead DBPROCESS). If you are retrieving a variety of BLOBs, you should try to set it as closely as you can to the size of the largest

blob; you can set this for each SQL script sent. You can also use the Sybase datalength() function to find out how long the value is that you want to retrieve, and use this to set the textsize parameter.

Under MacOS, you can increase the memory allocated to the DAM in two ways: When you install OMNIS, the installer prompts you to select an allocation size for the Sybase DAM. If you need to change this later, you can use the MacOS ResEdit utility to change the four numbers in the two SIZE resources in the DAM. You should set all four to be the same number, which is the number of bytes to allocate to the Sybase DAM. After doing this you should set the OMNIS memory allocation (through the Get Info command) to the sum of its current value and the increase in DAM memory allocation, since the DAM's memory is allocated out of OMNIS memory. If you expect to be building lists of BLOBs, you should give OMNIS even more memory.

Should you need to handle data larger than the limitations above allow, you can use the server-specific keyword <WRITEBLOB>. To use <WRITEBLOB>, you must already have the row in the database. <WRITEBLOB> updates data rather than inserting it, so you must qualify the keyword to identify the particular row. Also, the particular database column value must have data in it, not a NULL. You would usually create the row with a blank (' ') in the column, for instance, then use <WRITEBLOB> to update the value with the large blob.

```
Perform SQL {insert into mytable (x, mycol) values (2, ' ')}
Server specific keyword
   {<WRITEBLOB> /Use1 mytable.mycol, pictureField, where x = 2}
```

This command places the value of the OMNIS field pictureField into the column mycol of the table mytable in the row where x has the value of two. Thus, the keyword places a single blob value into a location that you specify. The clause after the name of the OMNIS blob field (in the example, this is the "where x = 2") uniquely specifies an existing row into which to place the blob. The syntax of the keyword is:

```
<WRITEBLOB> [/Log|/Nolog][/Use1|/Use2] table.column, OMNISField,
   where-clause
```

The **/Log or /Nolog** option denotes whether to log this action in the transaction log. If you do not log the action, you cannot roll it back. Using /Nolog requires that the select into/bulkcopy option be set to true with sp_dboption for the database on the DBMS. If you wish to use the /Nolog option, you must consult your documentation and system administrator since setting this option can have significant ramifications on being able to backup and recover your database. /Log is the default.

The **/Use1 or /Use2** option sets a single session or two sessions to write a blob. By default <WRITEBLOB> opens a second session when writing a blob, resulting in one or more "changed database context" messages. This lets you process a result set in the select table then write blob values back to the server in parallel, but with /Use1 you can force <WRITEBLOB> to use the same session. You should specify /Use1 unless you can guarantee that the table you are writing to will not have any transactions running against it,

including the main session. If you can't guarantee this, you risk a deadlock situation where, for example, the user might try to write a blob to a table that the main session locks. This call never returns, since the transaction never releases the lock on the first.

/Use1 discards any pending select tables and clears *sys(133)*, *sys(134)*, and *sys(138)*. You can actually specify the options in any order and in upper, lower, or mixed case. You must separate them by spaces. The commas between the parameters are optional.

The **table.column** parameter sets the table and column to which to write the blob.

OMNISField supplies the name of an OMNIS field from which to copy the blob.

The **where-clause** option supplies a where-clause for a select statement, including the word 'where'. If your where-clause is ambiguous OMNIS updates the first blob value it finds, so the value updated may not be the one you intended. Make sure your where clauses specifies a unique row.

If there are any errors, OMNIS sets the flag, *sys(131)*, and *sys(132)* appropriately.

<WRITEBLOB> reports an error and rolls back the transaction if you try to update a NULL value or a column that is other than a text or image type. The command cannot send NULL or empty fields itself, but it can send any field that is at least one byte long.

Remote Procedure Calls

The Sybase DAM provides commands that let you invoke Remote Procedure Calls (RPCs). When used with a SQL Server, these let you invoke a stored procedure without using the Transact SQL exec statement. An RPC has two advantages over the exec statement: it avoids converting the parameters to and from ASCII, thus potentially offering faster calling times, and more significantly, RPC commands let the DAM return output parameters from a stored method to your program automatically.

Using an RPC is easy. For example, the following statements invoke a stored procedure with one parameter. With an ISQL stored procedure call, use:

```
Perform SQL { exec MyStoredProc('[oField]') }
```

To do the same thing with an RPC:

```
Server specific keyword {<RPC> MyStoredProc(char:oField)}
```

The Sybase DAM provides three server-specific keywords related to issuing RPCs:

– **<RPC>**
  define and execute an RPC

– **<RPCRESULTS>**
  process output from an RPC

– **<RPCPASSWORD>**
  set passwords for remote connections

<RPC>

<RPC> defines and executes the remote procedure, and affects the flag, *sys(131)* to *sys(134)* and *sys(138).* Its arguments have the following format.

```
<RPC>name([parameter[;parameter]*])[status]
```

name is the name of the remote procedure you want to call. You can get access to group stored procedures by appending the number to the name; for example, to access procedure 4 in procedure group "helpme", the name would be "helpme;4"

parameter is one of up to 255 specifiers for the parameters to the remote procedure, in the format described below. If more than one is specified they must be separated by semicolons.

status holds the name of an OMNIS Long integer field into which the DAM places the return status code from the RPC (see <RPCResults>, below). If the remote procedure executes successfully without errors or does not explicitly set the return status, the return value is 0. This parameter is optional and has the format:

```
type:field[option]*
```

It is used as follows:

type is one of the following Sybase data types: bit, tinyint, smallint, int, smallmoney, money, real, float, char, varchar, binary, varbinary, smalldatetime, datetime, or binchar (see below). You cannot pass either image or text fields in an RPC call. This data type is what the Sybase DAM reports the parameter to the remote procedure as, and thus it should match the type defined in the remote procedure declaration.

field is the name of the OMNIS field that you wish to pass to the remote procedure. This field should have a type analogous to the type specified immediately before (for example, for tinyint, smallint, int, or float, types, you should use an OMNIS number field). However, OMNIS can do a wide variety of datatype conversions, and so you can actually get away with things like passing a long integer field as a char type. The DAM silently truncates fields with values longer than 255 bytes to that length when it sends them to the server.

An option for a parameter is a colon followed by one of the following three items. Note that all options are optional, and that you can specify them in any order.

@name is the name of the parameter in the remote procedure call. The DAM uses this parameter only if the order of the parameters in the <RPC> call is different from the order defined by the remote procedure. The character '@' is compulsory if you use this parameter.

OUTput determines whether the parameter can receive output values from Sybase. If the data specifier includes this constant, the DAM passes the field by reference and not by value. After the remote procedure executes successfully and the DAM processes all results rows, a call to <RPCRESULTS> updates these OMNIS fields with the return values. Note that only the first three letters of this option are significant. If you declare your parameters as OUTput, you *must* pass them to <RPC> in the same order that they were declared in the

remote procedure: the Sybase DAM *cannot* return output values if the parameters are not in the defined order.

ISNULL forces the value in the field to be treated as NULL. If the parameter includes this option, the DAM passes NULL to the remote procedure. This is generally an obsolete option since you can pass a NULL value directly by calculating the field to be #NULL.

While it is legal to pass no parameters to a remote procedure, you must still include the open and close parentheses. For example:

```
Server specific keyword { <RPC> JustDoIt() }
```

You may include spaces around any of the delimiters in the RPC argument string (that is, around '(', ')', ':', and ';') but you cannot pass bind variables with @[ ].

There are two phases involved in invoking an RPC with the Sybase DAM. In the first, the DAM parses the arguments to the <RPC> keyword, and locates the relevant OMNIS variables. If any errors occur, it will report information through the standard error handling mechanism about what the syntax error in the string was. Otherwise, it starts an RPC call to the server and passes the parameters to it. If any errors occur here, both the server and the DAM pass the error reports through the standard error handling system.

The binchar type is specific to the Sybase DAM. It lets you pass binary data as an ASCII hex string. This string must start with "0X" or "0x" followed by hexadecimal digits (for example, "0x001102FF008A45"). The <RPC> command converts this data to a binary value before passing it to the server.

If the following procedure is defined:-

```
create procedure appendbyte
   @parm varbinary(2) output
as
   select @parm = @parm+0x35
```

You could then issue the following <RPC> call:

```
; Local variable MYBINCHAR (Character  512)
Calculate MYBINCHAR as '0x34'
Server specific keyword  {<RPC>appendbyte (binchar:MYBINCHAR:OUT)}
Server specific keyword { <RPCRESULTS>}
```

After this, MYBINCHAR will contain the string "0x3435". You must take care when defining the character field for the type binchar. The maximum length allowed for data sent through the RPC interface is 255 bytes. When the data is retrieved, it is converted back to ASCII, which becomes 512 characters at maximum (255 bytes * 2 when converted to ASCII + 2 for the '0x').

<RPCRESULTS>

<RPCRESULTS> returns data into any parameters marked as OUTput in the <RPC> call. This also returns the return status value into the field specified in the <RPC> call. If the stored procedure did not return a status, the DAM puts a 0 into the target field. This keyword affects the flag, fields defined as OUTput in the <RPC> call, a field defined for return status, *sys(131)*, and *sys(132)*. <RPCRESULTS> takes no parameters.

Here is a more detailed example using both <RPC> and <RPCRESULTS>. Consider a table called person_data where the age for Karen Salt is 38.

```
create proc GetAge  @name varchar(32), @age int output as
   select @age = (select person_age from person_data
   where person_name = @name)
return 105
```

One could then invoke this as follows:

```
; Declare the following variables…
; Local variable THENAME (Character 32)
; Local variable THEAGE (Long integer)
; Local variable THESTATUS (Long integer)
Calculate THENAME as “Karen Salt”
Server specific keyword
   {<RPC> GetAge (varchar:THENAME; int:THEAGE:OUT) THESTATUS}
```

At this point, if you look at the value of THEAGE, it would contain the value it started out with. If the next line of the OMNIS method were:

```
Server specific keyword {<RPCRESULTS>}
```

then as soon as this command was executed, the OMNIS variables THEAGE would contain the number 38, and THESTATUS would contain 105.

