FORWARD - Origins of the Clarion Language

Contents

by Bruce D. Barrington, CEO, TopSpeed Corporation

As so often happens, I was just trying to please myself. I bought the first PC I ever saw and wanted to program it. That's what I do. Pascal was a straight-jacket and C wasn't available yet. So I tried BASIC. All it needed were some smart screen and keyboard routines. Right? Perhaps a little indexed sequential. Right?

Wrong! I could make it work. But I couldn't make it clean. I had just spent 10 years working with software development tools of my own design. I liked them. Maybe it was time to share what I had learned. Maybe the world really needed yet another computer language—a general-purpose, business programming language. Designed especially for PCs.

It may sound contradictory to call a business language "general-purpose," but in the PC world there are many business "languages" that are anything but general-purpose. Writing spreadsheet macros is programming, I suppose, but the macros hardly comprise a general-purpose language. For that matter, most database languages are not general-purpose languages. They are really scripts to be executed by their database manager. The scripts define a role the database manager plays while acting out your application. Even the dBase language, which can be compiled and run on stand-alone basis, is not really general-purpose.

According to my definition, a general-purpose language should be able to exercise the entire repertoire of capability offered by the underlying platform. That means a program should be able to read any section of any file that is visible to the operating system. It should pass through all the versatility available for the user interface. It should connect, in standard ways, to other general-purpose languages and componentware. A general-purpose language does not contaminate a program with its own "look and feel." It does not erect barriers to be surmounted. Rather, it grants wide latitude within the constraints of its platform to solve a broad range of programming problems with an extensive choice of styles.

But why restrict the new language to PCs? Other mainstream languages are meticulously portable. I decided that PCs deserve special treatment. Even in 1984, when I began designing Clarion in earnest, PCs already comprised a substantial percentage of all the computers installed in the world. And PCs were different than other computers. They were inherently single-user devices with an integrated keyboard and monitor. The keyboard and monitor could be accessed instantly, without modems and communications lines.

These machines begged for responsive, interactive application programs. I wanted to exploit this functionality by building memory-mapped video into my new language. If a Clarion program could "only" run on 40 or 50 million computers, that was all right by me.

I was driven by the steadfast belief that programming should be simpler. That programming languages should be easier to read and write. And that the poor productivity associated with software development stemmed from inadequate and poorly designed programming tools.

These feelings began as pet peeves: Why would anyone design an **IF** statement like **IF**...**THEN BEGIN**;*statements*;**END ELSE**... (Pascal). What possible value do the **THEN**, **BEGIN**, and **END** keywords serve in this structure? Why use "**:=**" instead of "**=**" for an assignment statement (Pascal, Modula-2, Ada). Didn't the language designer know that assignments are the most numerous statements in a program or that "**:=**" is a finger locking combination of shifted and unshifted keys? How about a **READ**...**AT END** (COBOL) clause that sets an end-of-file variable that is tested to terminate a read loop? Why can't the loop test for end-of-file? Having declared a variable, why must I remind the compiler to convert it in mixed expressions? Can't the compiler remember that for me? Have you ever done lint collection? Did you ask why? And hex dumps. What about HEX DUMPS! After twenty years of programming, I felt like the anchorman in the movie *Network* who shouted out the window: "I'm mad as hell and I'm not going to take it anymore."

Setting the Style

So I set out to design a new computer language that was compact (easy to write) and expressive (easy to read). I began at the back and worked toward the front: First, I wrote lots of programs, experimenting with syntax and semantics until the programs looked great. Then I wrote a small language reference manual. When the manual was well along, the development team started writing a compiler. The language was changing daily. Our old development memos describe an energetic and interactive process. Many ideas were proposed and rejected for reasons of art. Others for poor technology. Some were simply insane. Like Darwin's species, only the strong survived.

I classify programming languages into three styles: token oriented, sentence oriented, and statement oriented. Token oriented languages like Pascal and C are compact but not particularly expressive. Such languages treat a program as a set of tokens (keywords, data names, constants, punctuation, etc.) separated by "white space" (spaces, CR/LFs, comments, and sometimes commas). The compiler collects the tokens and ignores the white space. Token oriented languages are one-dimensional, so programmers use white space to add a second dimension to their programs:

```
typedef struct {
  unsigned char  Type;   /*the type of structure*/
  unsigned       Vlen;   /*variable length*/
  unsigned char  Dplac;  /*decimal places if decimal*/
  void           *Use;   /*pointer to variable*/
}Usedef
```

This C programmer has done about everything possible to code a readable type definition. But the left brace seems to "dangle" off the **struct** keyword. And Usedef dangles off the right brace. After all, braces aren't very artistic vertical delimiters.

Sentence oriented languages like COBOL and most database languages are expressive but not very compact. Sometimes statements in sentence oriented languages read like perfect English. This COBOL statement is certainly expressive:

```
MULTIPLY PRINCIPAL BY RATE GIVING PAYMENT ROUNDED.
```

But no more so than:

```
Payment = Principal * Rate
```

I would argue that in the context of an entire program, the second statement is easier to read than the first, which tends to melt into paragraphs full of verbiage. Other sentence formats are not very English-like at all. I found this "beauty" in an xBase language reference manual:

```
EDIT  [FIELDS <field list>] [<scope>][FOR <expL1>]
      [WHILE <expL2>][FREEZE <field>]
      [KEY<expr1> [,<expr2>]] [LAST] [LEDIT] [REDIT]
      [LPARTITION] [NOAPPEND] [NOCLEAR] [NODELETE]
      [NOEDIT | NOMODIFY] [NOLINK] [NOMENU] [NOOPTIMIZE]
      [NORMAL][NOWAIT][PARTITION <expN1>][PREFERENCE <expC1>]
      [SAVE][TIMEOUT <expN2>] [TITLE <expC2>]
      [VALID [:F] <expL3> [ERROR <expC3>]] [WHEN <expL4>]
      [WIDTH <expN3>] [[WINDOW <window name1>]
      [IN [WINDOW] <window name2> | IN SCREEN]]
      [COLOR SCHEME <expN4>] | COLOR <color pair list>]
```

Wow! These are certainly English words, but are they expressive? Could any programmer understand an instance of this statement format without a manual? Among many other questions I'd like to ask is: Who designed a **WHILE** clause and a **WHEN** clause in the same statement? It makes me want to scream out the window.

My experimental programs had become statement oriented—that old fashioned style used by FORTRAN and BASIC. Statement oriented languages exploit the fact that source programs are contained in ASCII source files—every line of a program is a record in the file. So record boundaries can be used to eliminate punctuation. I settled on a statement format that proved to be compact, expressive, and versatile:

```
label STATEMENT[(parameters)] [,ATTRIBUTE[(parameters)]] ...
```

Attributes are only used to declare data. Executable statements use the format of a standard procedure call. Of course, I defined different statement formats for assignment statements (A = B) and (IF, CASE, etc.).

A statement label starts in column one (the first position of the record). A statement without a label must not start in column one. A statement is terminated by the end of the line unless it is continued by a vertical bar (|). I adopted the semi-colon as an optional statement separator to allow more than one statement per line. By adopting the Modula-2 concept of ignoring empty statements, I eliminated the distinction between statement separators and terminators that had confounded countless Pascal programmers.

This design eliminates the punctuation otherwise necessary to identify labels and separate statements. Blocks of statements are initiated by a single compound statement such as IF and are terminated by a statement separator such as ELSE (which initiates another statement block) or by an END statement (or period). There are no “dangling” keywords.

Declaring Data

In its infancy, COBOL was said to be “self-documenting” because of its explicit data division and its expressive statement syntax.. Every element that a COBOL program processes must be declared in the data division: variables, constants, files, records, indexes—even sort sequences and report formats. I agreed that these declarations were essential for documenting business programs. And I felt that our new statement format would greatly improve their readability.

In the late 1960’s, IBM promoted PL/I as the successor to COBOL. The language was a disappointment to many, but it did offer a few fresh ideas. By condensing the data type keywords and introducing embedded comments (/*comment*/), PL/I provided enough space to comment every declaration statement. COBOL had been designed for long, descriptive data names. But programmers didn’t use long data names. There were good reasons for this: First of all, programmers like to columnarize programs to make them more readable. Arranging the data division in columns restricts data names to an arbitrary maximum length. Secondly, programmers don’t like long data names in the procedure division. Long names create unwieldy expressions and add to the writer’s cramp produced by an already verbose language. So most COBOL programmers used short, cryptic labels and wrote programs that weren’t nearly as self-documenting as they should have been.

PL/I programmers got around that problem by commenting their declaration statements. If there was a question about the meaning of a data name, it could be resolved by looking up its declaration. I had managed a large PL/I project in the 60’s and became convinced that declaration statements required three parts: a statement label, a data type, and a comment.

The new statement format was perfect. The statement label appeared on the left where it would be most visible. Data type keywords were short (BYTE, REAL, DIM, etc.) to maximize the space available for the comment. As a final space saver, a single exclamation character (!) was designated as a comment initiator.

COBOL and PL/I use “levels” to declare data structures. Every variable has a level number. A variable with a higher level number is “part of” a prior variable with a lower level number. If a variable is not part of a data structure, it is declared as an “01” or “77” level. I never liked using “levels” and was surprised that they were carried over in PL/I. I considered them arbitrary and a waste of space. (What does “77” mean and why do unstructured variables need a level anyway?) I chose GROUP (named after COBOL’s “group item”) as a compound statement to initiate data structures (which we then called “groups”). This mechanism is similar to record...end used in Pascal, Modula-2, and ADA; and struct{..} used in C. Indenting nested GROUP statements produces a very readable declaration:

```
Error   GROUP,PRE(Err)        !Error information
Date      DATE                !Date of error
Time      TIME                !Time of error
Device    STRING(12)          !Active device
Message   GROUP               !Error message
MsgCode     STRING(@P###P)    !Message Code
            STRING(‘ - ‘)
MsgText     STRING(32)        !Message text
          END
        END
```

COBOL and PL/I permit the same data name to be used in different data structures. Such data names are referenced by the data name qualified by the structure name. This is a useful construct, since the same fields frequently appear in more than one data structure (e.g. ACCT-NO IN OLD-VENDOR, ACCT-NO IN CURRENT-PAYEE, etc.). But many programmers refuse to use this feature because it creates such long references. Instead, they code mnemonic prefixes on every field (e.g., VND-ACCT-NO). This takes extra coding time and reduces the available name space.

To deal with this issue, I included an optional prefix attribute that could be attached to any data structure (e.g. **PRE(VND)**). Elements of the structure are qualified by placing the prefix and a colon in front of their data name (e.g. VND:AcctNo, PAY:AcctNo).

To match the functionality of “MOVE CORRESPONDING” in COBOL and “BY NAME” assignments in PL/I, a “deep” assignment statement was added to move matching elements between groups:

```
DestinationGroup :=: SourceGroup
```

As a business language, Clarion needed a rich set of basic data types: All sizes of integers and real numbers were included to provide compatibility with external record layouts and parameter lists. Packed decimals were included to solve rounding problems and reduce memory usage. (They can be declared in a range of sizes.) Various string formats (fixed, Pascal, and C), along with a complete set of string functions, were also included. And finally, data types for dates and times were designed to support direct arithmetic on these variables:

```
Tomorrow = Today + 1
```

But what about structured data types? In ALGOL-like languages such as

Pascal, Modula-2, Ada, and C, groups and arrays are declared as types. You declare the type, then you declare a group or array as an instance of the predeclared type. I never have liked this syntax. In business programs, most groups and arrays are only declared once. Thinking up a type name and coding a **TYPE** statement is usually unnecessary busy-work. I have never considered a group or an array to be a data type anyway. Groups and arrays describe storage relationships, not data types.

So I made the type declaration optional. A Clarion declaration with a **TYPE** attribute declares a data type that can be used for recurring structures or structures that are passed as parameters. A declaration with no **TYPE** attribute declares both a data type and a variable of the same name. I adopted the PL/I **LIKE** statement to declare a variable of predeclared type. I felt that this design offered the best of both worlds:

```
Totals      GROUP,PRE(QTR)
GrossPay      DECIMAL(12,2)
Deductions    DECIMAL(12,2)
NetPay        DECIMAL(12,2)
            END

YTD:Totals  LIKE(Totals),PRE(YTD)
```

Painless Typing

A computer language is strongly typed if every data element has a single data type and the language syntax makes it is impossible to view that element as a different type. Many experts feel that strong typing increases program reliability. Perhaps. But strongly typed programs are harder to write, restricting the use of general purpose procedures, and requiring an unnecessarily vigilant awareness of data types. Furthermore, I have never heard a COBOL programmer accuse **REDEFINES** (used solely to defeat strong typing) of causing reliability problems. (COBOL programmers, by the way, are not uncritical of their language. The **ALTER** statement fell into disuse years ago because it produced unstable programs.)

I didn't want our new language to be strongly typed. First of all, I wanted to support re-declarations similar to **REDEFINES** or the **union type** in C. Redeclarations are useful for implementing record types (variant records in Pascal) and for handling special programming cases. I assigned the **OVER** attribute to this purpose:

```
MonthNames    STRING('JanFebMarAprMayJunJulAugSepOctNovDec')
Month         STRING(3),DIM(12),OVER(MonthNames)
```

Secondly, I wanted group structures to be treated like strings. This weakened data typing because groups can contain data types other than strings. But groups need functionality. They must be moved, passed as parameters, even (carefully) compared. That's the rub, of course. Most numeric data types don't collate as strings, so groups containing numeric elements usually won't collate properly. Negative integers collate higher than positive integers and floating-point numbers collate somewhat randomly. Design involves compromise (sigh) and I elected the functionality while accepting the risk.

It was important for Clarion data types to permit simple construction of general-purpose procedures. If a procedure expected a numeric parameter, then any numeric data type should suffice. I thought it was ridiculous to require different numeric functions to handle different numeric data types like the ALGOL derivative languages. To go even further, I think polymorphism, as implemented in C++, that requires separate functions for each data type but permits them to be called by a single function name is a notational sham.

In the original version of Clarion, parameters were not even prototyped. Whatever appeared in the callers argument list was used by the procedure. Clarion now requires parameter prototypes but permits the data type to be unspecified. Clarion procedures have always been truly polymorphic for unstructured data.

Clarion parameters are prototyped to be passed by value or by address. Clarion does not support pointers. There are two reasons for this: First, pointers don't carry data type information with them and can be easily misused. And second, pointer dereferences (syntax differentiating the pointer from its target) needlessly complicate programs. It has been my experience that pointer mishaps are involved in most C program bugs.

We chose reference variables, as implemented in C++, to support indirection. A reference variable contains the data type as well as the identity of its target. And a reference variable is automatically dereferenced when it is used. There is no possibility of confusion between a reference variable and its target. Consider the following:

```
CompanyA  FILE
            :
          END
CompanyB  FILE
            :
          END
Company   &FILE             !Company being processed
  CODE
  CASE CompanyLetter        !Which company to process?
  OF 'A'
    Company &= CompanyA     !Point to Company A
  OF 'B'
    Company &= CompanyB     !Point to Company B
  END
  OPEN(Company)             !Open selected company
```

The reference variable *Company* is set by a reference assignment statement (&=). The compiler will object if the data types don't match. Thereafter, a reference variable can be used in any context its target is permitted.

Another important issue involved automatic type conversion. I felt strongly that you declared a data type so that the compiler would know! And that an obliging compiler would generate data type conversions as needed. I also felt that a great compiler would probe expressions for meaning and supply logical conversions.

For example, if I add a string to an integer, it is reasonable for the compiler to assume that the string contains an ASCII number and to generate such a conversion. Conversely, if I concatenate an integer to a string, I am asking the compiler to convert the integer first. By selecting appropriate data types for intermediate values, the compiler can safely convert data types in expressions without losing information. If you divide two integers, a good compiler will store the result in an intermediate value that will hold a fraction. If you add an integer to a string, the compiler will also use a fractional intermediate value because a string is capable of expressing a fraction.