If you have an RPC that returns a value to an output field and generates a result set with one row, you can call it with:

```
Server specific keyword {<RPC> my_proc (int:My_int:out)}
```

You should then fetch the RPC generated result set. If you issue only one fetch command and use <RPCRESULTS> you will get an error; you should keep fetching rows until #F=0.

<RPCPASSWORD>

<RPCPASSWORD> sets the password for remote servers and affects the flag, the session, *sys(131)*, and *sys(132)*. It has the syntax:

```
<RPCPASSWORD>server:password
```

server holds the name of the remote server, from the srvname column of its sysservers system table.

password is the password that the server to which you are logged on uses to access the server specified in the first part of this command.

An RPC or stored procedure can invoke procedures on another server. To do this, the server that the first procedure is running on must log onto the other server and for this you can set up a password to use when logging onto that server. This associates the specified password with the specified server for this connection to the DBMS. You can call this multiple times to establish passwords for different servers.

If the server name is empty (that is, you provide no text before the colon), then the password is a 'universal' password that you can use with any server for which you haven't already established a password.

You can put spaces before and after the colon: the DAM ignores these and does not consider them part of the servername or password.

Server Information

The *server()* function takes a parameter in which you request information from the DAM about the server. You can use the Calculate command to place the result in an OMNIS variable.

```
Calculate RESULT as server('Version') ;; Returns the version number
   of the active DAM

Calculate PATH as server('Path') ;; Directory path of the DAM

Calculate API as server('vendorAPI') ;; Directory path of server API
   if available
```

Every DAM handles the following set of *server()* parameters.

– **Version**
the version string, same as sys(130)

– **Vendorapi**
the version string of the client API with which the DAM compiled

– **Path**
the file path to the DAM

– **File**
the name of the DAM file

– **DAM**
the name of the DAM

In addition, the Sybase DAM supports the following server() parameters.

– **GETDATABASEVERSION**
returns the database version for current session; see section on Logging On above

- **GETERRORHANDLER**
  returns the name of the current session's error handler, if any, in standard format: name/number
- **GETLOGGEDON**
  returns Boolean YES if the session is logged on, NO if not
- **GETMESSAGEHANDLER**
  returns the name of the current session's message handler, if any, in standard format: name/number
- **GETPROGRAMNAME**
  returns the program name set with the <SETPROGRAMNAME> server-specific keyword or the name of the library that first loaded the DAM; no longer than thirty characters
- **GETSKIPEMPTYSETS**
  returns YES if you have set on the <SKIPEMPTYSETS> server-specific keyword, NO if not, and NOCOLUMNS if that option is set (see the keyword below for details)
- **GETSQLERROR**
  returns the name of the global error handler, if any, in the standard format: name/number
- **GETSQLMESSAGE**
  returns the name of the global message handler, if any, in the standard format: name/number
- **GETTIMEOUT**
  returns 0 if no timeout is set, otherwise returns the timeout period set with the <SETTIMEOUT> server-specific keyword
- **GETTRANSACTIONMODE**
  returns "AUTOMATIC", "SERVER", or "GENERIC" depending on the current transaction mode

Troubleshooting

The following points may help in resolving issues in programming OMNIS applications that use a Sybase database.

- Sybase is a case-sensitive RDBMS. Check the case of the table or column names if you can see a table but cannot select anything out of it
- Sybase defaults to NOT NULL columns; you must initialize columns to a specific value while inserting data, or insertion will fail
- Any number with no digits after the decimal point, that is > +/- 2^{31} will generate an error and not be inserted. This is because Sybase tries to parse numbers without decimal points as integers

- The Sybase DAM has the notion of a Primary Cursor which takes the name of the current session. This allows any SQL command, such as Select, Insert, Update or stored procedure to be invoked. Any additional cursors opened are regarded as secondary and allow only a single select statement to be issued, or a stored procedure returning only a single result set (from a single select statement). Results pending on the primary cursor will block the use of any secondary cursor.
- The Sybase DAM does not support the *Set SQL blob preferences* or *Set batchsize* command
- Sybase does not support binding a NULL Boolean field in OMNIS to a Sybase bit field
- Sybase does some character mapping, but you may need to do character conversion explicitly in your OMNIS code using character maps or other conversion functions
- Sybase interprets empty strings as single spaces.
- Fetching pictures from Sybase stored there by other applications, even in standard formats, is likely to cause problems, since OMNIS stores all pictures in a special format. This occurs even in platform-specific graphics formats such as PICT or BMP.
- The Sybase DAM reports errors both through the error handler and through *sys(131)* and *sys(132)* using error codes 31000-31999
- The Sybase DAM supports up to a maximum of twenty-five multiple sessions in one or more libraries simultaneously; note that some uses of the <WRITEBLOB> keyword may reduce the number (see below). The exact number may depend on your server configuration, so check with your System Administrator.
- The Sybase DAM does not ignore compute rows in select tables; if it encounters any, it simply skips over them to the next non-compute row
- If the Sybase dbprocess for a session is killed (usually because of memory problems), the DAM does not let you issue any further commands in that session except *Logoff* or *Quit*
- When you logoff a session, the DAM resets transaction mode to automatic and database version to the default value and clears the *sys(133)-sys(135)* and *sys(138)* values as well.
- If you issue a *Logon to host* command in a session that is already logged on, the DAM will immediately log off the first and establish a new logon. Since this does a full logoff, the settings for transaction mode and database version and the system flags end up with their default settings in the new logon session; you should explicitly set up the transaction mode and database version as you need
- Short number 0dp has at least nine digits of precision, which fits into the four-byte server type ‘int’, the *createnames()* mapping type; however, in some circumstances, it is possible to put more than nine digits of data into a short number, so you must be careful about what sort of numbers you pass into the database

- Using the /N qualifier with an OMNIS Boolean field causes the *createnames()* function to create a string that creates a NULL bit column, but this is illegal syntax for Sybase: avoid using /N with Boolean fields in *createnames()*
- When you reach the end of a select table (flag is set to false after the fetch), then *sys(133)* is set to the number of columns in the next select table
- *Describe database (tables)* does not list system tables
- *Describe database* (all varieties) does not report any errors if there are no tables; instead, it returns an empty select table
- *Describe server table (Columns)* accepts only the name of a table; it does not support any owner, database or other qualifier in the parameter
- The third column of *Describe server table (Columns)* is set to NULL for all types except char, binary, nchar, varchar, varbinary, varnchar, text, and image; the fourth column is set to NULL for all types except the float, integer and money types; column 7 is always NULL
- All the *Describe server table* and *Describe database* commands and the transaction commands (*Set transaction mode*, *Set autocommit*, *Commit session*, and *Rollback session*) issue a *Reset session* command
- After issuing the command *Describe server table (Indexes)*, sys(133) reports 3 even if there are no actual rows to return, even if <SKIPEMPTYSETS> is on
- You should not use *Describe server table (Indexes)* in the generic transaction mode because the stored procedure creates a table
- In automatic transaction mode, the Sybase DAM issues no commands to provide transaction support, since Sybase's default behavior matches the behavior OMNIS defines as automatic transaction mode, and the DAM ignores the OMNIS *Commit session* and *Rollback session* commands; in the generic transaction mode, the DAM issues certain Transact-SQL statements to begin, commit, and rollback transactions on the server, so you should never begin, commit, or rollback any transactions in SQL or in stored procedures, since these cause the DAM and the server to get confused and to generate errors; server transaction mode is identical to automatic mode
- You can end up with a select table with no rows in it, or a select table with no rows or columns; because these empty sets can be a nuisance in multiple-table situations, the server specific keyword <SKIPEMPTYSETS> lets you skip over these
- Even after statements that produce no results, such as insert and update, the Sybase DAM cannot verify whether there are select tables until you issue at least one *Fetch* or *Build list* command. This is why *sys(138)* is 1 even after such a statement.

Platform Specific Issues

- The alias in the interfaces file or SQL.INI must correspond to the name you use in the *Set hostname* command

– Under Windows, you can bind character, binary, picture, and list fields to Sybase columns as long as they are less than or equal to 32,767 bytes, and you can fetch data less than or equal to 32,766 bytes; MacOS is limited only by memory

– Under MacOS, the control panel **Sybaseconfig/ocscfg** must be installed and configured properly for the DAM to work

Data Type Mapping

The following tables describe the data type mapping for OMNIS and Sybase.

OMNIS to Sybase

OMNIS Data Type	Sybase Data Type
CHARACTER	
Character/National 0	varchar(1)
Character/National 1 <= n <= 255	varchar(n)
Character/National > 255	text
DATE/TIME	
Short date (all subtypes)	datetime
Short time	datetime
Date time (#FDT)	datetime
NUMBER	
Short integer (0 to 255)	tinyint
Long integer	int
Short number 0dp	numeric(9,0)
Short number 2dp	numeric(9,2)
Number floating dp	float/double
Number 0..14dp	numeric(15,0..14)
OTHER	
Boolean	bit
Sequence	int
Picture	image
Binary	image
List	image
Row	image
Object	image
Item reference	image

Sybase to OMNIS

Sybase Data Type	Describe Results Type	OMNIS Data Type
CHARACTER		
char(n)	CHARACTER(n)	Character n
varchar(n)	CHARACTER(n)	Character n
nchar(n)	CHARACTER(n)	Character n
nvarchar(n)	CHARACTER(n)	Character n
text	CHARACTER	Character 10,000,000
DATE/TIME		
datetime	DATETIME	Date time (#FDT)
smalldatetime	DATETIME	Date time (#FDT)
NUMBER		
tinyint	TINYINT	Short integer (0 to 255)
smallint	SMALLINT	Short number 0dp
int	INT	Long integer
numeric(p,n)	NUMBER	Number (n)dp
decimal(p,n)	NUMBER	Number (n)dp
real	FLOAT	Number floating dp
float	FLOAT	Number floating dp
double	FLOAT	Number floating dp
money	NUMBER (4dp)	Number 4dp
smallmoney	NUMBER (4dp)	Number 4dp
OTHER		
bit	BOOLEAN	Boolean
binary(n)	BINARY	Binary
varbinary(n)	BINARY	Binary
image	PICTURE	Picture

Informix

This section contains the additional information you need to access an Informix database, including server-specific programming, troubleshooting, and data type mapping to and from Informix. For general information about logging on to Informix and managing your database using the SQL Browser, refer to the earlier parts of this manual.