Information can be lost, of course, when a value is moved, for instance, by an assignment statement or as a parameter of a procedure call. Moving a real number to an integer truncates the fraction. Moving a real number to a packed decimal rounds to the least significant decimal digit. Some languages, such as Pascal, require that such data conversions be explicitly called. I felt that by declaring a data type, a programmer was requesting the compiler to implicitly restrict the data element to a given domain of value.

Earlier versions of Clarion used just two data types for numeric intermediate values: 32 bit signed integer (LONG) and a 64 bit floating point (REAL). A divide operation or any operation with one or more REAL operands would produce a REAL intermediate value. This strategy provided sufficient accuracy since a REAL could express the maximum numeric significance (15 digits) supported by Clarion. Although they are accurate, floating point values are not discreet. Two equivalent expressions such as 1 / 2 and 2 / 4 can produce floating point results that differ in the least significant bit. This is usually a meaningless difference in computations.

But not in comparisons. A programmer expects one-half to equal two-fourths. I may be willing to avoid comparing REALs but I expect a logical expression such as this to work every time:

```
IF Hours > Normal * 1.5
```

Using a REAL to receive the expression on the right casts doubt on the results of the comparison. We resolved this issue in Clarion for Windows by implementing fixed-point intermediate values with 31 decimal digits on each side of the decimal point. This change also increased our maximum numeric significance to 31 digits.

Control Structures

While the business languages, COBOL and PL/I, offered the preferred model for declaring data, the ALGOL derivatives, especially Modula-2, offered better control structures. I modified the Modula-2 **IF** statement by making the **THEN** keyword replaceable by a statement separator. This had the effect of eliminating superfluous **THEN**s from multi-line **IF** structures. By adopting Modula-2's **ELSIF,** I eliminated the massive indenting and multiple terminations caused by deeply nested **IF** structures:

```
IF Number < 0
  Sign = -1
ELSIF Number > 0
  Sign = +1
ELSE
  Sign = 0
END
```

I also used Modula-2 as a guide for Clarion's **CASE** statement. Modula-2's **CASE** supports enumerated case labels and case label ranges—very useful features. But I didn't like its punctuation. The **OF** keyword introduces the first case label, but subsequent case labels are initiated by a vertical bar ("|"). I felt this punctuation was ugly and not very intuitive. Instead, I used **OF** to introduce all case labels. I invented the **OROF** keyword to enumerate case labels and the **TO** keyword for case label ranges. These changes produced a very friendly syntax:

```
CASE SUB(Name,1,1)
OF('A') TO ('M') OROF('a') TO ('m')
  DO FirstHalf
OF('N') TO ('Z') OROF('n') TO ('z')
  DO SecondHalf
ELSE
  DO FirstHalf
END
```

Modula-2 was the first usage I had seen of the **LOOP** keyword in its proper context. In Modula-2, **LOOP...END** executes an unconditional loop that is terminated by executing an **EXIT** statement. I augmented this concept by adding a **CYCLE** statement to recycle the loop from within. (I also changed **EXIT** to **BREAK** because I was using **EXIT** for another purpose.) I implemented conditional loops by adding four optional clauses to the **LOOP** statement:

```
LOOP I = 1 TO 100 BY 2
LOOP 10000 TIMES
LOOP WHILE Count > 0
LOOP UNTIL EndOfFile
```

I felt that good program organization required local subroutines. A local subroutine is a block of statements that has been removed from the main logic and is executed by a subroutine call statement. If the subroutine is aptly named, the main logic becomes shorter without losing clarity. COBOL and BASIC use **PERFORM** and **GOSUB** for this purpose. Local procedures in Pascal and Modula-2 nearly fit the bill but they require a

prototype statement to declare the parameter types. I didn't want to support subroutine parameters because I wanted all the caller's data to be visible to the subroutine. I designed the **ROUTINE** statement to initiate a local subroutine. **ROUTINE**s are placed at the end of a procedure or function and are executed by a **DO** statement.

A number of languages support executing a single statement from a list of statements as indicated by a statement selection integer. FORTRAN uses the computed **GOTO**. COBOL uses **GOTO...DEPENDING ON**. BASIC uses **ON...GOTO** and **ON...GOSUB**. I wanted to implement a similar capability that would execute any type of statement from a list of statements depending on an integer expression. I named this structure **EXECUTE** after the common **XEQ** machine language instruction which executes the single instruction addressed by its operand. This new structure is, I believe, unique to the Clarion language, but has proven quite useful:

```
EXECUTE UpdateAction
  ADD(Master)
  PUT(Master)
  DELETE(Master)
END
```

Taming the User Interface

In 1970, I was working for McDonnell Douglas Automation Company when we purchased one of the first IV/70 computers built by Four Phase Systems, Inc. It was a marvelous machine—96K of solid-state memory, with a footprint not much larger than a PC. What made this box so interesting was its video support: 32 CRTs daisy-chained from 8 video ports that were refreshed directly from memory. Before the IV/70, every CRT I had used was a communications device. You could watch individual characters display as they arrived at the terminal. With the IV/70, an entire new screen was displayed every thirtieth of a second. It was the perfect platform for interactive programs. But no one seemed to notice. Four Phase was selling the system as a replacement for IBM's clustered CRTs and as a multi-station keypunch.

I had a higher use in mind. In 1973, I formed a company to develop a turn-key hospital information system based on the IV/70 computer. I wrote a multi-user operating system and a macro-language that exercised it. Then I wrote a macro pre-processor and a small hospital information system. The entire process took 9 months.

The macro language accessed the CRTs as if they were memory (that's what they were!) using move macros. The hospital application "painted" the screen by moving literals to the video memory, then placed entry field descriptions in a user field table and returned to the operating system for processing. When a field completed or a special key was pressed, control returned to the application.

This strategy had a distinct operating system "centric" viewpoint. Function keys were connected to screen procedures. Screen procedures created field tables that were connected to field edit procedures. A program didn't "run" in a conventional sense. In fact, there was no such thing as a program—just a set of procedures that responded to operating system events. The operating system was in control. It was up to the programmer to anticipate its needs. Our programmers eventually became so proficient with this approach that most hospital systems could be designed, implemented, and fully tested before the hardware was cabled together.

But it was never intuitive. Every one of our programmers climbed a steep learning curve. Event-driven programming is hard to grasp. Later, in one of the most vivid flashes of insight I have ever experienced, it dawned on me that an event-driven operating system could be controlled by a conventional program. The user interface would be invoked by a single statement. For Clarion, I called it ACCEPT. The leading edge of ACCEPT would return control to the operating system and the trailing edge could serve as the entry point for all event processing. A small set of functions would be crafted to identify the event that occurred and the fields involved.

Event-driven systems had always seemed "inside-out" to me. I was inside, chained to an oar, obeying the drummer, processing his events. I realized that ACCEPT would make me the master again. Now the drum was mine! I would call the operating system, not the other way around.

But how would Clarion depict a screen layout? Well, if screen literals are data and screen fields are data, then a screen layout has to be a data structure, doesn't it? I unimaginatively called it a **SCREEN** structure. **OPEN**(MyScreen) would display a screen. **ACCEPT** would enable the keyboard and handles all of the behavior of operator entry. When the operator completes a field or presses a "hot" key, the **ACCEPT** statement would "fall through," releasing control to the program. **CLOSE**(MyScreen) would restore the state of the monitor before "MyScreen" was opened.

Declaring screen layouts made them easy to process but even easier to design. The development team integrated a screen painter into the Clarion source code editor which could generate **SCREEN** structures. The screen painter could also read **SCREEN** structures. Get the picture? Position the cursor in a **SCREEN** structure and invoke the screen painter. The screen painter interprets the source and displays the screen layout. Now "paint" some changes on the screen and exit. The screen painter changes the source code by replacing the old **SCREEN** structure with the updated version. Interactive visual design like this is impossible without declared structures.

I designed a similar structure for report layouts. **REPORT** structures contain layouts for print lines, page headers and page footers. The **PRINT** statement handles data formatting and page overflow automatically. And a report painter is integrated with the source code editor to maintain **REPORT** structures just like **SCREEN** structures.

As luck would have it, our user interface design was perfectly suited to Microsoft Windows—an "inside out" operating system if I ever saw one. Windows programmers were having a very difficult time—who could blame them? The "Hello World" example shipped with a popular C++ product was 8 pages long! Windows was in desperate need of a simple messaging model like the Clarion ACCEPT loop. We decided to provide just that.

We changed our SCREEN structures to WINDOW structures, introducing the grammar necessary to declare and contain Windows objects and properties. We added multi-threading to accommodate the multiple document interface. We changed the grammar of REPORT structures to depict WYSIWYG reports, background forms, and nested group headers, footers, and sub-totals.

The ACCEPT statement became a structure defining the boundaries of an event processing loop. We designed the compiler to cooperate with the run-time library to hide the direction of the procedure calls used to process window events. A call to a run-time window processor is generated above the ACCEPT loop. The loop itself is generated as an embedded accept procedure.

The window processor creates the necessary objects, specifying a common event processing procedure for every event produced by every object. This event processor handles "housekeeping" events such as redraws and calls the embedded accept procedure to deal with other events. When the window closes, the window processor returns control to the statement following the ACCEPT loop.

To the Clarion programmer, it is all quite simple. Open a window, then fall into an ACCEPT loop. The ACCEPT loop cycles for every event the program needs to see. Close the window and fall out of the loop.

We defined a convenient set of functions to identify the events and objects involved. The code necessary to process a typical dialog box looks like this:

```
OPEN(Window)                !Open the window
ACCEPT                      !Enable the window
  CASE FIELD()              !Which field needs attention?
  OF ?OK                    ! 'OK' needs attention
    CASE EVENT()            !  Which event has occurred?
    OF EVENT:Selected       !   'OK' is pressed down
        :                   !    Process the OK button
      CLOSE(Window)         !    Close the window
    END                     !  End CASE EVENT()
  OF ?Cancel                ! 'Cancel' needs attention
    CASE EVENT()            !  Which event has occurred?
    OF EVENT:Selected       !   'Cancel' is pressed down
        :                   !    Process 'Cancel' button
      CLOSE(Window)         !    Close the window
    END                     !  End CASE EVENT()
  ELSE                      ! Must be a non-field event
    CASE EVENT()            !  Which event has occurred?
    OF EVENT:CloseWindow    !   The window will be closed
        :                   !    Process window close down
    END                     !  End CASE EVENT()
  END                       ! End CASE FIELD()
END                         !End ACCEPT
RETURN                      !Return to the caller
```

A by-product of our object-oriented run-time library corrected a serious deficiency in the Clarion language—compiler invariants. Declaring screens, reports, and files is very illuminating. But it can also be restrictive. Because they are compiled in, you can't change most declarations at run-time. Many of the language extensions requested by Clarion programmers involved making declared attributes visible to and changeable by the program

In our Windows run-time library, these structures are objects. Objects have properties. And properties can be changed. Anytime. Since we had already overloaded the period as both a structure terminator and a decimal point, we could not implement the standard object oriented notation of *object.property*. So we elected to use "curly brackets" to enclose properties. With this notation, any declared attribute, such as the text displayed on a button, can be modified by a statement such as:

```
?Button{PROP:Text} = 'My Button'
```

Designing a Database

I wanted to implement a simple database syntax that would support all three standard file access methods, direct, sequential, and indexed. The underlying file organization would also be simple: The file would contain a header followed by fixed length data records. The header would describe the record layout and associated keys and memos which would reside in separate files. This arrangement is similar to that used by dBase—a record could be accessed sequentially or directly by key or by its relative record number. I designed a **FILE** structure, similar to a COBOL **FD,** to declare files and their components:

```
Detail    FILE,PRE(DTL),NAME('C:\LEDGER\DETAIL.DAT')
AcctKey     KEY(DTL:AcctNo,DTL:Period,DTL:Date)
BatchKey    KEY(DTL:Batch,DTL:Period),DUP
Comment     MEMO(4096)
            RECORD              !Detail record
AcctNo        SHORT             !Account number
Period        BYTE              !Accounting period
Date          DATE              !Transaction date
Batch         STRING(12)        !Batch ID
Amount        DECIMAL(12,2)     !Amount (+/- = debit/credit)
            END
          END
```

I implemented sequential processing using **SET**, **NEXT**, **PREVIOUS**, and **SKIP** verbs. **SET** establishes the sequence (by key or relative record number) and starting point for the other three verbs which read records forward and backward, and skip over records. These verbs combine nicely with the end-of-file function (**EOF**) in a read loop:

```
SET(Dtl:AcctKey)                !Set account number sequence
LOOP UNTIL EOF(Detail)          !Loop through every record
  NEXT(Detail)                  !Read the next record
    :
END
```

The **GET** verb reads a record randomly by key or relative record number. Importantly, **GET** does not interfere with sequential processing by resetting the next record processed. **PUT** and **DELETE** process records accessed by **NEXT**, **PREVIOUS**, or **GET**. **ADD** inserts a new record in the database. This database access grammar proved to be efficient, robust and versatile—an essential and popular component of our product.

As the Clarion language spread, however, it took on new responsibilities. Clarion developers needed to access dBase files. So we added a dBase procedure library (we called Clarion procedure libraries "Language Extension Modules"—or LEMs). Then Novell came out with client-server support for Btrieve (server-based indexing). Some large Clarion applications needed Btrieve to improve their transaction throughput. So two of our third-party developers came out with Btrieve LEMs.

That left DB2. And RDB. And Oracle. And SQL Server. And every other variety of database that runs on or is accessed by PCs. We were planning to support direct C function calls in the next version of the language, so any database with a C language API could be accessed by a Clarion program. But it was clear to me that this was not the answer. Surely a general-purpose business language shouldn't be using a different grammar for every database format. Migrating a data file shouldn't require a major program overhaul. The Clarion language needed standardized, built-in support for all common databases.

It was suggested that we adopt SQL as our database grammar. I took the suggestion seriously and rewrote some typical Clarion programs using embedded SQL. It wasn't long before I realized this was a terrible idea. When used as a programming language, SQL is extremely verbose and inelegant. The little four statement record loop illustrated earlier becomes

this albatross under SQL:

```
DECLARE X CURSOR
  FOR SELECT     *
    FROM         Detail
    ORDER        BY Dtl:AcctNo,Dtl:Period,Dtl:Date
  END
END
OPEN X
LOOP
  FETCH X
  IF ReturnCode = 100 THEN BREAK.
    :
END
CLOSE X
```

Not only are SQL cursors inelegant, they are also nearly useless. You can't make a cursor skip—for example, to re-display a prior page of records. And you can't make it relocate—for example, to jump to "Jones" while browsing alphabetically. I concluded that if I were to replace the Clarion database access syntax with SQL, I would have been tarred and feathered and run out of town on a rail.

So we decided to implement replaceable database drivers. Clarion programmers liked their database grammar, they just needed support for other database formats. By building on the existing language structure, we would be leveraging their knowledge as well as enhancing their current applications. With our new database driver technology, we would make all databases look alike—a non-trivial benefit.

A New View

To produce SQL database drivers, we map SQL syntax onto our own database grammar. Our **SET** statement constructs an SQL **SELECT** statement which is issued at the first instance of a **NEXT** or **PREVIOUS** operation. If you change directions (e.g. **NEXT**...**PREVIOUS**), the driver issues another **SELECT** with a different **ORDER BY** clause. Our **GET** issues a **SELECT**...**FETCH**. **ADD** issues an **INSERT**; **GET**...**DELETE** issues a **DELETE**; and **GET**...**PUT** issues an **UPDATE**. A few features, such as relative record access, are not supported for SQL databases, but otherwise, the implementation is quite complete.

However, our database grammar was unable to exercise some very important SQL features. Clarion programs implement record filters by reading and throwing out unwanted records:

```
LOOP UNTIL EOF(Part)
  NEXT(Part)
  IF Prt:OnHand > 0 THEN CYCLE
    :
END
```

An SQL database can filter records on the server and save a lot of time. Clarion programs join files by reading the primary record to prime a key in

order to read the secondary record. An SQL database returns the primary and secondary records with a single access. And Clarion programs read every field in every record on every access. SQL returns only the fields you need.

Of course, an SQL database cannot read minds. You have to tell it what you want it to do. So we designed a VIEW structure for this purpose:

```
View  VIEW(Part),FILTER('PRT:OnHand = 0')
        PROJECT(PRT:Number,PRT:Name,PRT:OnHand,PRT:Usage)
        JOIN(Vendor,PRT:Vendor,VND:Number)
          PROJECT(VND:Name,VND:Address,VND:CityStateZip)
        END
      END
```

This VIEW structure consolidates the intentions of a Clarion program so that the database driver can utilize any services offered by its underlying database engine. The database driver either performs filter (record selection), join (record lookup), and project (field selection) operations or requests the database server to do so. In either case, performance is optimized.

There was also a problem implementing optimistic concurrency under SQL. To update a shared file, a Clarion program reads and saves a record. Then, before it is updated, the record is locked, reread, and compared to the saved copy. If they are the same, the changes are written to the database. Otherwise, the record has been changed by another workstation and the operator is so advised. This process is called "optimistic concurrency" and is based on the expectation that records are usually unchanged.

SQL implements optimistic concurrency with a WHERE clause that requires that all fields to be updated continue to have the same value. If one or more fields have changed, SQL returns an appropriate error. Since Clarion had no syntax to make such a request, we added a WATCH statement for this purpose. WATCH is issued before a GET, NEXT, or PREVIOUS to initiate optimistic concurrency. When the record is accessed, the driver saves a copy. In response to the PUT statement, the driver either rereads the record for comparison or issues an UPDATE...WHERE. to an SQL database. If the record has changed, PUT returns an error.

Our First Compiler

We shipped version 1.0 of Clarion in May of 1986 with both a compiler and an interpreter. The Clarion Compiler produced intermediate code that was then interpreted by the Clarion Processor. The intermediate code was so compact, that large Clarion applications would run on the small memory sizes (256K) that characterized PCs of that era. The compiler produced such tight code by generating a binary description of every declaration statement. Then the data was addressed by a two-byte pointer to the binary description. So it took five bytes to add an integer to a string and format the result according to a picture (one byte for the add operation and four bytes for the pointers to the integer and picture string descriptions). For every operation,

the Processor examined the data types of the elements involved and performed any necessary conversions.

But tight intermediate code wasn't the primary reason for this design. By interpreting the output from the compiler, the Processor could execute a Clarion application without requiring a link step. This was no small consideration. In 1985 and for a long time thereafter, linking was a time-consuming process. Our customers appreciated quick testing, but they also let us know that "real" programming languages produced **.**EXE files! Early the next year, we released the Clarion Translator that converted Clarion intermediate code into **.**OBJ files by replacing the operation codes with procedure calls. The pointers were passed as parameters. This strategy served us well for six years but also posed some problems:

- We had trouble with external libraries. **.**OBJ files could be linked into a Clarion **.**EXE, but they could not be executed directly by the Processor. We designed a process that converted a suitable **.**OBJ into a special binary format (LEM) that could be executed by the processor and changed back into an **.**OBJ by the Translator. But the process was complicated and was only used by sophisticated developers.
- Simple Clarion programs produced big **.**EXEs. The run-time decision making referenced library procedures that were included in the **.**EXE but never called. That made a "Hello World" program take 141K.
- Clarion applications ran slower than C, Pascal, and Modula-2 programs because Clarion programs examined data types at run-time while the other languages did so at compile time.
- It was no longer necessary to avoid linking in the test cycle. New linkers that supported run-time libraries could link a program for testing as fast as we could load the Processor.

Most importantly, we needed technology that would provide a development path to Windows, protected mode, OS/2, UNIX, 32-bit, and non-Intel architectures.

A New Partner

In May of 1990, we solved those problems and many others by licensing the TopSpeed technology from Jensen & Partners International (JPI), a British company. JPI was formed in 1988 when Niels Jensen, founder of Borland International, and his language development team left that company as a group. They purchased their work in progress and produced the TopSpeed product line, the top-rated compiler technology in the industry. JPI had

developed C, C++, Pascal, and Modula-2 compilers that shared the same optimizing code generator and project system. JPI called the compilers "front-ends" and the code generator the "back-end."

We started immediately writing a Clarion front-end. As usual, it was harder than we thought. The language required more changes than we expected. The project took longer and used more resources than we thought it would. But we were thrilled with the results.

We knew the TopSpeed back-end was good, but we were astonished when a Clarion "Sieve of Eratosthenes" (an algorithm for finding prime numbers) ran twice as fast as the same program written with Borland's Turbo C++. We had also licensed TopSpeed linking technology, but I hadn't realized just how good it was. TopSpeed's unique "Smart Linking" produced perfect granularity by eliminating all unreferenced procedures and static data elements from an .EXE. Better yet, while we were working on our front-end, JPI had developed an automatic overlay loader, DOS DLLs, a royalty-free DOS extender, and had announced 32-bit support. With this state-of-the-art technology, we had finally removed the performance penalty that had always been associated with high-level business languages.

In September of 1991, we announced our new product at the first Clarion Developers Conference. New features and the Clarion/TopSpeed connection drew rave reviews. Caught up in the festivity of the occasion, Niels Jensen, and I started talking about merging our companies. It made a great deal of sense. TopSpeed products would gain a US presence and access to a much larger programming market. Clarion products would own their core technology. We would be the first to apply leading edge compiler technology to business software development tools. After a lengthy negotiation, the merger was concluded in April of 1992. Two and a half years later, after the companies had completely homogenized their operations and product lines, the successor company was renamed TopSpeed Corporation. In October of 1994, TopSpeed Corporation released Clarion for Windows, the first product developed in its entirety by the merged companies.

Where We Stand Now

These remarks originally comprised the introduction to the *Programmer's Guide* that accompanied Clarion Database Developer Version 3.0, released in April of 1993. Extensive additions and revisions have been necessary for the Windows version of Clarion. Such is progress. I think of software development as the process of gently rocking a Chinese checker board until all the marbles fall into holes. I believe in the notion of a final, correct design. Until Clarion for Windows, I felt that we were a long way from our goal. Now I am not so sure. There are very few marbles rolling.

Introduction

Contents

The Language Reference Manual

Clarion for Windows is an integrated environment for writing data processing applications and management information systems for microcomputers using the Windows operating environment. Clarion's programming language is the foundation of this environment. In this manual, the language is concisely documented in a modular fashion. Although this is not a text book, you should consult this manual first when you want to know the precise syntax required to implement any declaration, statement, or function.

As far as possible, real-world example code is provided for each item.

Chapter Organization

CHAPTER 1 - Introduction provides an introduction to the Clarion Language Reference. It provides a brief overview of the contents of each chapter, and a guide to help the reader understand the documentation conventions used throughout the book.

CHAPTER 2 - Program Source Code Format provides the general layout of a Clarion Windows program. Punctuation, special characters, reserved words, and a detailed description of the "building blocks" required to create modular, structured Clarion source code are documented here.

CHAPTER 3 - Declaring Variables describes the data types and attributes used to declare variables in a Clarion program. In addition, formatting masks, called "picture tokens," are defined and illustrated.

CHAPTER 4 - Expressions and Assignments defines the syntax required to combine variables, functions, and constants into numeric, string, or logical expressions. It also defines how the value of an expression is assigned to variables.

CHAPTER 5 - Control Statements describes compound executable statements that control program flow and operation.

CHAPTER 6 - Window Structures describes the APPLICATION and WINDOW data structures and all their components and attributes.

CHAPTER 7 - Window Commmands describes the executable statements and functions that are specific to APPLICATION and WINDOW structures.

CHAPTER 8 - Reports describes the REPORT data structure and all its

components and attributes. The executable statements and functions that are specific to using a REPORT structure are also covered here.

CHAPTER 9 - Graphics Commands describes executable statements and functions that draw graphical figures in APPLICATION, WINDOW, and REPORT structures.

CHAPTER 10 - Data Files describes the FILE structure. This chapter covers the declarations, statements, and functions which access data files. The statements and functions required for multi-user and transaction processing systems are also documented here.

CHAPTER 11 - File Views describes the VIEW structure. This chapter covers the declarations, statements, and functions which access data files through the VIEW structure.

CHAPTER 12 - Memory Queues describes the QUEUE data structure, which is used to rapidly process information in random access memory. Along with all its components and attributes, the executable statements and functions that are specific to using a memory QUEUE are also covered here.

CHAPTER 13 - Miscellaneous Procedures and Functions documents the statements and functions that do not specifically apply to the subjects covered in chapters 1 through 12.

APPENDIX A - DDE Library Reference documents the statements and functions that perform Dynamic Data Exchange with other concurrently executing Windows programs.

Reference Item Format

Each Clarion programming language element referenced in this manual is printed in UPPER CASE letters. Components of the language are documented with a syntax diagram, a detailed description, and source code examples.

Items are documented in logical groupings, dependent upon their hierarchical relationships. Therefore, the table of contents for this book is not listed in alphabetical order. In general, data types and structures occur at the beginning of a chapter, followed by their attributes, and executable statements and functions at the end.

The documentation format used in this book is illustrated in the syntax diagram on the following page.

KEYWORD (short description of intended use)

[label] **KEYWORD(** | *parameter1* | [*parameter2*] **)** [**ATTRIBUTE1( )**] [**ATTRIBUTE2( )**]
| *alternate* |
| *parameter* |
| *list* |

KEYWORD A brief statement of what the KEYWORD does.

parameter1 A complete description of parameter1, along with how it relates to parameter2 and the KEYWORD.

parameter2 A complete description of parameter2, along with how it relates to parameter1 and the KEYWORD. Because it is enclosed in brackets, [ ], it is optional, and may be omitted.

alternate parameter list
A complete description of alternates to parameter1, along with how they relate to parameter2 and the KEYWORD.

ATTRIBUTE1 A sentence describing the relation of ATTRIBUTE1 to the KEYWORD.

ATTRIBUTE2 A sentence describing the relation of ATTRIBUTE2 to the KEYWORD.

A concise description of what the **KEYWORD** does. In many cases the KEYWORD will be an attribute of a keyword that was described in the preceding text. Sometimes a KEYWORD has no parameters and/or attributes.

Events Generated: If the KEYWORD generates events, they are listed here.

Return Data Type: The data type returned if KEYWORD is a function.

Errors Posted: If KEYWORD posts errors which may be trapped by the ERROR and ERRORCODE functions, they are listed here.

Example:

```
FieldOne = FieldTwo + FieldThree           !This is a source code example
FieldThree = KEYWORD(FieldOne,FieldTwo)    !Comments follow the "!" character
```

See Also: Other pertinent keywords and topics

Conventions and Symbols

Symbols are used in the syntax diagrams as follows:

Symbol	Meaning
[ ]	Brackets enclose an optional (not required) attribute or parameter.
( )	Parentheses enclose a parameter list.
| |	Vertical lines enclose parameters, where one, but only one, of the parameters is allowed.

Coding example conventions used throughout this manual:

CLARION KEYWORDS	All caps
DataNames	Mixed case with caps used for readability
Comments	Predominantly lower case

The purpose of these conventions is to make the code examples readable and clear.

Statement Format

Contents

Clarion is a “statement oriented” language. A statement oriented language makes use of the fact that its source code is contained in ASCII text files so every line of code is a separate record in the file. Therefore, the Carriage Return/Line Feed record delimiter can be used to eliminate punctuation.

In general, the Clarion statement format is:

```
label STATEMENT[(parameters)] [,ATTRIBUTE[(parameters)]] ...
```

Attributes specify the properties of the item and are only used on data declarations. Executable statements take the form of a standard procedure call, except assignment statements (A = B) and control structures (such as IF, CASE, and LOOP).

A statement’s label must begin in column one (1) of the source code. A statement without a label must not start in column one. A statement is terminated by the end of the line. A statement too long to fit on one line can be continued by a vertical bar (|). The semi-colon is an optional statement separator that allows you to place more than one statement on a line.

Being a statement oriented language eliminates from Clarion much of the punctuation required in other languages to identify labels and separate statements. Blocks of statements are initiated by a single compound statement, and are terminated by an END statement (or period).

Declaration and Statement Labels

The language statements in a source module can be divided into two general categories: data declarations and executable statements, or simply “data” and “code.”

During program execution, data declarations reserve memory storage areas that are manipulated by executable statements. A label is required for the data to be referenced in executable code. All variables, data structures, PROCEDUREs, FUNCTIONs, and ROUTINEs are referenced by labels.

A label defines a specific location in a PROGRAM. Any code statement may be identified and referenced by a label. This allows it to be used as the target of a GOTO statement. Each label on an executable statement adds ten bytes to the executable code size, even if not referenced.

The label on a PROCEDURE or FUNCTION statement is the procedure or function’s name. Using the label of a PROCEDURE in an executable statement executes the procedure. The label of a FUNCTION is used in expressions, or parameter lists of other functions, to assign the value returned by the function.

The rules for valid Clarion labels are:

- A label MUST begin in column one (1) of the source code.
- A label may contain letters (upper or lower case), numerals 0 through 9, the underscore character (_), and colon (:).
- The first character must be a letter or the underscore character.
- Labels are not case sensitive (i.e. CurRent and CURRENT are the same).
- A label may not be a reserved word.

StructureTermination

Compound data structures are created when data declarations are nested within other data declarations. There are many compound data structures within the Clarion language: APPLICATION, WINDOW, REPORT, FILE, RECORD, GROUP, VIEW, QUEUE, etc. These compound data structures must be terminated by a period (.) or the keyword END.

IF, CASE, EXECUTE, LOOP, BEGIN, and ACCEPT are all executable control structures. They must also be terminated with a period or the END statement.

Field Qualification

Variables declared as members of complex data structures (GROUP, QUEUE, FILE, RECORD, etc.) may have duplicate labels, as long as the duplicates are not contained within the same structure. To explicitly reference fields with duplicate labels in separate structures, you may use the PRE attribute on the structures just as it is documented (Prefix:FieldLabel) to provide unique names for each field. However, the PRE attribute is not required for this purpose and may be omitted.

Any field of any complex structure can be explicitly referenced by prepending the label of the structure containing the field to the field label, separated by a colon (StructureName:FieldLabel). You must use this Field Qualification syntax to reference any field in a complex structure that does not have a PRE attribute.

If the field is within nested complex data structures, you must prepend each successive level's structure label to the field label to explicitly reference the field (if the nested structure has a label). If any nested structure does not have a label, then that part is omitted from the qualification sequence. This is similar to anonymous unions in C++. This means that, in the case of a FILE structure (without a PRE attribute) in which the RECORD structure has a label, the individual fields in the file must be referenced as FileLabel:RecordLabel:FieldLabel. If the FILE's RECORD structure does not have a label, the individual fields are referenced as FileLabel:FieldLabel.