Server Information

The *server()* function takes a parameter in which you request information from the DAM about the server. You can use the *Calculate* command to place the result in an OMNIS variable:

```
Calculate RESULT as server('Version') ;; Returns the version number
   of the active DAM
Calculate PATH as server('Path') ;; Directory path of the DAM
Calculate API as server('vendorAPI') ;; Directory path of server API
   if available
```

Every DAM handles the following set of *server()* parameters.

– **Version**
the version string, same as sys(130)

– **Vendorapi**
the version string of the client API with which the DAM compiled

– **Path**
the file path to the DAM

– **File**
the name of the DAM file

– **DAM**
the name of the DAM

In addition, the Informix DAM supports the following server() parameters.

– **SQLERRD0 to SQLERRD5**
returns the corresponding member of the SQLERRD structure in the SQLCA

– **SQLWARN0 to SQLWARN7**
returns the corresponding member of the SQLWARN structure in the SQLCA

Stored Procedures

You can run stored procedures in INFORMIX with the EXECUTE METHOD statement.

Troubleshooting

The following points may help in resolving issues in programming OMNIS applications that use an Informix database.

– INFORMIX I-Net PC has a problem with FTP TCP/IP Version 2.3. Version 2.3 appears to work, but during the logon sequence, a dialog box reading “SYSTEM ERROR: CANNOT WRITE TO DEVICE AUX” appears. Clicking on the Cancel button removes the dialog box and communications will continue normally. The dialog box results from FTP having Windows debug calls in the file WINSOCK.DLL. You can avoid this problem using the WINSOCK.DLL that came with FTP version 2.2 in place of the file that came with the FTP Version 2.3 software.

– INFORMIX closes all cursors when you commit or roll back a transaction. It is recommended that you do not do long transactions as they may have serious effects on the INFORMIX database. To avoid these transactions, you should use the Server transaction mode.

– INFORMIX does not support the Set SQL blob preferences command.

Data Type Mapping

The following tables describe the data type mapping for OMNIS and Informix.

OMNIS to INFORMIX

OMNIS Data Type	INFORMIX Data Type
CHARACTER	
Character/National < 32767	CHAR
Character/National => 32767	TEXT
DATE/TIME	
Short date (all subtypes)	DATE
Short time	DATETIME HOUR TO MINUTE
Date time (#FDT)	DATETIME YEAR TO FRACTION
NUMBER	
Short integer (0-255)	SMALLINT
Long integer	INTEGER
Short number 0dp	DECIMAL(15,0)
Short number 2dp	DECIMAL(15,2)
Number floating dp	FLOAT
Number 0..14dp	DECIMAL(15,0..14)
OTHER	
Boolean	SMALLINT
Sequence	SERIAL
Picture	BYTE
Binary	BYTE
List	BYTE
Row	BYTE
Object	BYTE
Item reference	Not supported

Informix to OMNIS

INFORMIX Data Type	Describe Data Type	OMNIS Data Type
CHARACTER		
SQLCHAR	CHAR	National
SQLVCHAR	CHAR	National
DATE/TIME		
SQLDATE	DATE	Short date 1980-2079
SQLDTIME	DATETIME	Date time (#FDT)
NUMBER		
SQLSMINT	NUMBER	Number 0dp
SQLINT	NUMBER	Number 0dp
SQLINTERVAL	NUMBER	Number 0dp
SQLSERIAL	NUMBER	Number 0dp
SQLMONEY	NUMBER	Number floating dp
SQLDECIMAL	NUMBER	Number floating dp
SQLFLOAT	FLOAT	Number floating dp
SQLSMFLOAT	FLOAT	Number floating dp
SQLTEXT	CHAR	National
SQLBYTES	BINARY	Binary
SQLNULL	UNKNOWN	National

DB2

This section contains the additional information you need to access a DB2 Universal Server database, including server-specific programming, supporting DB2 extenders and data type mapping to and from DB2.

Server-specific Programming

Almost every DBMS has its own specific, extra features that are not part of the SQL standard. You can take advantage of many of these through *server-specific programming.* OMNIS provides special keywords and the *Server specific keyword* command to assist you in this. Server-specific keywords are single-word commands enclosed in angle braces, such as <SETBLOBSIZE>.

You use the *Server specific keyword* command to send a keyword command to your server.

```
Server specific keyword { < SETBLOBSIZE > 32768}
```

This command, after evaluating square bracket notation, sends the string to the DAM, which translates the command into the appropriate server instructions. The string often includes parameters.

The DB2 DAM supports the following keywords.

– **<SETBLOBSIZE> integer_value**
where 0 < integer_value < 2147483647. The createnames() function uses integer_value when columns of type BLOB are created. DB2 requires a minimum size; if SETBLOBSIZE has not been set, the default is 64K.

– **<USESCHEMANAMES> YES|NO**
If USESCHEMANAMES is set to NO, all of the OMNIS data dictionary querying commands will display information about any table, view or key the user has permission to see. If it is set to YES, only tables in the user's schema will be seen by the data dictionary commands.
In the case of the Describe table columns or keys commands, this behavior may be overloaded by qualifying the specified table name with a schema name.
e.g. **Describe server table (Columns) {Yourschema.MyTable}**
This type of overloading is not possible with the Describe Database (Tables) or (Views)_ commands, but the DB2 DAM will return two columns of data for these commands. The first contains the list of table or view names, the second the name of the schema in which the table or view occurs.
The default setting is NO.

– **<GET_DATASOURCES> list_name**
List_name is the name of a list defined as having two columns of type character(30). After execution of the command, the list will contain two columns of data, the first containing the names of all known data sources to the client machine, and the second holding a description.

– **<DATETIME_FORMAT> datetime_format_string**
where datetime_format_string is an OMNIS Date/Time format string.

– **<BIND_SESSION>**
Binds the current session's logon information with the DB2 externals to be used when the DB2 external commands log on to the DB2 database. This keyword overrides the information registered through the DB2 externals.

– **<UNBIND_SESSION>**
Unbinds the session's logon information being used by the DB2 externals The logon information contained in the externals is now used if registered.

Reserved Words

This section covers the DB2 specific reserved words.

The following schema names are reserved:

– SYSCAT

– SYSFUN

– SYSIBM

– SYSSTAT

In addition, it is strongly recommended that schema names never begin with the SYS prefix, as SYS is by convention used to indicate an area reserved by the system.

There are no words that are specifically reserved words in DB2. Keywords can be used as ordinary identifiers, except in a context where they could also be interpreted as SQL keywords. In such cases, the word must be specified as a delimited identifier. For example, COUNT cannot be used as a column name in a SELECT statement unless it is delimited.

IBM SQL and ISO/ANSI SQL92 include reserved words, these reserved words are not enforced by DB2 Universal Database, however it is recommended that they not be used as ordinary identifiers, since this reduces portability. Please see the final chapter in this manual which lists the SQL reserved words.

DB2 Extenders

DB2 extenders are extended or intelligent data types supported in IBM's DB2 Universal Database. They include *image*, *audio*, *video*, and rich *text* data types, although at present the OMNIS SQL Browser does not support the text type. The DB2 extenders take your data handling capabilities well beyond the standard character and numeric types, and allow you to create all types of application for storing, querying, and retrieving multimedia content for web-based and intranet solutions.

What makes these data types intelligent? Not only can you store image, audio, and video data in DB2, but you can search and retrieve this type of data by querying its content. For example, when querying image data you can specify a color or pattern in your query to retrieve images containing that color or pattern. You can do similar things with the audio and video extended data types.

The *Image extender* can store all types of image including GIF, JPEG, BMP, and TIFF. The image extender stores the file size, format, height, width, color information, and bit depth of the image. When you insert and retrieve image data to and from your DB2 database, you can convert the format of images and perform other operations such as compression and decompression, resampling, scaling, rotating, and color inversion. The image extender can also store a thumbnail or miniature version of an image suitable for display in a web browser.

The *Audio extender* supports a variety of audio file formats, including WAVE and MIDI files of up to 2 gigabytes in length. The audio extender stores attributes such as the number of audio channels, transfer time, and sampling rate, as well as the standard file information such as name, size, and modification date. Using the appropriate query you can retrieve an audio clip and play it in one operation. For example, in a music database you could retrieve tracks by a particular artist and playback audio clips of their work.

The *Video extender* supports a variety of video file formats, and can store video data up to 2 gigabytes in length. The video extender stores attributes such as frame rate, compression format, and number of video tracks. You can query your database for video clips based on the file format or modification date. Like the other DB2 extender types, you can retrieve a video clip and play it back using a single query. You can also use the video extender to automatically segment a video clip into shots based on scene changes or major differences between frames. The video extender stores information about the separate shots which you can query to produce a quick summary of a video, to display on a web page perhaps.

SQL Browser support for DB2 Extenders

OMNIS Studio supports connections to any DB2 database containing all standard data types, such as text and numeric. However several unique features have been added to the SQL Browser to support the DB2 extenders. Furthermore on Windows 95 and Windows NT platforms, a new set of external commands has been included to allow you to enable and disable the DB2 extenders: these are seamlessly integrated into the SQL Browser and allow you to enable DB2 extender data types using point-and-click context menus.

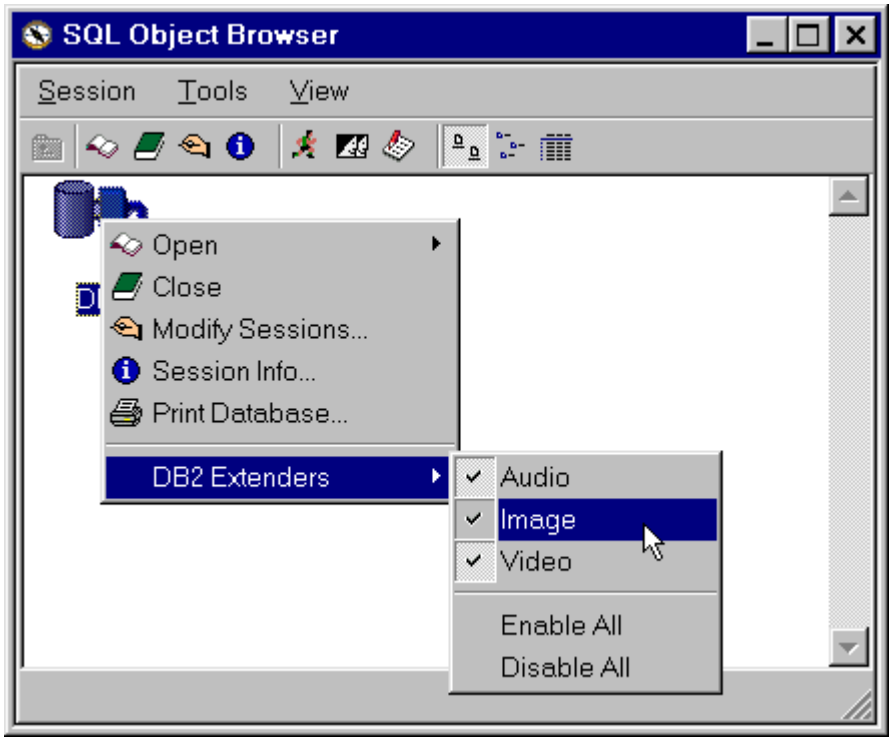
Enabling DB2 Extenders

Note that this feature is only available if you are working on a machine running Windows 95 or Windows NT.

To use the DB2 extenders you need to enable them in your database on several different levels: for your *database*, for a particular *table*, and for a specific *column* that will contain extended data. Furthermore you need to enable your database, tables, and columns for *each type of* DB2 extender, that is, you must enable your database for audio, video, and image data if a particular database table or column stores that type of data. IBM supplies tools to do this via the command line, but in OMNIS you can enable your database in the SQL Browser using point-and-click.

To enable your database for audio, video, or image data

- Start OMNIS and open the SQL Browser
- Logon to your DB2 database
- Right-click on your database to open its context menu



- Select DB2 Extenders>>Enable All

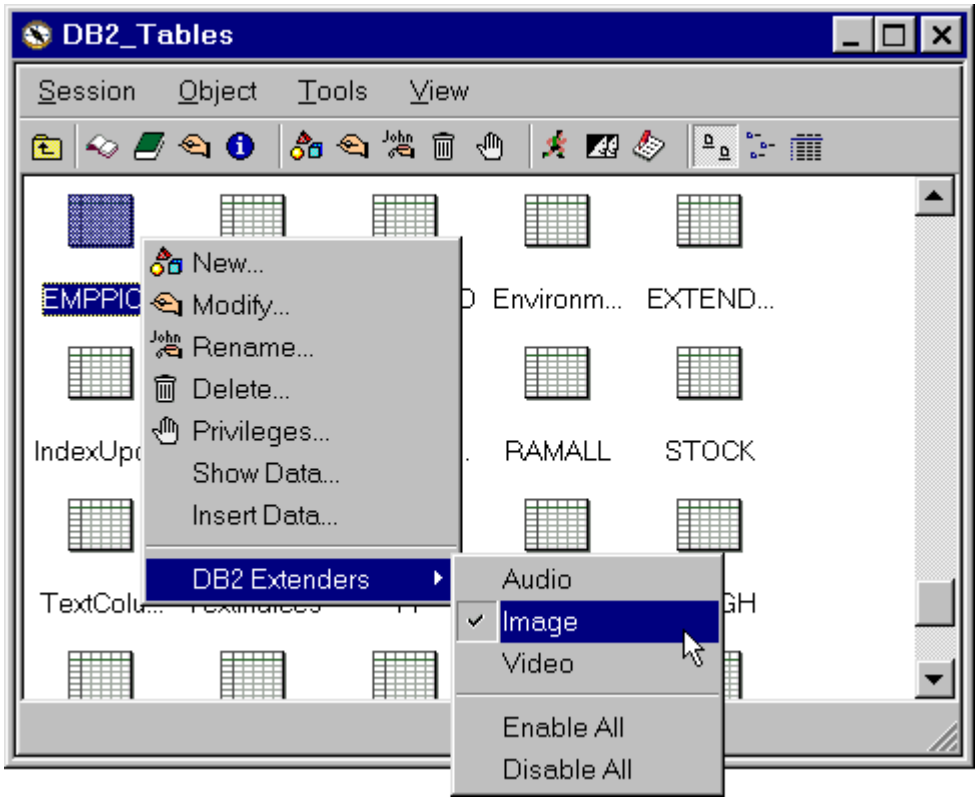
or to enable specific DB2 extenders

- Select DB2 Extenders>>Audio, Image, or Video

Next you need to enable specific tables in your DB2 database.

To enable a table for audio, video, or image data

- Double-click on your DB2 database in the SQL Browser
- Double-click on the Tables group
- Right-click on your table to open its context menu
- Select DB2 Extenders>>Enable All
- or to enable specific DB2 extenders
- Select DB2 Extenders>>Audio, Image, or Video



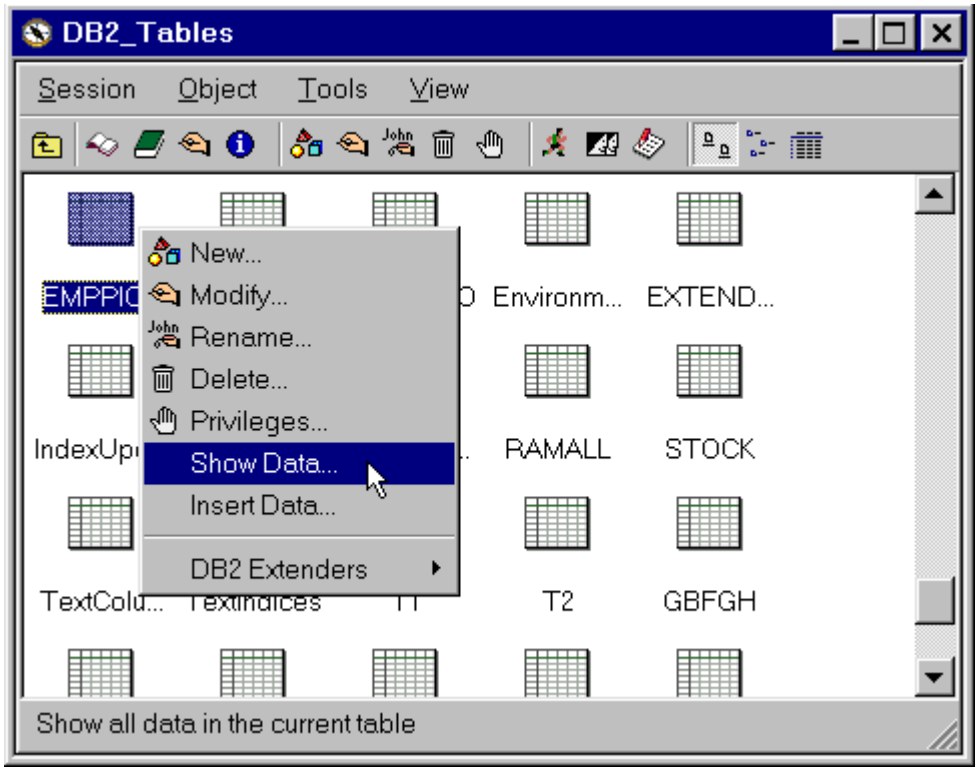
You can enable a column for audio, video, or image data when you create or alter the table containing a reference to the DB2 extender. In addition, the context menus for databases and columns includes an option to disable the DB2 extenders.

Playing Audio, Video, and Image Data

You can show or playback audio, video, and image data stored in your DB2 database using the Show Data option in the SQL Browser. At present you can display only one type of DB2 extender at a time, so for example, you cannot show audio and image data at the same time. The multimedia players in the SQL Browser display either audio, video, or image data depending on the data types in the current table.

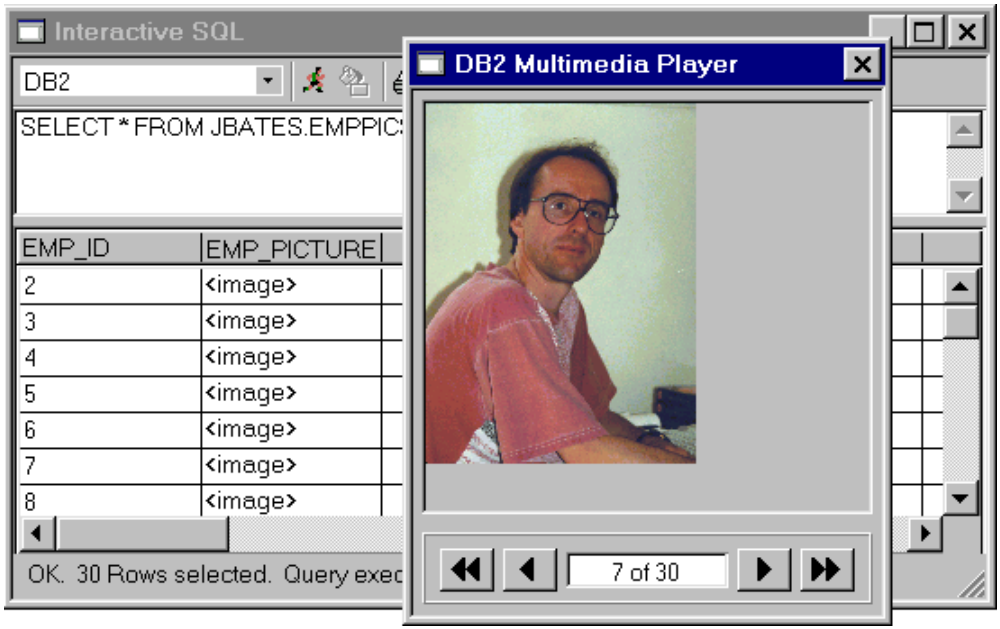
To show or playback audio, video, or image data

- Open your DB2 database in the SQL Browser and display its tables



- Right-click on the table and select Show Data

The SQL Browser creates a Select statement based on your table and sends it directly to the server via the Interactive SQL tool. The results are displayed in the lower pane of the ISQL tool. For example, if you show the data for a table containing image data, each image is shown in the ISQL tool as an <image> placeholder. You can click on the placeholder to display the image in the Multimedia Player.