Example:

```
MasterFile  FILE,DRIVER('TopSpeed')
Record        RECORD
AcctNumber      LONG          !Referenced as Masterfile:Record:AcctNumber
            . .
Detail      FILE,DRIVER('TopSpeed')
              RECORD
AcctNumber      LONG          !Referenced as Detail:AcctNumber
            . .
Memory      GROUP,PRE(Mem)
Message       STRING(30)      !May be referenced as Mem:Message or Memory:Message
            END
SaveQueue   QUEUE
Field1        LONG            !Referenced as SaveQueue:Field1
Field2        STRING          !Referenced as SaveQueue:Field2
            END

OuterGroup  GROUP
Field1        LONG            !Referenced as OuterGroup:Field1
Field2        STRING          !Referenced as OuterGroup:Field2
InnerGroup    GROUP
Field1          LONG          !Referenced as OuterGroup:InnerGroup:Field1
Field2          STRING        !Referenced as OuterGroup:InnerGroup:Field2
              END
            END
```

See Also: PRE

Reserved Words

The following keywords are reserved and may not be used as labels for any purpose:

ACCEPT	AND	BEGIN	BREAK
BY	CASE	COMPILE	CYCLE
DO	EJECT	ELSE	ELSIF
END	EXECUTE	EXIT	FUNCTION
GOTO	IF	INCLUDE	LOOP
MEMBER	NOT	OF	OMIT
OR	OROF	PROCEDURE	PROGRAM
RETURN	ROUTINE	SECTION	THEN
TIMES	TO	UNTIL	WHILE
XOR			

The following keywords may be used as labels of data structures or executable statements. They may not be used as labels of PROCEDURE or FUNCTION statements:

APPLICATION	CODE	DETAIL	FILE
FOOTER	FORM	GROUP	HEADER
ITEM	JOIN	MAP	MENU
MENUBAR	MODULE	OPTION	QUEUE
RECORD	REPORT	ROW	SHEET
SUBTITLE	TAB	TABLE	TITLE
TOOLBAR	VIEW	WINDOW	

Special Characters

Initiators:	!	Exclamation point begins a source code comment.
	?	Question mark begins a field equate label.
	@	"At" sign begins a picture token.
	*	Asterisk begins a parameter passed by address in a MAP prototype.
Terminators:	;	Semi-colon is an executable statement separator.
	CR/LF	Carriage-return/Line-feed is an executable statement separator.
	.	Period terminates a data or code structure (a substitute for END).
	|	Vertical bar is the source code line continuation character.
	#	Pound sign declares an implicit LONG variable.
	$	Dollar sign declares an implicit REAL variable.
	"	Double quote declares an implicit STRING variable.
Delimiters:	( )	Parentheses enclose a parameter list.
	[ ]	Brackets enclose an array subscript list.
	' '	Single quotes enclose a string constant.
	{ }	Curly braces enclose a repeat count in a string constant, or a property parameter in an assignment statement.
	< >	Angle brackets enclose an ASCII code in a string constant, or indicate a parameter in a MAP prototype which may be omitted.
	:	Colon separates the start and stop positions of a string "slice."
Connecters:	.	Period is a decimal point used in numeric constants.
	,	Comma connects parameters in a parameter list.
	:	Colon connects a prefix to a label, or a complex structure label to the label of one of its members.
	$	Dollar sign connects the window to a field equate label in a control property assignment statement.
Operators:	+	Plus sign indicates addition.
	-	Minus sign indicates subtraction.
	*	Asterisk indicates multiplication.
	/	Slash indicates division.
	%	Percent sign indicates modulus division.
	^	Carat indicates exponentiation.
	<	Left angle bracket indicates less than.
	>	Right angle bracket indicates greater than.
	=	Equal sign indicates assignment or equivalence.
	~	Tilde indicates the logical "NOT" operator.
	&	Ampersand indicates concatenation.

Program Format

PROGRAM (declare a program)

```
PROGRAM
[MAP
        prototypes
   [MODULE( )
        prototypes
    END ]
 END ]
global data
   CODE
   statements
   [RETURN]
procedures or functions
```

PROGRAM	The first declaration in a Clarion program source module. Required.
MAP	Global procedure and function declarations. Required.
MODULE	Declare member source modules.
prototypes	PROCEDURE and/or FUNCTION declarations.
global data	Declare Global data which may be referenced by all procedures and functions.
CODE	Begin executable statements.
statements	Executable program instructions.
RETURN	Terminate program execution. Return to operating system **control.**
procedures or functions	Source code for the procedures and functions in the PROGRAM module.

The **PROGRAM** statement is required to be the first declaration in a Clarion program source module. It may only be preceded by source code comments or a TITLE or SUBTITLE compiler directive. The PROGRAM source file name is used as the object (.OBJ) and executable (.EXE) file name, when compiled. The PROGRAM statement may have a label, but the label is ignored by the compiler.

A PROGRAM with PROCEDUREs and/or FUNCTIONs must have a MAP structure. The MAP declares the PROCEDURE and/or FUNCTION prototypes. Any PROCEDURE or FUNCTION contained in a separate source file must be declared in a MODULE structure within the MAP.

Data declared in the PROGRAM module, between the keywords PROGRAM and CODE, is Global data that may be accessed by any

PROCEDURE or FUNCTION in the PROGRAM. Its memory allocation is Static.

Example:

```
                  PROGRAM                          !Sample program declaration
                  INCLUDE('EQUATES.CLW')           !Include standard equates
                  MAP
                    CalcTemp                       !Procedure Prototype
                  END
  CODE
  CalcTemp                                         !Call procedure

CalcTemp      PROCEDURE
Fahrenheit    REAL(0)                            !Global data declarations
Centigrade    REAL(0)
Window  WINDOW('Temperature Conversion'),CENTER,SYSTEM
          STRING('Enter Fahrenheit Temperature: '),AT(34,50,101,10)
          ENTRY(@N-04),AT(138,49,60,12),USE(Fahrenheit)
          STRING('Centigrade Temperature:'),AT(34,71,80,10),LEFT
          ENTRY(@N-04),AT(138,70,60,12),USE(Centigrade),SKIP
          BUTTON('Another'),AT(34,92,32,16),USE(?Another)
          BUTTON('Exit'),AT(138,92,32,16),USE(?Exit)
        END
  CODE                                          !Begin executable code section
  OPEN(Window)
  ACCEPT
    CASE ACCEPTED()
    OF ?Fahrenheit
      Centigrade = (Fahrenheit - 32) / 1.8
      DISPLAY(?Centigrade)
    OF ?Another
      Fahrenheit = 0
      Centigrade = 0
       DISPLAY
      SELECT(?Fahrenheit)
    OF ?Exit
      BREAK
    END
  END
  CLOSE(Window)
  RETURN
```

See Also: MAP, MODULE, PROCEDURE, FUNCTION, Data Declarations and Memory Allocation

MEMBER (identify member source file)

```
        MEMBER(program)
        [MAP
           prototypes
         END ]
[label] local data
        procedures or functions
```

MEMBER The first statement in a source module that is not a PROGRAM source file. Required.

program A string constant containing the filename (without extension) of a PROGRAM source file. This parameter is required.

MAP Local procedure and function declarations. Any procedures or functions declared here may be referenced only by the procedures or functions in the MEMBER module.

prototypes PROCEDURE and/or FUNCTION declarations.

local data Declare Local Static data which may be referenced only by the procedures and functions whose source code is in the MEMBER module.

procedures or functions
Source code for the procedures and functions in the MEMBER module.

MEMBER is the first statement required to be in a source module that is not a PROGRAM source file. It may only be preceded by source code comments or a TITLE or SUBTITLE compiler directive. It is required at the beginning of any source file that contains PROCEDUREs or FUNCTIONs that are used by a PROGRAM. The MEMBER statement identifies the *program* to which the source MODULE belongs.

A MEMBER module may have a local MAP structure. Procedures and functions declared in this MAP are visible only to the other procedures and functions in the MEMBER module. The source code for the procedures and functions declared in this MEMBER MAP may be contained in the MEMBER source file, or another file.

If the source code for the PROCEDURE or FUNCTION declared in a MEMBER MAP is contained in a separate file, the PROCEDURE or FUNCTION's *prototype* must be declared in a MODULE structure within the MEMBER MAP. That separate source file MEMBER MODULE must also contain its own MAP which declares the same *prototype* for that PROCEDURE or FUNCTION. Any PROCEDURE or FUNCTION not declared in the Global (PROGRAM) MAP must be declared in a local MAP in the MEMBER MODULE which contains its source code.

Data declared in the MEMBER module, after the keyword MEMBER and

before the first PROCEDURE or FUNCTION statement, is Member Local data that may only be accessed by PROCEDUREs or FUNCTIONs within the module (unless passed as a parameter). Its memory allocation is Static.

Example:

```
!Source1 module contains:
    MEMBER('OrderSys')        !Module belongs to the OrderSys program
    MAP                       !Declare local procedures
      Func1(STRING),STRING    !Func1 is known only in both module
      MODULE('Source2.clw')
        HistOrd2              !HistOrd2 is known only in both modules
      END
    END

LocalData STRING(10)          !Declare data local to MEMBER module

HistOrd   PROCEDURE           !Declare order history procedure
HistData  STRING(10)          !Declare data local to PROCEDURE
  CODE
  LocalData = Func1(HistData)

Func1 FUNCTION(RecField)      !Declare local function
  CODE
    !Executable code statements


!Source2 module contains:
    MEMBER('OrderSys')        !Module belongs to the OrderSys program
    MAP                       !Declare local procedures
      HistOrd2                !HistOrd2 is known only in both modules
      MODULE('Source1.clw')
        Func1(STRING),STRING  !Func1 is known only in both modules
      END
    END

LocalData STRING(10)          !Declare data local to MEMBER module

HistOrd2  PROCEDURE           !Declare second order history procedure
  CODE
  LocalData = Func1(LocalData)
```

See Also: MODULE, PROCEDURE, FUNCTION, Data Declarations and Memory Allocation

MAP (declare PROCEDURE and/or FUNCTION prototypes)

```
MAP
  prototypes
  [MODULE( )
    prototypes
   END ]
END
```

MAP Contains the *prototypes* which declare the functions, procedures and external source modules used in a PROGRAM or MEMBER module.

prototypes Declare a PROCEDURE or FUNCTION.

MODULE Declare a member source module.

A **MAP** structure contains the *prototypes* which declare the functions, procedures and external source modules used in a PROGRAM or MEMBER module. A MAP declared in the PROGRAM source module declares PROCEDUREs or FUNCTIONs that are available throughout the program. A MAP in a MEMBER module declares PROCEDUREs or FUNCTIONs that are available in that MEMBER module only.

A MAP structure is mandatory for any Clarion program because the BUILTINS.CLW file is automatically included in your PROGRAM's MAP structure by the compiler. This file contains prototypes of most of the procedures and functions in the Clarion internal library that are available as part of the Clarion language. This file is required because the compiler does not have these prototypes built into it (making it more efficient).Since the prototypes in the BUILTINS.CLW file use some constant EQUATEs that are defined in the EQUATES.CLW file, this file is also automatically included by the compiler in every Clarion program.

Example:

```
!One file contains:
  PROGRAM             !Sample program in sample.cla
  MAP                 !Begin map declaration
    LoadIt            ! LoadIt procedure
  END                 !End of map

!A separate file contains:
 MEMBER('Sample')     !Declare MEMBER module
 MAP                  !Begin local map declaration
  ComputeIt           ! compute it procedure
 END                  !End of map
```

See Also: PROGRAM, MEMBER, MODULE, FUNCTION and PROCEDURE Prototypes

MODULE (specify MEMBER source file)

```
MODULE(sourcefile)
  procedure prototype
  function prototype
END
```

MODULE Names a MEMBER module or external library file.

sourcefile A string constant. If the sourcefile contains Clarion language source code, this specifies the filename (without extension) of the source file which contains the PROCEDUREs and/or FUNCTIONs. If the sourcefile is an external library, this string may contain any unique identifier.

procedure prototype The prototype of a PROCEDURE contained in the sourcefile.

function prototype The prototype of a FUNCTION contained in the source***file*.**

A **MODULE** structure names a MEMBER module or external library file. It contains the *prototypes* for the PROCEDUREs and FUNCTIONs contained in the *sourcefile*. A MODULE structure can only be declared within a MAP structure.

Example:

```
!The "sample.cla" file contains:
  PROGRAM                 !Sample program in sample.cla
  MAP                     !Begin map declaration
   MODULE('Loadit')       ! source module loadit.cla
     LoadIt               !   load it procedure
    END                   ! end module
   MODULE('Compute')      ! source module compute.cla
     ComputeIt            !   compute it procedure
    END                   ! end module
  END                     !End map

!The "loadit.cla" file contains:
  MEMBER('Sample')        !Declare MEMBER module
  MAP                     !Begin local map declaration
   MODULE('Process')      ! source module process.cla
     ProcessIt            !   process it procedure
    END                   ! end module
  END                     !End map
```

See Also: MEMBER, MAP, FUNCTION and PROCEDURE Prototypes

PROCEDURE (declare a procedure)

```
label   PROCEDURE [(parameter list)]
        local data
        CODE
                statements
        [RETURN]
```

PROCEDURE	Begins a section of source code that can be executed from within a PROGRAM.
label	Names the PROCEDURE.
parameter list	An optional list of variables which pass values to the PROCEDURE. This list defines the name of each parameter as used within the PROCEDURE's source code. Each parameter is separated by a comma. The data type of each parameter is specified in the procedure's prototype in the MAP structure.
local data	Declare Local data which may be referenced only by this procedure.
CODE	Begin executable statements.
statements	Executable program instructions.
RETURN	Terminate procedure execution. Return to the point from which the procedure was called.

PROCEDURE begins a section of source code that can be executed from within a PROGRAM. It is called by naming the PROCEDURE *label* (with its *parameter list*, if any) as an executable statement in the code section of a PROGRAM, PROCEDURE, or FUNCTION.

A PROCEDURE terminates and returns to its caller when a RETURN statement is executed. An implicit RETURN occurs at the end of the executable code. The end of executable code for the PROCEDURE is defined as the end of the source file, or the first encounter of a FUNCTION, ROUTINE, or another PROCEDURE.

Data declared within a PROCEDURE, between the keywords PROCEDURE and CODE, is Procedure Local data that can only be accessed by that PROCEDURE (unless passed as a parameter to another PROCEDURE or FUNCTION). This data is allocated memory upon entering the procedure, and de-allocated when it terminates. If the data is smaller than the stack threshold (5K is the default) it is placed on the stack, otherwise it is allocated from the heap.

A PROCEDURE must be declared in the MAP of a PROGRAM or MEMBER module. If declared in the PROGRAM MAP, it is available to any other procedure or function in the program. If declared in a MEMBER MAP, it is only available to other procedures or functions in that MEMBER module.

Example:

```
  PROGRAM               !Example program code
  MAP
   OpenFile(FILE)       !Procedure prototype with parameter
   ShoTime              !Procedure prototype without parameter
  END
  CODE
  OpenFile(FileOne)     !Call procedure to open file
  ShoTime               !Call ShoTime procedure
   !More executable statements

OpenFile  PROCEDURE(AnyFile)      !Open any file
  CODE                            !Begin code section
  OPEN(AnyFile)                   !Open the file
  IF ERRORCODE() = 2              !If file not found
    CREATE(AnyFile)               ! create it
  END
  RETURN                          !Return to caller

ShoTime PROCEDURE                 !Show time
Time    LONG                      !Local variable
Window  WINDOW,CENTER
          STRING(@T3),USE(Time),AT(34,70)
          BUTTON('Exit'),AT(138,92,32,16),USE(?Exit)
        END
  CODE                            !Begin executable code section
  Time = CLOCK()                  !Get time from system
  OPEN(Window)
  ACCEPT
    CASE ACCEPTED()
    OF ?Exit
      BREAK
    END
  END
  RETURN                          !Return to caller
```

See Also: FUNCTION and PROCEDURE Prototypes, Data Declarations and Memory Allocation

FUNCTION (declare a function)

```
label   FUNCTION [(parameter list)]
        local data
        CODE
                statements
        RETURN(value)
```

FUNCTION	Begins a section of source code that can be executed from within a PROGRAM.
label	Names the FUNCTION.
parameter list	An optional list of variables which pass values to the FUNCTION. This list defines the name of each parameter as used within the FUNCTION's source code. Each parameter is separated by a comma. The data type of each parameter is specified in the procedure's prototype in the MAP structure.
local data	Declare Local data which may be referenced only by this function.
CODE	Begin executable statements.
statements	Executable program instructions.
RETURN	Terminate function execution and return the value to the expression in which the function was used.
value	A numeric or string constant or variable which specifies the result of the function call.