The Multimedia Player lets you go to the next and previous row in the table, and you can jump to the beginning or end of the results set. The player is virtually the same for each type of DB2 extender, that is, for audio each sound clip in your database is displayed in the player, and likewise for video you can load each video and play it back.

Inserting Audio, Video, and Image Data

You can insert data into a table containing DB2 extenders using the Insert Data option in the SQL Browser. When you use the Insert Data option the SQL Browser creates a data entry window depending on the type of DB2 extender contained in the current table.

To insert audio, video, or image data into your DB2 database

- Open your DB2 database in the SQL Browser and display its tables
- Right-click on the table and select Insert Data

The data entry window contains a prompt for you to load the path to your audio, video, or image file.

DB2 Extender Wizards

The SQL browser provides many tools for examining your DB2 database and the extended data types such as audio, video, and image data. However you may want to add DB2 support to your application. The DB2 Extenders palette in the Component Store contains some easy-to-use wizards to build multimedia players and DB2 extender data entry windows.

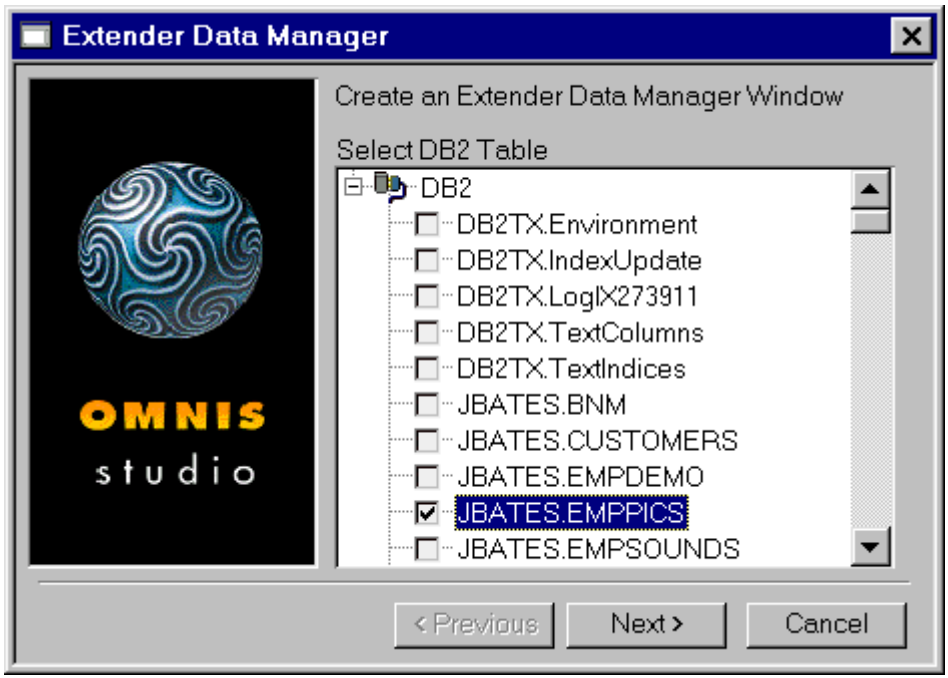
Extender Data Manager Wizard

The Extender Data Manager lets you insert any of the DB2 extender data types into your DB2 server database, including audio, video, and image data.

To create an Extender Data Manager

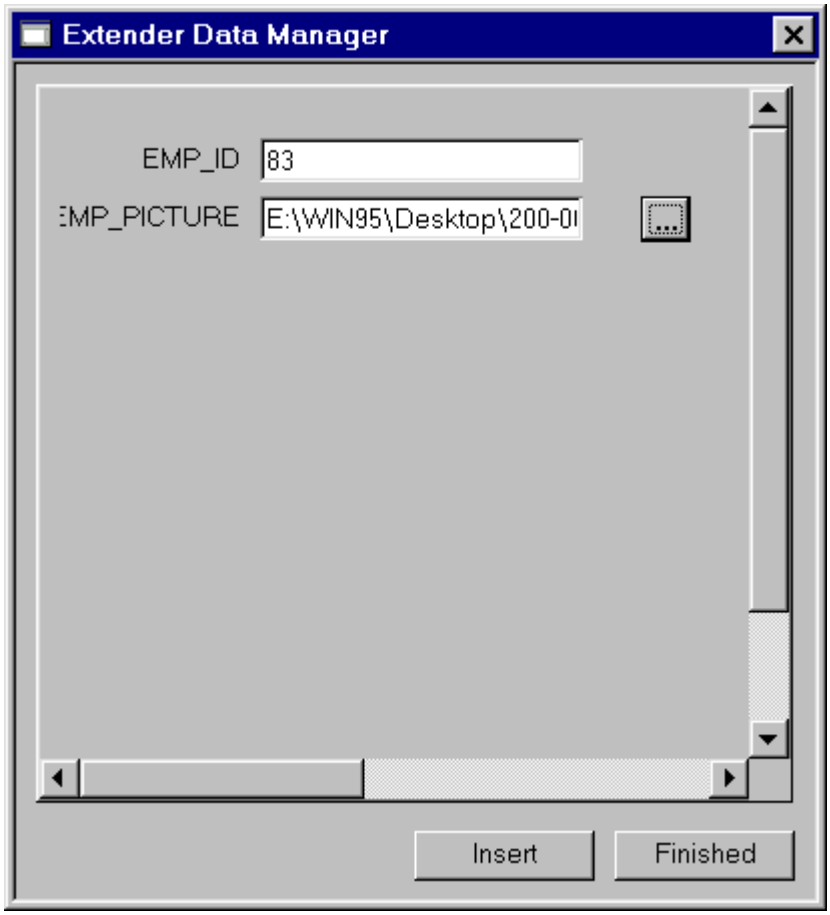
- Open your DB2 database in the SQL Browser
- Open your library and show the Component Store
- Scroll the Component Store toolbar and click on the DB2 Extenders button

- Drag the Extender Data Manager Wizard onto your library in the IDE Browser
- Name the new window and press Return, or click in the Browser



- Select the database table into which you want to insert data
- Click on Next and select the key field for the table
- Click on the Create button

The window includes a prompt or Browse button for the DB2 extender entry field that lets you add the path to your audio, video, or image file. For example, if your database table contains an image column, the Extender Data Manager looks something like the following.



Multimedia Wizard

The Multimedia Player lets you view any of the DB2 extender data types in your DB2 server database, including audio, video, and image data.

To create a Multimedia Player in your library

- Open your DB2 database in the SQL Browser
- Open your library and show the Component Store
- Scroll the Component Store toolbar and click on the DB2 Extenders button
- Drag the Multimedia Wizard onto your library in the IDE Browser
- Name the new window and press Return, or click in the Browser

- Select the database table you want to view
- Click on Next and select the key field for the table
- Click on the Create button

The Multimedia Player lets you go to the next and previous row in the table, and you can jump to the beginning or end of the data. The player is virtually the same for each type of DB2 extender, that is, for audio each sound clip in your database is displayed in the player, and likewise for video you can load each video and play it back.

DB2 Commands

The DB2 external commands let you enable and disable your DB2 database for Audio, Image, and Video extender data types. They are available in the external commands group in the method editor; their names are prefixed by “DB2” and they are described in the OMNIS Help. These commands let you control access to the DB2 extenders at a lower level than the enablement and disablement supported in the SQL Browser.

Data Type Mapping

The following tables describe the data type mapping for OMNIS and DB2.

OMNIS to DB2 UDB

OMNIS data type	**Server data type**
CHARACTER	
Character/National <= 4000	varchar
4000 < Character/National <= 32,700	long varchar
Character/National > 32,700	clob
DATE/TIME	
Short date (all subtypes)	date
Short time	time
Date time (#FDT)	datetime
NUMBER	
Short integer	smallint
Long integer	integer
Short number 0 dp	decimal (15,0)
Short number 2 dp	decimal (15,2)
Number 0..14 dp	decimal (15,0..14)
OTHER	
Boolean	varchar (3)
Sequence	integer
Picture	blob
Binary	blob
List	blob
Row	blob
Object	blob
Item reference	n/a

DB2 UDB to OMNIS

Server Data Type	Describe Data Type	OMNIS Data Type
NUMBER		
SMALLINT	INTEGER	Long integer
INTEGER	INTEGER	Long integer
FLOAT	FLOAT	Number floating dp
DECIMAL	NUMBER	Number dp
CHARACTER		
CHAR	CHAR	Character
VARCHAR	CHAR	Character
LONG VARCHAR	CHAR	Character
DATE/TIME		
DATE	DATE	Short date
TIME	DATE	Short time
TIMESTAMP	DATETIME	Date time (#FDT)
EXTENDERS		
IMAGE	BINARY	Binary
AUDIO	BINARY	Binary
VIDEO	BINARY	Binary
TEXT	BINARY	Binary

ODBC

This section contains the additional information you need to access a database using ODBC middleware, including server-specific programming, troubleshooting, and data type mapping. For general information about logging on via ODBC and managing your database using the SQL Browser, refer to the earlier parts of this manual.

Server-specific Programming

Almost every DBMS has its own specific, extra features that are not part of the SQL standard. You can take advantage of many of these through *server-specific programming*. OMNIS provides special keywords and the *Server specific keyword* command to assist you in this. Server-specific keywords are single-word commands enclosed in angle braces, such as <WRITEBLOB>.

You use the *Server specific keyword* command to send a keyword command to your server.

```
Server specific keyword { <WRITEBLOB> }
```

This command, after evaluating square bracket notation, sends the string to the DAM, which translates the command into the appropriate server instructions. The string often includes parameters.