FUNCTION begins a section of source code that can be executed by naming the FUNCTION label with its *parameter list* (empty parentheses are required if no parameters are passed). FUNCTION execution is terminated by a RETURN statement in its CODE section (required).

A function can be used as an expression component, or a parameter of a PROCEDURE or another FUNCTION. A FUNCTION may also be called in the same manner as a PROCEDURE, if the program logic does not require the RETURN *value*. In this case, the compiler will generate a warning (unless its prototype has the PROC attribute) which may be safely ignored.

Data declared within a FUNCTION, between the keywords FUNCTION and CODE, is Procedure Local data that can only be accessed by that FUNCTION (unless passed as a parameter to another PROCEDURE or FUNCTION). This data is allocated memory on the stack upon entering the function, and de-allocated when it terminates.

A FUNCTION must be declared in the MAP of a PROGRAM or MEMBER module. If declared in the PROGRAM MAP, it is available to any other procedure or function in the program. If declared in a MEMBER MAP, it is only available to other procedures or functions in the MEMBER module.

Example:

```
  PROGRAM
  MAP
    FullName(STRING,STRING,STRING),STRING   !Function prototype with parameters
    DayString,STRING                        !Function prototype without parameters
  END
TodayString STRING(9)
  CODE
  TodayString = DayString()               !Function call without parameters
                                          ! the () is required for a function
    !Global executable statements

  START(NewThread)                        !Clarion START function called as a
                                          ! procedure -- generates compiler warning
                                          ! but executes correctly

FullName  FUNCTION(Last,First,Init)       !Full name function
  CODE                                    !Begin executable code section
  IF Init = ''                            !If no middle initial
    RETURN(CLIP(First) & ' ' & Last)      ! return full name
  ELSE                                    !Otherwise
    RETURN(CLIP(First) & ' ' & Init & '. ' & Last)
                                          ! return full name
  END

DayString FUNCTION                        !Day string function
  CODE                                    !Begin executable code section
  Day# = (TODAY() % 7) + 1                !Find day of week from system date
  EXECUTE Day#                            !Execute, return day string
    RETURN('Sunday')
    RETURN('Monday')
    RETURN('Tuesday')
    RETURN('Wednesday')
    RETURN('Thursday')
    RETURN('Friday')
    RETURN('Saturday')
  END
```

See Also: FUNCTION and PROCEDURE Prototypes

CODE (begin executable statements)

CODE

The **CODE** statement separates the data declaration section from the executable statement section within a PROGRAM, PROCEDURE, or FUNCTION. The first statement executed in a PROGRAM, PROCEDURE or FUNCTION is the statement following CODE.

Example:

```
OrdList PROCEDURE                   !Declare a procedure
!Data declarations go here
  CODE                              !This is the beginning of the "code" section
  !Executable statements go here
```

See Also: PROGRAM, PROCEDURE, FUNCTION

ROUTINE (declare local subroutine)

label **ROUTINE**

ROUTINE Declares the beginning of a local subroutine of executable statements.

label The name of the ROUTINE.

ROUTINE declares the beginning of a local subroutine of executable statements. It is local to the PROCEDURE or FUNCTION in which it is written and must be at the end of the CODE section of the PROCEDURE or FUNCTION to which it belongs. All variables visible to the PROCEDURE or FUNCTION are available in the ROUTINE. This includes all Procedure Local, Module Local, and Global data.

A ROUTINE is called by the DO statement followed by the label of the ROUTINE. Program control following execution of a ROUTINE is returned to the statement following the calling DO statement. A ROUTINE is terminated by the end of the source module, or by another ROUTINE, PROCEDURE, or FUNCTION. The EXIT statement can also be used to terminate execution of a ROUTINE's code (similar to RETURN in a PROCEDURE).

A ROUTINE is internally implemented by the compiler as a local procedure. Therefore, there are some efficiency issues that are not immediately obvious:

- DO and EXIT statements are very efficient.
- Accessing the PROCEDURE's local data is less efficent than accessing module data.
- Implicit variables used only within the ROUTINE are less efficient than using local variables.
- Each RETURN statement within a ROUTINE incurs a 40-byte overhead.

Example:

```
SomeProc PROCEDURE
  CODE
  !Code statements
  DO Tally                    !Call the routine
  !More code statements
Tally ROUTINE                 !Begin routine, end procedure
  IF CountVar < 55            !If less than 55
    CountVar += 1             ! increment counter
  ELSE                        ! otherwise
    CountVar = 0              ! reset the counter
    EXIT                      ! and exit the routine
  END                         !End if
```

See Also: EXIT, DO

END (terminate a structure)

END

END terminates a data declaration structure or a compound executable statement. It is functionally equivalent to a period (.).

Example:

```
Customer  FILE,DRIVER('Clarion')    !Declare a file
            RECORD                  ! begin record declaration
Name          STRING(20)
Number        LONG
            END                     ! end record declaration
          END                       !End file declaration

  CODE

  IF Number <> SavNumber            !Begin if structure
    DO GetNumber
  END                               !End if structure

  CASE Action                       !Begin case structure
  OF 1
    DO AddRec
  OF 2
    DO ChgRec
  OF 3
    DO DelRec
  END                               !End case structure
```

Statement Execution Sequence

In the CODE section of a Clarion program, statements are normally executed line-by-line, in the sequence in which they appear in the source module. Control statements, procedure calls, and function calls are used to modify this execution sequence.

PROCEDURE calls modify the execution sequence by branching to the called procedure and executing the code contained in it. Control returns to the executable statement following the procedure call when a RETURN statement is executed in the called procedure, or there are no more statements in the called procedure to execute.

FUNCTION calls modify the execution sequence by branching to the called function and executing the code contained in it. Control returns to the executable statement containing the function call when a RETURN statement is executed in the called function, returning the value of the function.

Control structures—IF, CASE, LOOP, and EXECUTE—change the execution sequence by evaluating expressions. When the expression is evaluated, the control structure conditionally executes statements contained within the structure.

Branching also occurs with the GOTO, DO, CYCLE, BREAK, EXIT, and RETURN statements. These statements immediately and unconditionally alter the normal execution sequence.

The START function begins a new execution thread, unconditionally branching to that thread. However, the user may choose to activate another thread by clicking the mouse on the other thread's active window.

Example:

```
  PROGRAM

  MAP
   ComputeTime(*GROUP)      !Passing a group parameter
   MatchMaster              !Passing no parameters
  END

ParmGroup GROUP             !Declare a group
FieldOne    STRING(10)
FieldTwo    LONG
          END
  CODE                      !Begin executable code
  FieldTwo = CLOCK()        !Executes 1st
  ComputeTime(ParmGroup)    !Executes 2nd, passes control to procedure
  MatchMaster               !Executes after procedure executes fully
```

PROCEDURE and FUNCTION Calls

procname[(*parameters*)]
return = *funcname*[(*parameters*)]

procname	The name of the PROCEDURE as declared in the procedure's prototype in the MAP. If this is not the label of a PROCEDURE statement, compiler errors are issued.
parameters	An optional parameter list passed to the PROCEDURE or FUNCTION. A parameter list may be one or more variable labels or expressions. The *parameters* are separated by commas and are declared in the prototype in the MAP.
return	The label of a variable to receive the value returned by the FUNCTION.
funcname	The name of the FUNCTION as declared in the procedure's prototype in the MAP. If this is not the label of a FUNCTION statement, compiler errors are issued.

A PROCEDURE is called by its label (including any parameter list) as a statement in the CODE section of a PROGRAM, PROCEDURE, or FUNCTION. The parameter list must match the parameter list declared in the procedure's prototype in the MAP. Procedures cannot be called in expressions.

A FUNCTION is called by its label (including any parameter list) as a component of an expression or parameter list passed to another PROCEDURE or FUNCTION. The parameter list must match the parameter list declared in the function's prototype in the MAP. A FUNCTION may also be called by its label (including any parameter list), in the same manner as a PROCEDURE, if its return value is not needed. This will generate a compiler warning that can be safely ignored.

Example:

```
  PROGRAM
  MAP
   ComputeTime(*GROUP)        !Passing a group parameter
   MatchMaster(),BYTE         !FUNCTION passing no parameters
  END
ParmGroup GROUP               !Declare a group
FieldOne    STRING(10)
FieldTwo    LONG
          END
  CODE
  FieldTwo = CLOCK()          !Built-in function called as expression
  ComputeTime(ParmGroup)      !Call the compute time procedure
  MatchMaster()               !Call the function as a procedure
```

See Also: FUNCTION and PROCEDURE Prototypes

Procedure Prototyping

FUNCTION and PROCEDURE Prototypes

name[(*parameter list*)] [,*return type*] [,*calling convention*] [, **RAW**] [, **NAME( )**] [, **TYPE**] [, **DLL**] [, **PROC**][, **PRIVATE**]

name	The label of a PROCEDURE or FUNCTION statement.
parameter list	The data types of the parameters. Each parameter's data type may be followed by a label used to document the parameter (only). Each parameter may also include an assignment of the default value (a constant) to pass if the parameter is omitted.
return type	The data type the FUNCTION will RETURN**.**
calling convention	Specify the C or PASCAL stack-based parameter calling convention.
RAW	Specifies that STRING or GROUP parameters pass only the memory address (without passing the length of the passed string). It also alters the behaviour of ? and *? parameters. This attribute is only for C compatability and is not valid on a Clarion language procedure.
NAME	Specify an alternate, "external" name for the PROCEDURE or FUNCTION. This attribute is only for other language compatability and is not valid on a Clarion language procedure.
TYPE	Specify the prototype is a type definition for procedures passed as parameters.
DLL	Specify the PROCEDURE or FUNCTION is in a .DLL.
PROC	Specify the FUNCTION may be called as a PROCEDURE without generating a compiler warning.
PRIVATE	Specify the PROCEDURE or FUNCTION may be called only from another PROCEDURE or FUNCTION within the same MODULE.

All PROCEDUREs and FUNCTIONs in a PROGRAM must be declared as a prototype in a MAP. A prototype is defined as the *name* of the PROCEDURE or FUNCTION, an optional *parameter list*, and the data *return type* (if prototyping a FUNCTION). You may specify the parameter *calling convention*, if you are linking in objects that require stack-based parameter passing (such as objects that were not compiled with a Clarion TopSpeed compiler).

The optional *parameter list* is a list of the data types that are passed to the PROCEDURE or FUNCTION. Each passed parameter in the *parameter list*

is delimited by commas, and the entire *parameter list* is enclosed in the parentheses following the *name*.

In the *parameter list*, each parameter's data type may be followed by a valid Clarion label which is completely ignored by the compiler (used only to document the purpose of the parameter). Each numeric value parameter's (passed by value) definition may also include the assignment of a constant value to the data type (or the documentary label, if present) that defines the default value to pass if the parameter is omitted.

Any parameter that may be omitted when the PROCEDURE or FUNCTION is called must be included in the prototype's *parameter list* and enclosed in angle brackets (< >) unless a default value is defined for the parameter. The OMITTED function allows you to test for unpassed paramters at runtime (except those parameters which have a default value defined).

You can optionally specify the C (right to left) or PASCAL (left to right and compatible with Windows for both 16-bit and 32-bit) stack-based parameter *calling convention* for your PROCEDURE or FUNCTION. This provides compatibility with third-party libraries written in other languages (if they were not compiled with a TopSpeed compiler). If you do not specify a *calling convention*, the default is the internal, register-based parameter passing convention used by all the TopSpeed compilers.

The RAW attribute allows you to pass just the memory address of a *?, STRING, or GROUP parameter (whether passed by value or by reference) to a non-Clarion language procedure or function. Normally, STRING or GROUP parameters pass both the address and the length of the string. The RAW attribute eliminates the length portion. This is provided for compatibility with external library functions which expect only the address of the string.

The NAME attribute provides the linker an external name for the PROCEDURE or FUNCTION. This is also provided for compatibility with libraries written in other languages. For example: in some C language compilers, with the C calling convention specified, the compiler adds a leading underscore to the function name. The NAME attribute allows the linker to resolve the name of the function correctly.

The TYPE attribute indicates the prototype does not reference a specific PROCEDURE or FUNCTION. Instead, it defines a prototype *name* used in other prototypes to indicate the type of procedure passed to another PROCEDURE or FUNCTION as a parameter.

The DLL attribute specifies that the PROCEDURE or FUNCTION for prototype on which it is placed is in a .DLL. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the procedure.

The PRIVATE attribute specifies that only another PROCEDURE or FUNCTION that is in the same MODULE may call it. This would most commonly be used on a prototype in a module's MAP structure, but may also be used in the global MAP.

When the *name* of a prototype is used in the *parameter list* of another prototype, it indicates the procedure being prototyped will receive the label of a PROCEDURE or FUNCTION that receives the same *parameter list* (and has the same *return type*, if it is a FUNCTION). A prototype with the TYPE attribute may not also have the NAME attribute.

Example:

```
MAP
  MODULE('Test')              !'test.clw' contains these procedure and functions
    MyProc1(LONG)             !LONG value-parameter
    MyProc2(<*LONG>)          !Omittable LONG variable-parameter
    MyProc3(LONG=23)          !Passes 23 if omitted
    MyProc4(LONG Count, REAL Sum)   !LONG passing a Count and REAL passing a Sum
    MyProc5(LONG Count=1, REAL Sum=0) !Count defaults to 1 and Sum to 0
    MyFunc1(*SREAL),REAL,C    !SREAL variable-parameter, REAL return, C call conv
    MyFunc2(FILE),STRING      !FILE entity-parameter, returning a STRING
    ProcType(FILE),TYPE       !Procedure-parameter type definition
    MyFunc3(ProcType),STRING
                              !ProcType procedure-parameter, returning a STRING,
                              ! must be passed a procedure that takes a FILE
                              ! as a parameter
    MyFunc4(FILE),STRING,PROC     !May be called as a procedure without warnings
    MyProc6(FILE),PRIVATE     !May only be called by other procs in TEST.CLW
  END
  MODULE('Party3.Obj')        !A third-party library
    Func46(*CSTRING),REAL,C,RAW
                              !Pass CSTRING address-only to C function
    Func47(*CSTRING),*CSTRING,C,RAW
                              !Returns pointer to a CSTRING
    Func48(REAL),REAL,PASCAL
                              !PASCAL calling convention
    Func49(SREAL),REAL,C,NAME('_func49')
                              !C convention and external function name
  END
  MODULE('STDFuncs.DLL')        !A standard functions .DLL
    Func50(SREAL),REAL,PASCAL,DLL
  END
END
```

See Also: MAP, MEMBER, MODULE, NAME, PROCEDURE, FUNCTION, RETURN, Passing Parameters

FUNCTION Return Types

A FUNCTION must RETURN a value. The data type to be returned is listed, separated by a comma, after the optional parameter list. Valid RETURN types are:

BYTE SHORT USHORT LONG ULONG SREAL REAL DATE TIME STRING CSTRING *BYTE *SHORT *USHORT *LONG *ULONG *SREAL *REAL *DATE *TIME
Untyped value-parameter return value (?)

An untyped value-parameter return value (?) indicates the data type of the value returned by the FUNCTION is not known. This functions in exactly the same manner as an untyped value-parameter. When the value is returned from the FUNCTION, standard Clarion Data Conversion Rules apply, no matter what data type is returned.

Functions which return pointers (the address of some data) should be prototyped with an asterisk prepended to the return data type (except CSTRING). This is provided just for compatibility with external library functions (written in another language) which return only the address of data. The compiler automatically handles the returned pointer at runtime. Functions prototyped this way act just like a variable defined in the program—when the function is used in Clarion code, the data referenced by the returned pointer is automatically used. This data can be assigned to other variables, passed as parameters to procedures or functions, or the ADDRESS function may return the address of the data.

CSTRING is an exception because all the others are fixed length datums, and a CSTRING is not. So, any C function that returns a pointer to a CSTRING can be prototyped as “char *” at the C end, but the compiler thunks the procedure and copies the datum onto the stack. Therefore, just like the other pointer return values, when the function is used in Clarion code the data referenced by the returned pointer is automatically used (the pointer is dereferenced).

As an example of this, assume that the XYZ() function returns a pointer to a CSTRING, CStringVar is a CSTRING variable, and LongVar is a LONG variable. The simple Clarion assignment statement, CStringVar = XYZ(), places the data referenced by the XYZ() function’s returned pointer, in the CStringVar variable. The assignment, LongVar = ADDRESS(XYZ()), places the memory address of that data in the LongVar variable.

Example:

```
MAP
  MODULE('Party3.Obj')       !A third-party library
   Func46(*CSTRING),REAL,C,RAW
                             !Pass CSTRING address-only to C function, return REAL
   Func47(*CSTRING),CSTRING,C,RAW
                             !Returns pointer to a CSTRING
   Func48(REAL),REAL,PASCAL
                             !PASCAL calling convention, return REAL
   Func49(SREAL),REAL,C,NAME('_func49')
                             !C convention and external function name, return REAL
  END
END
```

See Also: MAP, MEMBER, MODULE, NAME, FUNCTION, RETURN

C, PASCAL (parameter passing conventions)

C
PASCAL

The **C** and **PASCAL** attributes of a PROCEDURE or FUNCTION prototype specifies that parameters are always passed on the stack. The C convention passes the parameters from right to left as they appear in the parameter list, while the PASCAL convention passes them from left to right. PASCAL is also completely compatible with the Windows API calling convention for both 16-bit and 32-bit compiled applications, it gives the operating system's default calling convention. These calling conventions provide compatibility with third-party libraries written in other languages (if they were not compiled with a TopSpeed compiler). If you do not specify a calling convention in the prototype, the default calling convention is the internal, register-based parameter passing convention used by all the TopSpeed compilers.

Example:

```
MAP
  MODULE('Party3.Obj')       !A third-party library
   Func46(*CSTRING),REAL,C,RAW
                             !Pass CSTRING address-only to C function
  END
END
```

See Also: FUNCTION and PROCEDURE Prototypes, Passing Parameters

RAW (pass address only)

RAW

The **RAW** attribute of a PROCEDURE or FUNCTION prototype specifies that STRING or GROUP parameters pass the memory address only. This allows you to pass just the memory address of a *?, STRING, or GROUP parameter, whether passed by value or by reference, to a non-Clarion language procedure or function. Normally, STRING or GROUP parameters pass the address and the length of the string. The RAW attribute eliminates the length portion. For a prototype with a ? parameter, the parameter is taken as a LONG but passed as a "void *" which just eliminates linker warnings. This is provided for compatibility with external library functions which expect only the address of the string.

Example:

```
MAP
  MODULE('Party3.Obj')           !A third-party library
    Func46(*CSTRING),REAL,C,RAW  !Pass CSTRING address-only to C function
. .
```

See Also: FUNCTION and PROCEDURE Prototypes, Passing Parameters

NAME (set prototype's external name)

NAME(*constant* **)**

NAME Specifies an "external" name for the linker.

constant A string constant. This is case sensitive.

The **NAME** attribute specifies an "external" name for the linker. The NAME attribute may be placed on a FUNCTION or PROCEDURE Prototype. The *constant* supplies the external name used by the linker to identify the procedure or function from an external library.

Example:

```
PROGRAM
MAP
 MODULE('External.Obj')
  AddCount(LONG),LONG,C,NAME('_AddCount')   !C function named '_AddCount'
. .
```

See Also: FUNCTION and PROCEDURE Prototypes

TYPE (specify procedure or function type defintion)

TYPE

The **TYPE** attribute specifies a prototype that does not reference an actual PROCEDURE or FUNCTION. Instead, it defines a prototype *name* to use in other prototypes to indicate the type of procedure passed to another PROCEDURE or FUNCTION as a parameter.

When the *name* of the TYPEd prototype is used in the *parameter list* of another prototype, the procedure being prototyped will receive, as a passed parameter, the label of a PROCEDURE or FUNCTION that has the same type of *parameter list* (and has the same *return type*, if it is a FUNCTION).

Example:

```
MAP
   ProcType(FILE),TYPE       !Procedure-parameter type definition
   MyFunc3(ProcType),STRING
                             !ProcType procedure-parameter, returning a STRING,
                             ! must be passed the label of a procedure that takes
                             ! a FILE as a required parameter
END
```

See Also: FUNCTION and PROCEDURE Prototypes, Passing Parameters

DLL (set procedure defined externally in .DLL)

DLL([*flag*] **)**

DLL Declares a PROCEDURE or FUNCTION defined externally in a .DLL.

flag A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active. Uniquely, it may be an undefined label, in which case the attribute is active.

The **DLL** attribute specifies that the PROCEDURE or FUNCTION on whose prototype it is placed is defined in a .DLL. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the procedure.

Example:

```
MAP
  MODULE('STDFuncs.DLL')        !A standard functions .DLL
    Func50(SREAL),REAL,PASCAL,DLL
  END
END
```

PROC (set function called as procedure without warnings)

PROC

The **PROC** attribute specifies that the FUNCTION on whose prototype it is placed may be called as a PROCEDURE without generating compiler warnings. This allows you to use a FUNCTION as a PROCEDURE in those instances in which you do not need the return value from the FUNCTION.

Example:

```
MAP
  MODULE('STDFuncs.DLL')        !A standard functions .DLL
    Func50(SREAL),REAL,PASCAL,PROC
  END
END
```

PRIVATE (set procedure private to a single module)

PRIVATE

The **PRIVATE** attribute specifies that the PROCEDURE or FUNCTION on whose prototype it is placed may be called only from a PROCEDURE or FUNCTION within the same source MODULE. This encapsulates it from other modules.

Example:

```
MAP
  MODULE('STDFuncs.DLL')        !A standard functions .DLL
    Func49(SREAL),REAL,PASCAL,PROC
    Proc50(SREAL),PRIVATE   !Callable only from Func49
  END
END
```

Parameter Passing

ParameterTypes

There are four types of parameters that may be passed to a PROCEDURE or FUNCTION: **value-parameters**, **variable-parameters**, **entity-parameters**, and **procedure-parameters**.

Value-parameters are "passed by value." A copy of the variable passed in the parameter list of the "calling" PROCEDURE or FUNCTION is used in the "called" PROCEDURE or FUNCTION. The "called" PROCEDURE or FUNCTION cannot change the value of the variable passed to it by the "caller." Value-parameters are listed by data type in the PROCEDURE or FUNCTION prototype in the MAP. Valid value-parameters are:

```
BYTE SHORT USHORT LONG ULONG SREAL REAL DATE
TIME STRING
```

Variable-parameters are "passed by address." A variable passed by address has only one memory address. Changing the value of the variable in the "called" PROCEDURE or FUNCTION also changes its value in the "caller." Variable-parameters are listed by data type with a leading asterisk (*) in the PROCEDURE or FUNCTION prototype in the MAP. Valid variable-parameters are:

```
*BYTE *SHORT *USHORT *LONG *ULONG *SREAL *REAL
*BFLOAT4 *BFLOAT8 *DECIMAL *PDECIMAL *DATE *TIME
*STRING *PSTRING *CSTRING *GROUP
```

Entity-parameters pass the name of a data structure to the "called" PROCEDURE or FUNCTION. Passing the entity allows the "called" PROCEDURE or FUNCTION to use those Clarion commands that require the label of the structure as a parameter. Entity-parameters are listed by entity type in the PROCEDURE or FUNCTION prototype in the MAP. Entity-parameters are always "passed by address." Valid entity-parameters are:

```
FILE VIEW KEY INDEX QUEUE APPLICATION WINDOW
REPORT BLOB
```

Procedure-parameters pass the name of another PROCEDURE or FUNCTION to the "called" PROCEDURE or FUNCTION. Procedure-parameters are listed by the name of a preceding prototype of the same type in the PROCEDURE or FUNCTION prototype in the MAP (which may or may not have the TYPE attribute). When called in executable code, the "called" PROCEDURE or FUNCTION must be passed the name of a PROCEDURE or FUNCTION whose prototype is exactly the same as the

procedure named in the “called” procedure’s prototype.

Each parameter in the list may be followed by a valid Clarion label which is completely ignored by the compiler. This label is used only to document the the parameter to make the prototype more readable.

Each passed parameter’s definition may also include the assignment of a constant value to the data type (or the documentary label, if present) that defines the default value to pass if the parameter is omitted.

Example:

```
MAP
 MODULE('Test')              !'test.clw' contains these procedure and functions
   MyProc1(LONG)             !LONG value-parameter
   MyProc2(<*LONG>)          !Omittable LONG variable-parameter
   MyProc3(LONG=23)          !Passes 23 if omitted
   MyProc4(LONG Count, REAL Sum)   !LONG passing a Count and REAL passing a Sum
   MyProc5(LONG Count=1, REAL Sum=0) !Count defaults to 1 and Sum to 0
   MyFunc1(*SREAL),REAL,C    !SREAL variable-parameter, REAL return, C call conv
   MyFunc2(FILE),STRING      !FILE entity-parameter, returning a STRING
   ProcType(FILE),TYPE       !Procedure-parameter type definition
   MyFunc3(ProcType),STRING
                             !ProcType procedure-parameter, returning a STRING,
                             ! must be passed a procedure that takes a FILE
                             ! as a parameter
  END
  MODULE('Party3.Obj')       !A third-party library
   Func46(*CSTRING),REAL,C,RAW
                             !Pass CSTRING address-only to C function
   Func47(*CSTRING),*CSTRING,C,RAW
                             !Returns pointer to a CSTRING
   Func48(REAL),REAL,PASCAL
                             !PASCAL calling convention
   Func49(SREAL),REAL,C,NAME('_func49')
                             !C convention and external function name
  END
END
```

See Also: MAP, MEMBER, MODULE, NAME, PROCEDURE, FUNCTION, RETURN

Passing Parameters of Unspecified Data Type

The desire to write general purpose functions which perform some operation on a passed parameter, where the exact data type of the parameter may vary from one call to the next, is fairly common. Therefore, the function's prototype must indicate that the data type of the parameter is unknown at compile time. The Clarion language allows for this with **untyped value-parameters** and **untyped variable-parameters**. These are polymorphic parameters; they may become any other data type depending upon the data type passed to the procedure or function.

Untyped value-parameters are represented in the PROCEDURE or FUNCTION prototype with a question mark (?). When the procedure executes, the parameter is dynamically typed and acts as a data object of the base type (LONG, STRING, or REAL) of the passed variable, or the base type of whatever it was last assigned. This means that the "assumed" data type of the parameter can change within the PROCEDURE or FUNCTION, allowing it to be treated as any data type.

An untyped value-parameter is "passed by value" to the PROCEDURE or FUNCTION and its assumed data type is handled by Clarion's automatic Data Conversion Rules. Any changes made to the passed parameter within the PROCEDURE or FUNCTION do not affect the variable which was passed in.

Data types which may be passed as untyped value-parameters:

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4 BFLOAT8 DECIMAL PDECIMAL DATE TIME STRING PSTRING CSTRING GROUP (treated as a STRING) Untyped value-parameter (?) Untyped Variable-parameter (*?)

The RAW attribute can be specified if the untyped value-parameter (?) is being passed to external library functions written in other languages than Clarion. This converts the data to a LONG then passes the data as a C or C++ "void *" parameter (which eliminates "type inconsistency" warnings).

Untyped variable-parameters are represented in the PROCEDURE or FUNCTION prototype with an asterisk and a question mark (*?). Inside the procedure, the parameter acts as a data object of the type of the variable passed in at runtime. This means the data type of the parameter is fixed during the execution of the PROCEDURE or FUNCTION.

An untyped variable-parameter is "passed by address" to the PROCEDURE or FUNCTION. Therefore, any changes made to the passed parameter within the PROCEDURE or FUNCTION are made directly to the variable which was passed in. This allows you to write polymorphic functions.

Within a PROCEDURE or FUNCTION which receives an untyped variable-parameter, it is not safe to make any assumptions about the data type coming in. The danger of making assumptions is the possiblity of assigning an out-of-range value which the variable's actual data type cannot handle. If this happens, the result may be disastrously different from that expected.

Data types which may be passed as untyped variable-parameters:

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4 BFLOAT8 DECIMAL PDECIMAL DATE TIME STRING PSTRING CSTRING Untyped variable-parameter (*?)

The RAW attribute can be specified if the untyped variable-parameter (*?) is being passed to external library functions written in other languages than Clarion. This has the same effect as passing a C or C++ "void *" parameter.

Arrays may not be passed as either kind of untyped parameter.

Example:

```
  PROGRAM
  MAP
   Proc1(?)                !Untyped value-parameter
   Proc2(*?)               !Untyped variable-parameter
   Proc3(*?)               !Untyped variable-parameter (set to crash)
   Max(?,?),?              !Function returning Untyped value-parameter
  END
GlobalVar1 BYTE(3)         !BYTE initialized to 3
GlobalVar2 DECIMAL(8,2,3)
GlobalVar3 DECIMAL(8,1,3)
MaxInteger LONG
MaxString STRING(255)
MaxFloat   REAL
  CODE
  Proc1(GlobalVar1)        !Pass in a BYTE, value is 3
  Proc2(GlobalVar2)        !Pass it a DECIMAL(8,2), value is 3.00 - it prints 3.33
  Proc2(GlobalVar3)        !Pass it a DECIMAL(8,1), value is 3.0 - it prints 3.3
  Proc3(GlobalVar1)        !Pass it a BYTE and watch it crash
  MaxInteger = Max(1,5)    !Max function returns the 5
  MaxString = Max('Z','A') !Max function returns the 'Z'
  MaxFloat = Max(1.3,1.25) !Max function returns the 1.3
Proc1 PROCEDURE(ValueParm)
  CODE                                ! ValueParm starts at 3 and is a LONG
  ValueParm = ValueParm & ValueParm   !Now Contains '33' and is a STRING
  ValueParm = ValueParm / 10          !Now Contains 3.3 and is a REAL
Proc2 PROCEDURE(VariableParm)
  CODE
  VariableParm = 10 / 3               !Assign 3.33333333... to passed variable
Proc3 PROCEDURE(VariableParm)
  CODE
  LOOP
    IF VariableParm > 250 THEN BREAK. !If passed a BYTE, BREAK will never happen
    VariableParm += 10
  END
Max FUNCTION(Val1,Val1)               !Find the larger of two passed values
  CODE
  IF Val1 > Val2                      !Check first value against second
    RETURN(Val1)                      ! return first, if largest
  ELSE                                !otherwise
    RETURN(Val2)                      ! return the second
  END
```

See Also: FUNCTION and PROCEDURE Prototypes, Passing Parameters, Data Conversion Rules

Passing GROUPs and QUEUEs as Parameters

Passing a GROUP or a QUEUE to a PROCEDURE or FUNCTION which has been prototyped with GROUP or QUEUE types in its *parameter list* does not allow you to reference the component fields within the structure in the receiving PROCEDURE or FUNCTION. However, you can place the label of a GROUP or QUEUE in the prototype's *parameter list* to pass it by address and allow references to the component fields.

The GROUP or QUEUE named in the *parameter list* does not need the TYPE attribute, and does not have to be declared before the MAP structure, but it must be declared before the PROCEDURE or FUNCTION that will receive the parameter is called. This is the only case in the Clarion language that allows such a "forward reference."

The PROCEDURE or FUNCTION statement for the prototype may declare the local name of the passed group with a prefix to prevent name clashes, however this is unnecessary as long as you use the Field Qualification syntax to reference members of the passed group. The passed group can be a "superset" of the named paremeter, as long as the first fields in the "superset" group are the same as the named group.