The ODBC DAM supports the following server specific keywords:

– **<TYPEB_ON>** and **<TYPEB_OFF>**
these enable or disable multiple cursors

– **<RPC>**
executes remote procedure calls.

– **<RPCPROCS>**
returns a result set containing information on the parameters defined by an RPC.

– **<RPCSUPPORT>**
indicates whether remote procedure calls are supported or not.

The ODBC DAM turns the server Autocommit off when you log on to let OMNIS directly control transaction behavior. Consequently you may need to modify ODBC.INI for some ODBC data sources for various data isolation levels. When OMNIS Autocommit is off, any DDL operation (CREATE, ALTER, DROP) or GRANT and REVOKE causes the current transaction to commit immediately.

A DDL operation in a multiple-statement block causes an error.

A result set from a select operation may not be valid across transaction boundaries, depending on the driver. If you commit or roll back a transaction, you should execute the select again.

When Autocommit is on, OMNIS commits only groups of statements containing UPDATE, INSERT or DELETE statements, and then only if there is no pending result set.

Generally, turning Autocommit off results in much increased performance because the DAM does not force a commit after each statement. For best results, the ODBC driver should conform to the core SQL language. However, many drivers support core SQL with only minimal data types.

OMNIS maps Boolean values to the closest representation possible on the target server. If BIT is not available, it will generally map a Boolean to a numeric type. It represents TRUE and FALSE values by 1 and 0, respectively. When these values come into OMNIS target variables of type BOOLEAN, OMNIS converts them, otherwise it converts the values to appropriately sized NUMBER values. OMNIS currently expands numeric and Boolean deferred bind variables inline rather than deferring the binding.

When you map date and timestamp data types in the commands *Describe server table (columns)* and *Describe results*, the types map to OMNIS types Date1980 and DateTime1980, respectively. OMNIS maintains the correct date; the 1980 in the type name refers to setting the default if you don't supply the full year.

You must specify literals in SQL statements with single quotes ('), not double quotes (").

You should always check the value of *sys(131)* and *sys(132)* after SQL operations even if the command reports that the flag is TRUE. Most ODBC operations report the status "SQL SUCCESS WITH INFO" with a TRUE flag, but there may still be informative messages that you can obtain through *sys(131)* and *sys(132).* For example, if you connect through Sybase, the connection can succeed with the information "database context changed to xxxx", your default database. In most cases, the additional information does not refer to problems, but you must be aware of possibly important messages.

Multiple cursors

To allow multiple select cursors with Microsoft ODBC Driver for SQLServer 6, there are two server-specific keywords: <TYPEB_ON> and <TYPEB_OFF>.

The Microsoft ODBC Driver for SQLServer 6 needs to be able to execute SQL while processing a select statement. To allow multiple cursors, you should use *Set transaction mode(Generic)*, open your cursors and, for each, issue the following command prior to filling the SQL buffer.

```
Server specific keyword {<TYPEB_ON>}
```

If you want to reuse a particular cursor, that is, use it for something other than select statements, or select statements that will be fully processed prior to further SQL operations, or you want to close the cursor altogether, you should issue the command:

```
Server specific keyword {<TYPEB_OFF>}
```

Remote Procedure Calls

<RPC>

The <RPC> keyword executes remote procedure calls. It relies upon the ODBC driver used, being able to execute ODBC SQLProcedureColumns calls (see conformance issues in your driver documentation).

All output parameters including 'return values' should be defined OMNIS variables or fields. Input parameters may either be literals or OMNIS variables or fields. When using OMNIS fields, as either input or output parameters, they must be preceded by a colon ":" character. When using literals no special characters are required. In some cases you may wish to force a parameter to be an output parameter. You can do this by placing ':OUT' after the parameter.

The following example calls an RPC called 'name' and passes two parameters. The ODBC DAM will determine parameter types by the use of the 'SQLProcedureColumns' call. To obtain a list of parameters for a given RPC, use '<RPCPROCS> name'.

```
<RPC> RETURN = name ( :INPARAM, :OUTPARAM )
```

The following example calls the same RPC, but passes a literal as the first parameter and forces the last parameter to be treated as an output. In this case SQLProcedureColumns is used to determine all parameter types apart from the last.

```
<RPC> RETURN = name ( 'abc', :OUTPARAM:OUT )
```

In the above examples 'RETURN' is a defined OMNIS variable. It does not require a ":" character since in this case a literal would never be used.

After executing an <RPC> call, all output parameter fields will contain values set by the RPC, if successful. If a problem occurs the sys(131) and sys(130) functions are called. If the RPC builds a result set, it can be retrieved using the *Build list from select table* command.

<RPCPROCS>

<RPCPROCS> can take the following parameters, or none at all.

d:	database name
o:	owner name
p:	stored procedure, or RPC, name

The following example returns a result set containing information on all RPCs available to the currently logged on user.

```
<RPCPROCS>
```

The following is the same, but in addition it returns all RPCs owned by user 'colin'.

```
<RPCPROCS> o:colin
```

The following returns a result set containing information on all the parameters defined by the RPC called 'name'.

```
<RPCPROCS> p:name
```

The following returns a result set containing information on all the parameters defined by the RPC called ‘name’ and owned by user ‘colin’ on database ‘dbo’. This is used primarily when RPCs with the same name have been defined by two different users.

```
<RPCPROCS> d:dbo o:colin p:name
```

<RPCSUPPORT>

<RPCSUPPORT> takes one parameter, and returns a simple ‘Y’ if RPCs are supported in the current ODBC driver. For example, where SUPPORT is an OMNIS character string

```
<RPCSUPPORT> SUPPORT
```

Server Information

The *server()* function takes a parameter in which you request information from the DAM about the server. You can use the *Calculate* command to place the result in an OMNIS variable:

```
Calculate RESULT as server('Version') ;; Returns the version number
   of the active DAM

Calculate PATH as server('Path') ;; Directory path of the DAM

Calculate API as server('vendorAPI') ;; Directory path of server API
   if available
```

Every DAM handles the following set of *server()* parameters:

– **Version**
the version string, same as sys(130)

– **Vendorapi**
the version string of the client API with which the DAM compiled

– **Path**
the file path to the DAM

– **File**
the name of the DAM file

– **DAM**
the name of the DAM

In addition, the ODBC DAM supports the following server() parameters.

– **DBMSNAME**
returns the name of the server DBMS

– **DBMSVERSION**
returns the version of the server DBMS

– **DRIVERNAME**
returns the name of the ODBC driver

– **DRIVERVERSION**
returns the version of ODBC.DLL and the ODBC driver

Troubleshooting

The following points may help in resolving issues in programming OMNIS applications that use the ODBC middleware.

– Bind variables with ODBC v2 Drivers work with MS Access, SQL Anywhere, and other drivers.

– Implicit datatype conversions generally result in error 257: “Implicit conversion from datatype "x" to "y" is not allowed.” Use the CONVERT function to run this query. This error will occur, for example, when a select statement mixes data types between the SQL data and a bind parameter.

```
SELECT ... WHERE F = @[INT]
; "F" and "INT" are different datatypes
```

– Implicit conversions involving a datetime field will result in error 260: “Operand type clash: "x" is incompatible with datetime.”

– Whether or not the ODBC DAM can transfer data values larger than 32K bytes is dependent on the individual ODBC driver

– *Describe server table (Columns)* and *Describe server table (Indexes)* accept [owner].tablename as the parameter

– The *Describe results* and *Describe server table (columns)* commands generally report column size only for character and numeric or decimal date. In these cases, the commands report the actual display width of the field. For fields of other types, such as "INTEGER", the commands report the internal storage size

– You cannot issue DDL statements within a block with several SQL statements

– ODBC does not support any extended ORACLE cursor operations

Data Type Mapping

The following table describes the data type mapping for OMNIS to ODBC connections.

OMNIS to ODBC

OMNIS Data Type	ODBC Data Type
CHARACTER	
Character/National	SQL_VARCHAR
DATE/TIME	
Short date (all subtypes)	SQL_DATE
Short time	SQL_TIME
Date time (#FDT)	SQL_TIMESTAMP
NUMBER	
Short integer (0 to 255)	SQL_TINYINT
Long integer	SQL_INTEGER
Short number 0dp	SQL_NUMERIC(9, 0)
Short number 2dp	SQL_NUMERIC(9,2)
Number floating dp	SQL_DOUBLE
Number 0..14dp	SQL_NUMERIC(15, 0..14)
OTHER	
Boolean	SQL_BIT
Sequence	SQL_INTEGER
Picture	SQL_LONGVARBINARY
Binary	SQL_LONGVARBINARY
List	SQL_LONGVARBINARY
Row	SQL_LONGVARBINARY
Object	SQL_LONGVARBINARY
Item reference	SQL_LONGVARBINARY

EDA

This section contains the additional information you need to use the EDA/SQL middleware. For general information about logging on to EDA and managing your database using the SQL Browser, refer to the earlier parts of this manual.

Server-specific Programming

Almost every DBMS has some special features you can take advantage of through *server-specific programming*. OMNIS provides special keywords and the *Server specific keyword* command to assist you in this. Server-specific keywords are single-word commands enclosed in angle braces, such as <WAIT>.

You use the *Server specific keyword* command to send a keyword command to your server.

```
Server specific keyword { <WAIT> }
```

This command, after evaluating square bracket notation, sends the string to the DAM, which translates the command into the appropriate server instructions. The string often includes parameters. The keywords available with EDA are used to call FOCUS procedures on an EDA/SQL Server, and are:

– **<RPC>**
  Executes a remote procedure call on the server

– **<LONGRPC>**
  Executes a remote procedure call on the server with parameters > 80 characters in length

– **<ACCEPT>**
  Retrieves error messages into a variable

– **<DATE_FORMAT>**
  Sets the date format.

– **<TIME_FORMAT>**
  Sets the time format.

– **<DATETIME_FORMAT>**
  Sets the date/time format.

These keywords are described in the following section.