Example:

```
                PROGRAM
                MAP
                  MyProc1(PassGroup,NameQue)
                        !Receives a GROUP defined the same as PassGroup and a QUEUE
                        ! defined the same as NameQue
                END
PassGroup       GROUP,TYPE              !Type definition: GROUP with 2 STRING(20) fields
F1                STRING(20)
F2                STRING(20)
                END
NameGroup       GROUP                       !Name group
First             STRING(20)                ! first name
Last              STRING(20)                ! last name
Company           STRING(30)
                END
NameQue         QUEUE                       !Name Queue
First             STRING(20)                ! first name
Last              STRING(20)                ! last name
                END
  CODE
  MyProc1(NameGroup,NameQue)            !Pass NameGroup and NameQue as parameters

MyProc1         PROCEDURE(PassedGroup,PassedQue)
  CODE
  PassedQue:First = PassedGroup:F1      !Assign NameGroup:First to NameQue:First
  PassedQue:Last  = PassedGroup:F2      !Assign NameGroup:Last to NameQue:Last
  ADD(PassedQue)                        !Add an entry into NameQue
```

See Also: FUNCTION and PROCEDURE Prototypes, GROUP, QUEUE, Field Qualification

Passing Arrays as Parameters

An array may be passed to a PROCEDURE or FUNCTION. The prototype in the MAP structure must declare the array's data type as a variable-parameter ("passed by address") with an empty subscript list. If the array is more than one dimension, commas must be used as position holders to indicate the number of dimensions in the array.

The calling statement should pass the entire array to the PROCEDURE or FUNCTION, not just one element.

Example:

```
  PROGRAM
  MAP
    MainProc
   AddCount(*LONG[,],*LONG[,])        !Passing two two-dimensional long arrays
  END
  CODE
  MainProc                            !Call first procedure

MainProc PROCEDURE
TotalCount  LONG,DIM(10,10)
CurrentCnt  LONG,DIM(10,10)
  CODE
  AddCount(TotalCount,CurrentCnt)     !Call the procedure passing the arrays

AddCount  PROCEDURE(Tot,Cur)          !Procedure expects two arrays
  CODE
  LOOP I# = 1 TO MAXIMUM(Tot,1)       !Loop through first subscript
    LOOP J# = 1 TO MAXIMUM(Tot,2)     !Loop through second subscript
      Tot[I#,J#] += Cur[I#,J#]        ! increment TotalCount from CurrentCnt
    END
  END
  CLEAR(Cur)                          !Clear CurrentCnt array
  RETURN
```

See Also: DIM, FUNCTION and PROCEDURE Prototypes, MAXIMUM

Program Structure Compiler Directives

Compiler Directives are statements that tell the compiler to take some action at compile time. These statements are not included in the executable program object code which the compiler generates. Therefore, there is no run-time overhead associated with their use.

BEGIN (define code structure)

```
BEGIN
  statements
END
```

BEGIN Declares a single code statement structure.

statements Executable program instructions.

The **BEGIN** compiler directive tells the compiler to treat the *statements* as a single structure. The BEGIN structure must be terminated by a period or the END statement.

BEGIN is used in an EXECUTE control structure to allow several lines of code to be treated as one.

Example:

```
EXECUTE Value
  Proc1           !Execute if Value = 1
  BEGIN           !Execute if Value = 2
    Value += 1
    Proc2
  END
  Proc3           !Execute if Value = 3
END
```

See Also: EXECUTE

COMPILE (specify source to be compiled)

COMPILE(*terminator* [,*expression*]**)**

COMPILE	Specifies a block of source code lines to be included in the compilation.
terminator	A string constant that marks the last line of a block of source code.
expression	An expression allowing conditional execution of the COMPILE. The expression is: EQUATE = integer.

The **COMPILE** directive specifies a block of source code lines to be included in the compilation. The included block begins with the COMPILE directive and ends with the line that contains the same string constant as the *terminator*. The entire terminating line is included in the COMPILE block.

The optional *expression* parameter permits conditional COMPILE. The form of the *expression* is fixed. It is the label of an EQUATE statement, or a Conditional Switch set in the Project System, followed by an equal sign (=), followed by an integer constant. The code between COMPILE and the *terminator* is compiled only if the *expression* is true. Although the *expression* is not required, COMPILE without an *expression* parameter is not necessary because all source code is compiled unless explicitly omitted. COMPILE and OMIT are opposites and may not be nested within each other, or themselves.

Example:

```
Demo EQUATE(1)                      !Specify the Demo EQUATE value
  CODE
  COMPILE('EndDemoChk',Demo = 1)    !COMPILE only if Demo equate is turned on
  DO DemoCheck                      !Check for demo limits passed
EndDemoChk                          !End of conditional COMPILE code
```

See Also: OMIT, EQUATE

EJECT (start new listing page)

EJECT([*module subtitle*]**)**

EJECT Starts a new page in a Clarion listing.

module subtitle A string constant containing the subtitle to be printed. On the next page of the listing, the *module subtitle* is printed in the first column of the third line.

The **EJECT** directive starts a new page and an optional new *module subtitle* in a Clarion listing. If the *module subtitle* parameter is omitted, the subtitle set by a previous SUBTITLE or EJECT directive will be used on the next page.

Example:

```
EJECT('File Declarations')      !Start new page, new subtitle
```

INCLUDE (compile code in another file)

INCLUDE(*filename* [,*section*]**)**

INCLUDE Specifies source code to be compiled which exists in a separate file which is not a MEMBER module.

filename A string constant that contains the DOS file specification for a source file. If the extension is omitted, .CLW is assumed.

section A string constant which is the *string* parameter of the SECTION directive marking the beginning of the source code to be included.

The **INCLUDE** directive specifies source code to be compiled which exists in a separate file which is not a MEMBER module. Starting on the line of the INCLUDE directive, the source file, or the specified *section* of that file, is compiled as if it appeared in sequence within the source module being compiled. You can nest INCLUDEs up to 3 deep, so you can INCLUDE a file that includes a file that includes a file but that latter file must not include anything....

The compiler uses the Redirection file (CW15.RED) to find the file, searching the path specified for that type of *filename* (usually by extension). This makes it unnecessary to provide a complete path in the *filename* to be included. A discussion of the Redirection file is in the *User's Guide.*

Example:

```
GenLedger PROCEDURE             !Declare procedure
  INCLUDE('filedefs.clw')       !Include file definitions here
   CODE                         !Begin code section
  INCLUDE('Setups','ChkErr')    !Include error check from setups.clw
```

OMIT (specify source not to be compiled)

OMIT(*terminator* [,*expression*]**)**

OMIT	Specifies a block of source code lines to be omitted from the compilation.
terminator	A string constant that marks the last line of a block of source code.
expression	An expression allowing conditional execution of the OMIT. The expression must be: EQUATE = integer.

The **OMIT** directive specifies a block of source code lines to be omitted from the compilation. These lines may contain source code comments or a section of code that has been "stubbed out" for testing purposes. The omitted block begins with the OMIT directive and ends with the line that contains the same string constant as the *terminator*. The entire terminating line is included in the OMIT block.

The optional *expression* parameter permits conditional OMIT. The form of the *expression* is fixed. It is the label of an EQUATE statement, or a Conditional Switch set in the Project System, followed by an equal sign (=), followed by an integer constant. The OMIT directive executes only if the *expression* is true.

COMPILE and OMIT are opposites and may not be nested within each other, or themselves.

Example:

```
  OMIT('**END**')   !Unconditional OMIT
************************************************
*
* Main Program Loop
*
************************************************
**END**
Demo EQUATE(0)                  !Specify the Demo EQUATE value
  CODE
  OMIT('EndDemoChk',Demo = 0)   !OMIT only if Demo is turned off
  DO DemoCheck                  !Check for demo limits passed
EndDemoChk                      !End of omitted code
```

See Also: COMPILE, EQUATE

SECTION (specify source code section)

SECTION(*string*)

SECTION Identifies the beginning of a block of executable source code or data declarations.

string A string constant which names the SECTION.

The **SECTION** compiler directive identifies the beginning of a block of executable source code or data declarations which may be INCLUDEd in source code in another file. The SECTION's *string* parameter is used as an optional parameter of the INCLUDE directive to include a specific block of source code. A SECTION is terminated by the next SECTION or the end of the file.

Example:

```
 SECTION('FirstSection')        !Begin section

FieldOne STRING(20)
FieldTwo LONG

 SECTION('SecondSection')       !End previous section, begin new section

  IF Number <> SavNumber
    DO GetNumber
  END

 SECTION('ThirdSection')        !End previous section, begin new section

  CASE Action
  OF 1
    DO AddRec
  OF 2
    DO ChgRec
  OF 3
    DO DelRec
  END                           !Third section ends at end of file
```

See Also: INCLUDE

SUBTITLE (print MODULE subtitle)

SUBTITLE(*module subtitle***)**

SUBTITLE Declares a listing subtitle printed in the first column of the third line of a Clarion listing.

module subtitle A string constant containing the subtitle to be printed.

A **SUBTITLE** is printed in the first column of the third line of a Clarion listing. The SUBTITLE directive does not print in the listing. The SUBTITLE directive must be placed at the beginning of a source module prior to the PROGRAM or MEMBER declarations. The subtitle remains the same on every page of the listing unless it is changed by an EJECT directive.

Example:

```
SUBTITLE('Global Data Declarations')
```

TITLE (print MODULE title)

TITLE(*module title***)**

TITLE Declares a listing title printed in the first column of the first line of a Clarion listing.

module title A string constant containing the title to be printed.

A **TITLE** is printed in the first column of the first line of a Clarion listing. The TITLE directive does not print in the listing. The TITLE directive must be placed at the beginning of a source module prior to the PROGRAM or MEMBER declarations. The title remains the same on every page of the listing.

Example:

```
TITLE('ORDERSYS - Order Entry System Listing')
```

Variable Declaration Statements

Contents

BYTE (one-byte unsigned integer)

label **BYTE(***initial value***)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

BYTE A one-byte unsigned integer.

```
Format: magnitude
        | ....... |
Bits:   7         0
Range:  0 to 255
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data**.**

AUTO Specify the variable has no *initial value.*

BYTE declares a one-byte unsigned integer.

Example:

```
Count1  BYTE                        !Declare one byte integer
Count2  BYTE,OVER(Count1)           !Declare OVER the one byte integer
Count3  BYTE,DIM(4)                 !Declare it an array of 4 bytes
Count4  BYTE(5)                     !Declare with initial value
Count5  BYTE,EXTERNAL               !Declare as external
Count6  BYTE,EXTERNAL,DLL           !Declare as external in a .DLL
Count7  BYTE,NAME('SixCount')       !Declare with external name
ExampleFile  FILE,DRIVER('Clarion') !Declare a file
Record       RECORD
Count8        BYTE,NAME('Counter')  !Declare with external name
             . .
```

SHORT (two-byte signed integer)

label **SHORT(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

SHORT A two-byte signed integer.

```
Format:    ±     magnitude
         | . | ............... |
Bits:   15   14                0
Range:  -32,768 to 32,767
```

i*nitial value*	A numeric constant. If omitted, the initial value is zero.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no *initial value*.

SHORT declares a two-byte signed integer, using the Intel 8086 word integer format. The high-order bit of this configuration is the sign bit (0 = positive, 1 = negative). Negative values are represented in standard two’s complement notation.

Example:

```
Count1  SHORT                          !Declare two-byte signed integer
Count2  SHORT,OVER(Count1)             !Declare OVER the two-byte signed integer
Count3  SHORT,DIM(4)                   !Declare it an array of 4 shorts
Count4  SHORT(5)                       !Declare with initial value
Count5  SHORT,EXTERNAL                 !Declare as external
Count6  SHORT,EXTERNAL,DLL             !Declare as external in a .DLL
Count7  SHORT,NAME('SixCount')         !Declare with external name
ExampleFile FILE,DRIVER('Clarion')     !Declare a file
Record        RECORD
Count7          SHORT,NAME('Counter') !Declare with external name
              . .
```

USHORT (two-byte unsigned integer)

label **USHORT(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

USHORT A two-bye unsigned integer.

```
Format:      magnitude
         | ............... |
Bits:   15                 0
Range:   0 to 65,535
```

initial value	A numeric constant. If omitted, the initial value is zero.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no *initial value.*

USHORT declares a two-byte unsigned integer in the Intel 8086 word format. There is no sign bit in this configuration.

Example:

```
Count1  USHORT                          !Declare two-byte unsigned integer
Count2  USHORT,OVER(Count1)             !Declare OVER the two-byte unsigned integer
Count3  USHORT,DIM(4)                   !Declare it an array of 4 unsigned shorts
Count4  USHORT(5)                       !Declare with initial value
Count5  USHORT,EXTERNAL                 !Declare as external
Count6  USHORT,EXTERNAL,DLL             !Declare as external in a .DLL
Count7  USHORT,NAME('SixCount')         !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')      !Declare a file
Record        RECORD
Count8          USHORT,NAME('Counter')    !Declare with external name
           . .
```

LONG (four-byte signed integer)

label **LONG(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

LONG A four-byte unsigned integer.

```
Format:    ±            magnitude
          | . | ............................ |
Bits:    31   30                              0
Range:  -2,147,483,648 to 2,147,483,647
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data**.**

AUTO Specify the variable has no *initial value*.

LONG declares a four-byte signed integer, using the Intel 8086 long integer format. The high-order bit is the sign bit (0 = positive, 1 = negative). Negative values are represented in standard two’s complement notation.

Example:

```
Count1  LONG                        !Declare four-byte signed integer
Count2  LONG,OVER(Count1)           !Declare OVER the four-byte signed integer
Count3  LONG,DIM(4)                 !Declare it an array of 4 longs
Count4  LONG(5)                     !Declare with initial value
Count5  LONG,EXTERNAL               !Declare as external
Count6  LONG,EXTERNAL,DLL           !Declare as external in a .DLL
Count7  LONG,NAME('SixCount')       !Declare with external name
ExampleFile  FILE,DRIVER('Clarion') !Declare a file
Record        RECORD
Count8          LONG,NAME('Counter')  !Declare with external name
              . .
```

ULONG (four-byte unsigned integer)

label **ULONG(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

ULONG A four-byte unsigned integer.

```
Format:              magnitude
         | ............................. |
Bits:   31                                0
Range:  0 to 4,294,967,295
```

initial value	A numeric constant. If omitted, the initial value is zero.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no *initial value.*

ULONG declares a four-byte unsigned integer, using the Intel 8086 long integer format. There is no sign bit in this configuration.

Example:

```
Count1  ULONG                       !Declare four-byte unsigned integer
Count2  ULONG,OVER(Count1)          !Declare OVER four-byte unsigned integer
Count3  ULONG,DIM(4)                !Declare it an array of 4 unsigned longs
Count4  ULONG(5)                    !Declare with initial value
Count5  ULONG,EXTERNAL              !Declare as external
Count6  ULONG,EXTERNAL,DLL          !Declare as external in a .DLL
Count7  ULONG,NAME('SixCount')      !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')  !Declare a file
Record        RECORD
Count8          ULONG,NAME('Counter') !Declare with external name
           . .
```

SREAL (four-byte signed floating point)

label **SREAL(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

SREAL A four-byte floating point number.

```
Format:   ±    exponent      significand
        | . | ......... | ....................... |
Bits:  31   30          23                        0
Range:  0, ± 1.175494e-38 .. ± 3.402823e+38 (6 significant digits)
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

SREAL declares a four-byte floating point signed numeric variable, using the Intel 8087 short real (single precision) format.

Example:

```
Count1  SREAL                       !Declare four-byte signed floating point
Count2  SREAL,OVER(Count1)          !Declare OVER the four-byte
                                    ! signed floating point
Count3  SREAL,DIM(4)                !Declare it an array of 4 floats
Count4  SREAL(5)                    !Declare with initial value
Count5  SREAL,EXTERNAL              !Declare as external
Count6  SREAL,EXTERNAL,DLL          !Declare as external in a .DLL
Count7  SREAL,NAME('SixCount')      !Declare with external name
ExampleFile  FILE,DRIVER('Btrieve') !Declare a file
Record       RECORD
Count8       SREAL,NAME('Counter') !Declare with external name
             . .
```

REAL (eight-byte signed floating point)

label **REAL(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

REAL An eight-byte floating point number.