EDA Keywords

<RPC>

The <RPC> keyword executes a remote method call on the server and has the syntax:

```
<RPC>procedure_name [[varname=]value[,[varname=]value]...]
```

<LONGRPC>

The <LONGRPC> keyword is similar to <RPC> but allows RPCs with parameters greater than 80 characters in length.

<ACCEPT>

The <ACCEPT> keyword has the syntax:

```
<ACCEPT>ListName
```

ListName refers to a list which must be defined as:

```
Define list { MSGTYPE, MSGORIGIN, MSGCODE, MSGTEXT }
```

where the columns in the list are as follows:

- **MSGTYPE (numeric)** is the message type: message type 1 is a system error message, and message type 2 is a message sent from the remote method.
- **MSGORIGIN (character)** is the originator of the message. This string is 'EDA' for system error messages (type 1 above) or is otherwise specific to the remote method.
- **MSGCODE (numeric)** is the numeric code returned from the remote method.
- **MSGTEXT (Character)** is the message text returned from the remote method.

These variables are case-sensitive, but the order is not important. You can use $linemax (maximum number of lines in list) to control the number of messages returned in each call to <ACCEPT>. <ACCEPT> differs from *sys(131)* and *sys(132)* in that it returns all messages from the server, whereas the sys*()* functions return only the first system error message.

The following example illustrates the use of the EDA keywords.

```
Server specific keyword { <RPC>TABLIST }
.. check sys(131) for errors
Redraw WindowName

Server specific keyword { <WAIT> }
; Wait for the server to finish RPC
.. check sys(131) for errors
Redraw WindowName

; Bring the results back from the server. This call must be made
   every time an RPC is called.
If server("EXECUTING")
  Redraw WindowName
  Quit method
End If
```

```
; Set up message list and retrieve all waiting messages into it.
Set current list MSGLIST
Define list {MSGTYPE,MSGORIGIN,MSGCODE,MSGTEXT}
Server specific keyword { <ACCEPT>MSGLIST }
.. check sys(131) for errors
; Fetch all the rows in the result set into another list.
Set current list RESULT_LIST
Define list {NAME,FILLER1,CREATOR,FILLER2}
Build list from select table
.. check for errors
Redraw WindowName
```

<DATE_FORMAT>

The <DATE_FORMAT> keyword has the syntax:

```
<DATE_FORMAT> date_format_string
```

where date_format_string is an OMNIS Date format string

<TIME_FORMAT>

The <TIME_FORMAT> keyword has the syntax:

```
<TIME_FORMAT> time_format_string
```

where time_format_string is an OMNIS Time format string

<DATETIME_FORMAT>

The <DATETIME_FORMAT> keyword has the syntax:

```
<DATETIME_FORMAT> datetime_format_string
```

where datetime_format_string is an OMNIS Date/Time format string

Troubleshooting

The following points may help in resolving issues in programming OMNIS applications that use the EDA/SQL middleware.

- Under MacOS, if the **EDALINK.CFG** file does not exist in the **Preferences** folder inside the **System Folder** or if another application has this file open, sys(131) returns **Error -7, Fatal error: Error connecting to server**
- SQL statements that you pass to the EDA DAM must conform to the dialect of SQL that EDA/SQL supports; the DAM will not pass through native SQL to the target databases
- Describe server table (Indexes) and Set batch size do nothing when using EDA
- sys*(131)* returns the most recent EDA error code.

Data Type Mapping

There is no EDA/SQL-to-OMNIS data type mapping, since you cannot create tables using EDA/SQL. The following table describes the OMNIS to EDA/SQL mapping.

OMNIS Data Type	Server Data Type
CHARACTER	
Character/National <= 255	CHAR(n)
Character/National > 255	CHAR(255)[trunc]
DATE/TIME	
Short date (all subtypes)	CHAR(30)
Short time	CHAR(30)
Date time (#FDT)	CHAR(30)
NUMBER	
Short integer (0 to 255)	INTEGER
Long integer	Unknown
Short number 0dp	INTEGER
Short number 2dp	DOUBLE PRECISION
Number floating dp	DOUBLE PRECISION
Number 0dp	INTEGER
Number 2..14dp	DOUBLE PRECISION
OTHER	
Boolean	CHAR(3)
Sequence	INTEGER
Picture	CHAR(255)[trunc]
Binary	CHAR(255)[trunc]
List	CHAR(255)[trunc]
Row	Not supported
Object	Not supported
Item reference	Not supported

[trunc] types do not report a warning to OMNIS when the truncation occurs. EDA/SQL INTEGER data is a 32-bit signed int on most platforms, while SMALLINT is a 16-bit signed integer and DOUBLE PRECISION is a 64-bit float.

Chapter 16—SQL Reserved Words

This chapter contains a list of SQL reserved words. The table shows whether the reserved word is reserved in the ANSI-1989 or ANSI-1992 standards and whether it is reserved in any of the supported direct DBMS versions of SQL, as well as in OMNIS SQL.

For maximum portability of your application between data managers, you should not use any of the reserved words in this list as file class or variable names.

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
ABS				X			
ABSOLUTE		X				X	
ACCESS		X			X		
ACOS				X			
ACQUIRE						X	
ACTION		X				X	
ADD		X		X	X	X	X
ALL	X	X	X	X	X	X	X
ALLOCATE		X				X	
ALTER		X		X	X	X	X
AND	X	X	X	X	X	X	X
ANY	X	X	X		X	X	X
ARE		X				X	
ARITH_OVERFLOW							X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
AS	X	X	X	X	X	X	X
ASC	X	X	X	X	X	X	X
ASCII				X			
ASIN				X			
ASSERTION		X				X	
AT		X				X	X
ATAN				X			
ATAN2				X			
AUDIT					X	X	
AUTHORIZATION	X	X	X			X	X
AVG	X	X	X	X		X	X
BEGIN	X	X	X				X
BETWEEN	X	X	X	X	X	X	X
BIT		X		X			
BIT_LENGTH		X					
BOTH		X					
BREAK							X
BROWSE							X
BUFFERPOOL						X	
BULK							X
BY	X	X	X	X	X	X	X
CALL						X	
CAPTURE						X	

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
CASCADE		X					X
CASCADED		X					
CASE		X		X		X	
CAST		X				X	
CATALOG		X					
CCSID						X	
CHAR	X	X	X	X	X	X	
CHAR_CONVERT							X
CHAR_LENGTH		X					
CHARACTER	X	X	X	X		X	
CHARACTER_LENGTH		X					
CHARINDEX				X			
CHECK	X	X	X		X	X	X
CHECKPOINT							X
CHILD				X			
CHR				X			
CLOSE	X	X	X				X
CLUSTER		X			X	X	
CLUSTERED							X
COALESCE		X					
COBOL	X	X	X				
COLLATE		X					

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
COLLATION		X					
COLLECTION						X	
COLUMN		X			X	X	
COMMENT					X	X	
COMMIT	X	X	X			X	X
COMPRESS		X			X		
COMPUTE							X
CONCAT						X	
CONFIRM							X
CONNECT		X			X	X	
CONNECTION		X				X	
CONNECTIONS				X			
CONSTRAINT		X				X	X
CONSTRAINTS		X					
CONTINUE	X	X	X				X
CONTROLROW							X
CONVERT		X					X
CORRESPONDING		X					
COS				X			
COUNT	X	X	X	X		X	X
CREATE	X	X	X	X	X	X	X
CROSS		X				X	
CURRENT	X	X	X	X	X	X	X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
CURRENT_DATE		X				X	
CURRENT_SERVER						X	
CURRENT_TIME		X				X	
CURRENT_TIMESTAMP		X				X	
CURRENT_TIMEZONE						X	
CURRENT_USER		X				X	
CURSOR	X	X	X			X	X
DAT				X			
DATA-PGS							X
DATABASE						X	X
DATE		X		X	X	X	
DAY		X				X	
DAYS						X	
DBA						X	
DBSPACE						X	
DBCC							X
DEALLOCATE		X					X
DEC	X	X	X				
DECIMAL	X	X	X	X	X		
DECLARE	X	X	X				X
DEFAULT	X	X		X	X	X	X
DEFERRABLE		X					

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
DEFERRED		X					
DELETE	X	X	X	X	X	X	X
DESC	X	X	X	X	X	X	X
DESCRIBE		X					
DESCRIPTOR		X				X	
DIAGNOSTICS		X					
DIM				X			
DISCONNECT		X					
DISK							X
DISTINCT	X	X	X	X	X	X	X
DOMAIN		X					
DOUBLE	X	X	X			X	X
DROP		X		X	X	X	X
DTCY				X			
DTD				X			
DTM				X			
DTW				X			
DTY				X			
DUMMY							X
DUMP							X
EDITPROC						X	
ELSE		X			X	X	X
END	X	X	X				X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
END-EXEC		X				X	
ENDTRAN							X
ERASE						X	
ERRLEVEL							X
ERROREXIT							X
ESCAPE	X	X	X			X	X
EXCEPT		X				X	X
EXCEPTION		X				X	
EXCLUSIVE					X	X	
EXEC	X	X	X				X
EXECUTE		X				X	X
EXISTS	X	X	X		X	X	X
EXIT							X
EXP				X			
EXPLAIN						X	
EXTERNAL		X				X	X
EXTRACT		X					
FALSE		X				X	
FETCH	X	X	X			X	X
FIELDPROC						X	
FILE					X		
FILLFACTOR							X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
FIRST		X					
FLOAT	X	X	X		X		
FLOAT_TYPE				X			
FOR	X	X	X		X	X	X
FOREIGN	X	X				X	X
FORTRAN	X	X	X				
FOUND	X	X	X				
FROM	X	X	X	X	X	X	X
FULL		X				X	
GET		X					
GLOBAL		X					
GO	X	X	X			X	
GOTO	X	X	X			X	X
GRANT	X	X			X	X	X
GRAPHIC						X	
GROUP	X	X	X	X	X	X	X
HAVING	X	X	X	X	X	X	X
HOLDLOCK							X
HOUR		X				X	
HOURS						X	
IDENTIFIED					X	X	
IDENTITY		X					X
IDENTITY_INSERT							X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
IF							X
IMMEDIATE		X			X	X	
IN	X	X	X	X	X	X	X
INCREMENT					X		
INDEX				X	X	X	X
INDICATOR	X	X	X			X	
INITCAP				X			
INITIAL					X		
INITIALLY		X					
INNER		X				X	
INOUT						X	
INPUT		X					
INSENSITIVE		X					
INSERT	X	X	X	X	X	X	X
INT	X	X	X				
INTEGER	X	X	X	X	X		
INTERSECT		X			X	X	X
INTERVAL		X					
INTO	X	X	X	X	X	X	X
IS	X	X	X		X	X	X
ISOLATION		X				X	X
JOIN		X				X	
KEY	X	X				X	X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
KILL							X
LABEL						X	
LANGUAGE	X	X	X				
LEADING		X					
LEFT		X				X	
LENGTH				X			
LEVEL		X			X		X
LIKE	X	X	X	X	X	X	X
LINENO							X
LIST				X			
LOAD							X
LOCAL		X					
LOCK					X	X	
LOCKSIIZE						X	
LOG				X			
LOG10				X			
LONG				X	X	X	
LOWER		X		X			
MATCH		X					
MAX	X	X	X	X		X	X
MAXEXTENTS					X		
MICROSECOND						X	

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
MICROSECONDS						X	
MIN	X	X	X	X		X	X
MINUS					X		
MINUTE		X				X	
MINUTES						X	
MIRROR							X
MIREXIT							X
MOD				X			
MODE					X	X	
MODIFY		X			X		
MODULE	X	X	X				
MONTH		X				X	
MONTHS						X	
NAMED						X	
NAMES		X					
NATIONAL		X		X			X
NATURAL		X					
NCHAR		X		X			
NEW							
NEXT		X					
NHEADER						X	
NO		X					

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
NOAUDIT					X		
NOCOMPRESS					X		
NOHOLDLOCK							X
NONCLUSTERED							X
NOT	X	X	X	X	X	X	X
NOWAIT					X		
NULL	X	X	X	X	X	X	X
NULLIF		X					
NUMBER				X	X		
NUMERIC	X	X	X	X			
NUMERIC_TRUNCATION							X
NUMPARTS						X	
OBID						X	
OCTET_LENGTH		X					
OF	X	X	X	X	X	X	X
OFF							X
OFFLINE					X		
OFFSETS							X
ON	X	X	X	X	X	X	X
ONCE							X
ONLINE					X		
ONLY		X				X	X
OPEN	X	X	X				X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
OPTIMIZE						X	
OPTION	X	X	X		X	X	X
OR	X	X	X	X	X	X	X
ORDER	X	X	X	X	X	X	X
OUT						X	
OUTER		X				X	
OUTPUT		X					
OVER							X
OVERLAPS		X					
PACKAGE						X	
PAD		X					
PAGE						X	
PAGES						X	
PARENT				X			
PART						X	
PARTIAL		X					
PASCAL	X	X	X				
PCTFREE					X	X	
PCTINDEX						X	
PERM							X
PERMANENT							X
PICTURE				X			
PLAN						X	X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
PLI	X	X	X				
POSITION		X					
POWER				X			
PRECISION	X	X	X			X	X
PREPARE		X					X
PRESERVE		X					
PRIMARY	X	X	X			X	X
PRINT							X
PRIOR		X			X		
PRIVATE						X	
PRIVILEGES	X	X	X		X	X	X
PROC							X
PROCEDURE	X	X	X			X	X
PROCESSEXIT							X
PROGRAM						X	
PUBLIC	X	X	X		X	X	X
RAISERROR							X
RAW					X		
READ		X					X
READTEXT							X
REAL	X	X	X	X			
RECONFIGURE							X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
REFERENCES	X	X				X	X
RELATIVE		X					
RELEASE						X	
RENAME					X		
REPLACE							X
RESERVED_PAGES							X
RESET						X	
RESOURCE					X	X	
RESTRICT		X					
RETURN							X
REVOKE		X			X	X	X
RIGHT		X				X	
ROLE							X
ROLLBACK	X	X	X			X	X
ROUND				X			
ROW					X	X	
ROWCNT							X
ROWCOUNT							X
ROWID					X		
ROWLABEL					X		
ROWNUM					X		
ROWS		X			X	X	X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
RRN						X	
RULE							X
RUN						X	
SAVE							X
SCHEDULE						X	
SCHEMA	X	X	X			X	X
SCROLL		X					
SECOND		X				X	
SECONDS						X	
SECQTY						X	
SECTION	X	X	X				
SELECT	X	X	X	X	X	X	X
SENSITIVE				X			
SEQUENCE_TYPE				X			
SESSION		X			X		
SESSION_USER		X					
SET	X	X	X	X	X	X	X
SETUSER							X
SHARE					X	X	
SHARED							X
SHUTDOWN							X
SIMPLE						X	
SIN				X			

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
SIZE		X			X		
SMALLINT	X	X	X	X	X		
SOME	X	X	X			X	X
SPACE		X					
SQL	X	X	X				
SQLCODE	X	X	X				
SQLERRM		X					
SQLERROR	X	X	X				
SQLSTATE		X					
SQRT				X			
START		X			X		
STATISTICS						X	X
STOGROUP						X	
STOPOOL						X	
STRING				X			
STRIPE							X
SUBPAGES						X	
SUBSTR				X			
SUBSTRING		X		X		X	
SUCCESSFUL					X		
SUM	X	X	X	X		X	X
SYB_IDENTITY							X

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
SYB_RESTREE							X
SYNONYM					X	X	
SYSDATE					X		
SYSTEM		X					
SYSTEM_USER		X					
TABLE	X	X	X	X	X	X	X
TABLESPACE						X	
TAN				X			
TEMP							X
TEMPORARY		X					X
TEXTSIZE							X
THEN		X			X		
TIME		X		X			
TIMESTAMP		X		X			
TIMEZONE_HOUR		X					
TIMEZONE_MINUTE		X					
TINYINT				X			
TO	X	X	X		X	X	X
TRAILING		X					
TRAN							X
TRANSACTION		X				X	X
TRANSLATE		X					

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
TRANSLATION		X					
TRIGGER					X		X
TRIM		X				X	
TRUE		X					
TRUNCATE							X
TSEQUAL							X
UID					X		
UNION	X	X	X		X	X	X
UNIQUE	X	X	X	X	X	X	X
UNKNOWN		X					
UPDATE	X	X	X	X	X	X	X
UPPER		X		X			
USAGE		X					
USED_PGS							X
USER	X	X	X		X	X	X
USER_OPTION							X
USING		X				X	X
VALIDATE					X		
VALIDPROC						X	
VALUE		X					
VALUES	X	X	X	X	X	X	X
VARBINARY				X			
VARCHAR		X		X	X		

Reserved Word	ANSI-1989	ANSI-1992	INFOR-MIX	OMNIS	ORA-CLE	IBM SQL	SYB-ASE
VARCHAR2					X		
VARIABLE						X	
VARYING		X		X			X
VCAT						X	
VIEW	X	X	X		X	X	X
VOLUMES						X	
WAITFOR							X
WHEN		X					
WHENEVER	X	X	X		X		
WHERE	X	X	X	X	X	X	X
WHILE							X
WITH	X	X	X	X	X	X	X
WORK	X	X	X			X	X
WRITE		X					
WRITETEXT							X
YEAR		X				X	
YEARS						X	
ZONE		X					

Index

OMNIS

OMNIS Programming

Version 2

How to use this manual

The on-line documentation is designed to make the task of identifying and accessing information about OMNIS Studio as easy and intuitive as possible.

You can navigate this document, or find topics, in a number of different ways.

Bookmarks

Bookmarks mark each topic in a document. To view the bookmarks in this document, click on the Bookmark icon on the Acrobat toolbar or select the **View>>Bookmarks and Page** menu item.

Click on an arrow icon ▷ to open or close a topic, and click on a topic name or double-click a page icon 🗋 to move directly to a topic.

Thumbnails

Thumbnails are small images of each page in the document. To view the Thumbnails in this document click on the Thumbnails button on the Acrobat toolbar, or select the **View>>Thumbnails and Page** menu item.

You can click on a thumbnail to jump to that page. Also you can adjust the view of the current page by moving and/or sizing the gray page-view box shown on the current thumbnail.

Links

Links in this document connect related information or take you to a specific location in the document. Links are indicated with *blue italic* text. To jump to a related topic, move the pointer over a linked area (the pointer changes to a pointing finger) and simply click your mouse. Try it!

To return to your last view or location, click on the **Go back** button on the Acrobat toolbar.

Browsing

You can use the Browse buttons on the Acrobat toolbar to move back and forth through the document on a page by page basis. You can also click on the **Go Back** to return to your last view or location.

Find

You can find a text string using the **Tools>>Find** menu item. To find the next occurrence of the text you can use the **Tools>>Find Again** option. If you reach the end of the document, you can use the Ctrl-Home key to go to the beginning and continue your find.

Search

If you have the Acrobat Search plug-in (available under the **Tools>>Search** menu in some versions of Acrobat Exchange and Reader), you can use the Studio Index to perform full-text searches of the entire OMNIS Studio on-line documentation set. Searching the Studio Index is much faster than using the **Find** command, which reads every word on every page in the current document only.

To Search the Studio Index, select **Tools>>Search>>Indexes** to locate the Studio Index (Studio.pdx) on the OMNIS CD. Next, select **Tools>>Search>>Query** to define your search text: you can use Word Stemming, Match Case, Sounds Like, wildcards, and so on (refer to the Acrobat Search.pdf file for details about specifying a query). In the Search Results window, double-click on a document name (the first one probably contains the most references). Acrobat opens the document and highlights the text. To go to the next or previous occurrence of the text, use the Search Next or Search Previous button on the Acrobat toolbar.

Grabbing Text from the Screen

You can cut and paste text from this document into the clipboard using the Text tool. For example, you could copy a code segment and paste it into the OMNIS method editor.

Getting Help

For more information about using Acrobat Reader see the PDF documents installed with the Reader files, or select the **Help** menu on the main Reader menu bar.