```
Format:   ±     exponent                  significand
        | . | ........... | ..................................... |
Bits:   63  62            52                                      0
Range:  0, ± 2.225073858507201e-308 .. ± 1.79769313496231e+308
          (15 significant digits)
```

initial value	A numeric constant. If omitted, the initial value is zero.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no *initial value.*

REAL declares an eight-byte floating point signed numeric variable, using the Intel 8087 long real (double precision) format.

Example:

```
Count1  REAL                        !Declare eight-byte signed floating point
Count2  REAL,OVER(Count1)           !Declare OVER the eight-byte
                                    !  signed floating point
Count3  REAL,DIM(4)                 !Declare it an array of 4 reals
Count4  REAL(5)                     !Declare with initial value
Count5  REAL,EXTERNAL               !Declare as external
Count6  REAL,EXTERNAL,DLL           !Declare as external in a .DLL
Count7  REAL,NAME('SixCount')       !Declare with external name
ExampleFile FILE,DRIVER('Clarion')  !Declare a file
Record        RECORD
Count8          REAL,NAME('Counter')    !Declare with external name
              . .
```

BFLOAT4 (four-byte signed floating point)

label **BFLOAT4(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

BFLOAT4 A four-byte floating point number.

```
Format:    exponent     ±                 significand
        | ........... | . | ..................................... |
Bits:  31             23  22                                      0
Range:  0, ± 5.87747e-39 .. ± 1.70141e+38   (6 significant digits)
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

BFLOAT4 declares a four-byte floating point signed numeric variable, using the Microsoft BASIC (single precision) format. This data type is normally used for compatibility with existing data since it is internally converted to a REAL before all arithmetic operations.

Example:

```
Count1  BFLOAT4                        !Declare four-byte signed floating point
Count2  BFLOAT4,OVER(Count1)           !Declare OVER the four-byte
                                       !  signed floating point
Count3  BFLOAT4,DIM(4)                 !Declare array of 4 single-precision reals
Count4  BFLOAT4(5)                     !Declare with initial value
Count5  BFLOAT4,EXTERNAL               !Declare as external
Count6  BFLOAT4,EXTERNAL,DLL           !Declare as external in a .DLL
Count7  BFLOAT4,NAME('SixCount')       !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')     !Declare a file
Record       RECORD
Count8         BFLOAT4,NAME('Counter')   !Declare with external name
            . .
```

BFLOAT8 (eight-byte signed floating point)

label **BFLOAT8(**[*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

BFLOAT8 An eight-byte floating point number.

```
Format:   exponent      ±               significand
        | .......... | . | ..................................... |
Bits:  63             55  54                                     0
Range:  0, ± 5.877471754e-39 .. ± 1.7014118346e+38
            (15 significant digits)
```

***i**nitial value*	A numeric constant. If omitted, the initial value is zero.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no *initial value.*

BFLOAT8 declares an eight-byte floating point signed numeric variable, using the Microsoft BASIC (double precision) format. This data type is normally used for compatibility with existing data since it is internally converted to a REAL before all arithmetic operations.

Example:

```
Count1  BFLOAT8                        !Declare eight-byte signed floating point
Count2  BFLOAT8,OVER(Count1)           !Declare OVER the eight-byte
                                       !  signed floating point
Count3  BFLOAT8,DIM(4)                 !Declare it an array of 4 reals
Count4  BFLOAT8(5)                     !Declare with initial value
Count5  BFLOAT8,EXTERNAL               !Declare as external
Count6  BFLOAT8,EXTERNAL,DLL           !Declare as external in a .DLL
Count7  BFLOAT8,NAME('SixCount')       !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')     !Declare a file
Record        RECORD
Count8          BFLOAT8,NAME('Counter')   !Declare with external name
              . .
```

DECIMAL (signed packed decimal)

label **DECIMAL(***length* [,*places*] [,*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

DECIMAL A packed decimal floating point number.

```
Format:   ±                    magnitude
        | . | ..........................................  |
Bits: 127   124                                          0
Range:  -9,999,999,999,999,999,999,999,999,999,999,999  to
        +9,999,999,999,999,999,999,999,999,999,999,999
```

length A required numeric constant containing the total number of decimal digits (integer and fractional portion combined) in the variable. The maximum *length* is 31.

places A numeric constant that fixes the number of decimal digits in the fractional portion (to the right of the decimal point) of the variable. It must be less than or equal to the *length* parameter. If omitted, the variable will be declared as an integer.

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

DECIMAL declares a variable length packed decimal signed numeric variable. Each byte of a DECIMAL holds two decimal digits (4 bits per digit). The left-most byte holds the sign in its high-order nibble (0 = positive, anything else is negative) and one decimal digit. Therefore, DECIMAL variables always contain a fixed "odd" number of digits (DECIMAL(10) and DECIMAL(11) both use 6 bytes).

Example:

```
Count1  DECIMAL(5,0)                     !Declare three-byte signed packed decimal
Count2  DECIMAL(5),OVER(Count1)              !Declare OVER the three-byte
                                             !  signed packed decimal
Count3  DECIMAL(5,0),DIM(4)                  !Declare it an array of 4 decimals
Count4  DECIMAL(5,0,5)                       !Declare with initial value
Count5  DECIMAL(5,0),EXTERNAL                !Declare as external
Count6  DECIMAL(5,0),EXTERNAL,DLL            !Declare as external in a .DLL
Count7  DECIMAL(5,0),NAME('SixCount')        !Declare with external name
ExampleFile FILE,DRIVER('Clarion')           !Declare a file
Record        RECORD
Count8          DECIMAL(5,0),NAME('Counter')  !Declare with external name
             . .
```

PDECIMAL (signed packed decimal)

label **PDECIMAL(***length* [,*places*] [,*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

PDECIMAL A packed decimal floating point number.

```
Format:                    magnitude                          ±
       | ..................................................... | . |
Bits:  127                                                   4   0
Range:  -9,999,999,999,999,999,999,999,999,999,999,999  to
       +9,999,999,999,999,999,999,999,999,999,999,999
```

length A required numeric constant containing the total number of decimal digits (integer and fractional portion combined) in the variable. The maximum *length* is 31.

places A numeric constant that fixes the number of decimal digits in the fractional portion (to the right of the decimal point) of the variable. It must be less than or equal to the *length* parameter. If omitted, the variable will be declared as an integer.

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

PDECIMAL declares a variable length packed decimal signed numeric variable in the Btrieve and IBM/EBCDIC type of format. Each byte of an PDECIMAL holds two decimal digits (4 bits per digit). The right-most byte holds the sign in its low-order nibble (0Fh or 0Ch = positive, 0Dh = negative) and one decimal digit. Therefore, PDECIMAL variables always contain a fixed “odd” number of digits (PDECIMAL(10) and PDECIMAL(11) both use 6 bytes).

Example:

```
Count1  PDECIMAL(5,0)                    !Declare three-byte signed packed decimal
Count2  PDECIMAL(5),OVER(Count1)             !Declare OVER the three-byte
                                            ! signed packed decimal
Count3  PDECIMAL(5,0),DIM(4)                !Declare it an array of 4 decimals
Count4  PDECIMAL(5,0,5)                     !Declare with initial value
Count5  PDECIMAL(5,0),EXTERNAL              !Declare as external
Count6  PDECIMAL(5,0),EXTERNAL,DLL          !Declare as external in a .DLL
Count7  PDECIMAL(5,0),NAME('SixCount')      !Declare with external name
ExampleFile FILE,DRIVER('Btrieve')          !Declare a file
Record        RECORD
Count8          PDECIMAL(5,0),NAME('Counter') !Declare with external name
              . .
```

STRING (fixed-length string)

```
                  |  length         |
label  STRING(    | string constant |) [,DIM( )][,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC]
                  |  picture        |       [,THREAD] [,AUTO]
```

STRING A character string.

```
Format: A fixed number of bytes.
Size:   1 to 65,520 bytes in 16-bit, or 4MB in 32-bit.
```

length	A numeric constant that defines the number of bytes in the STRING. String variables are not initialized unless given a *string constant.*
string constant	The initial value of the STRING. The length of the STRING (in bytes) is set to the length of the *string constant*.
picture	Used to format the values assigned to the STRING. The length is the number of bytes needed to contain the formatted STRING.
DIM	Dimension the variable as an array.
OVER	Share a memory location with another variable.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
AUTO	Specify the variable has no *initial value*.

STRING declares a fixed-length character string. The memory assigned to the STRING is initialized to all blanks unless the AUTO attribute is present.

In addition to its explicit declaration, all STRING variables are also implicitly declared as STRING(1),DIM(*length of string*). This allows each character in the STRING to be addressed as an array element. If the STRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a STRING using the “string slicing” technique. This technique performs similar action to the

SUB function, but is much more flexible and efficient. It is more flexible because a “string slice” may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a “slice” of the STRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the STRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Example:

```
Name        STRING(20)                      !Declare 20 byte name field
ArrayString STRING(5),DIM(20)               !Declare array
Company     STRING('Clarion Software, Inc.')  !The software company - 22 bytes
Phone       STRING(@P(###)###-####P)        !Phone number field - 13 bytes
ExampleFile FILE,DRIVER('Clarion')          !Declare a file
Record        RECORD
NameField       STRING(20),NAME('Name')     !Declare with external name
            . .
  CODE
  NameField = 'Tammi'                     !Assign a value
  NameField[5] = 'y'                      ! change fifth letter
  NameField[5:6] = 'ie'                   ! and change a "slice"
                                          !  -- the fifth and sixth letters
  ArrayString[1] = 'First'                !Assign value to first element
  ArrayString[1,2] = 'u'                  !Change first element 2nd character
  ArrayString[1,2:3] = NameField[5:6]     !Assign slice to slice
```

CSTRING (fixed-length null terminated string)

```
                    |  length  |
label   CSTRING(|string constant|) [,DIM( )] [,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL]
                    |  picture  |        [,STATIC] [,THREAD] [,AUTO]
```

CSTRING A character string.

```
Format: A fixed number of bytes.
Size:   2 to 65,520 bytes in 16-bit, or unlimited in 32-bit.
```

length A numeric constant that defines the number of bytes of storage the string will use. This must include a position for the terminating null character. String variables are not initialized unless given a *string constant*.

string constant A string constant containing the initial value of the string. The length of the string is set to the length of the *string constant* plus the terminating null character.

picture The picture token used to format the values assigned to the string. The length of the string is the number of bytes needed to contain the formatted string and the terminating null character.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

CSTRING declares a character string terminated by a null character (ASCII zero). The memory assigned to the CSTRING is initialized to a zero length string unless the AUTO attribute is present.

CSTRING matches the string data type used in the “C” language and the “ZSTRING” data type of the Btrieve Record Manager. Storage and memory requirements are fixed-length, however the terminating null character is placed at the end of the data entered. CSTRING should be used to achieve compatibility with outside files or procedures.

In addition to its explicit declaration, all CSTRINGs are implicitly declared as a CSTRING(1),DIM(*length of string*). This allows each character in the CSTRING to be addressed as an array element. If the CSTRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a CSTRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the CSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the CSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Since a CSTRING must be null-terminated, the programmer must be responsible for ensuring that an ASCII zero is placed at the end of the data if the field is only accessed through its array elements or as a "slice" (not as a whole entity). Also, a CSTRING can have "junk" stored after the null terminator. Because of this they do not work well inside GROUPs.

Example:

```
Name        CSTRING(21)                  !Declare 21 byte field - 20 bytes data
OtherName   CSTRING(21),OVER(Name)       !Declare field over name field
Contact     CSTRING(21),DIM(4)           !Array 21 byte fields - 80 bytes data
Company     CSTRING('Clarion Software, Inc.')  !23 byte string - 22 bytes data
Phone       CSTRING(@P(###)###-####P)    !   Declare 14 bytes - 13 bytes data
ExampleFile FILE,DRIVER('Btrieve')       !Declare a file
Record        RECORD
NameField       CSTRING(21),NAME('ZstringField')  !Declare with external name
            . .
  CODE
  Name = 'Tammi'                !Assign a value
  Name[5] = 'y'                 ! then change fifth letter
  Name[6] = 's'                 ! then add a letter
  Name[7] = '<0>'               ! and handle null terminator
  Name[5:6] = 'ie'              ! and change a "slice"
                                !  -- the fifth and sixth letters
  Contact[1] = 'First'          !Assign value to first element
  Contact[1,2] = 'u'            !Change first element 2nd character
  Contact[1,2:3] = Name[5:6]    !Assign slice to slice
```

PSTRING (embedded length-byte string)

```
                | length          |
label  PSTRING( |string constant| ) [,DIM( )] [,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC]
                | picture         |          [,THREAD] [,AUTO]
```

PSTRING A character string.

```
Format: A fixed number of bytes.
Size:   2 to 255 bytes.
```

length A numeric constant that defines the number of bytes in the string. This must include the first position length-byte.

string constant A string constant containing the initial value of the string. The length of the string is set to the length of the *string constant* plus the length-byte.

picture The picture token used to format the values assigned to the string. The length of the string is the number of bytes needed to contain the formatted string plus the first position length byte. String variables are not initialized unless given a *string constant.*

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data**.**

AUTO Specify the variable has no *initial value*.

PSTRING declares a character string with a leading length byte included in the number of bytes declared for the string. The memory assigned to the PSTRING is initialized to a zero length string unless the AUTO attribute is present.

PSTRING matches the string data type used by the Pascal language and the “LSTRING” data type of the Btrieve Record Manager. Storage and memory requirements are fixed-length, however, the leading length byte will contain the number of characters actually stored. PSTRING is internally converted

to a STRING intermediate value for use during program execution. It should be used to achieve compatibility with outside files or procedures.

In addition to its explicit declaration, all PSTRINGs are implicitly declared as a PSTRING(1),DIM(*length of string*). This allows each character in the PSTRING to be addressed as an array element. If the PSTRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a PSTRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the PSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the PSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Since a PSTRING must have a leading length byte, the programmer must be responsible for ensuring that its value is always correct if the field is only accessed through its array elements or as a "slice" (not as a whole entity). The PSTRING's length byte is addressed as element zero (0) of the array (the only case in Clarion where an array has a zero element). Therefore, the valid range of array indexes for a PSTRING(30) would be 0 to 29. Also, a PSTRING can have 'junk' stored outside the active portion of the string. Because of this they do not work well inside GROUPs.

Example:

```
Name        PSTRING(21)                     !Declare 21 byte field - 20 bytes data
OtherName   PSTRING(21),OVER(Name)          !Declare field over name field
Contact     PSTRING(21),DIM(4)              !Array 21 byte fields - 80 bytes data
Company     PSTRING('Clarion Software, Inc.')   !23 byte string - 22 bytes data
Phone       PSTRING(@P(###)###-####P)       !Declare 14 bytes - 13 bytes data
ExampleFile FILE,DRIVER('Btrieve')          !Declare a file
Record        RECORD
NameField       PSTRING(21),NAME('LstringField') !Declare with external name
              . .
  CODE
  Name = 'Tammi'                !Assign a value
  Name[5] = 'y'                 ! then change fifth letter
  Name[6] = 's'                 ! then add a letter
  Name[0] = '<6>'               ! and handle length byte
  Name[5:6] = 'ie'              ! and change a "slice" -- the 5th and 6th letters
  Contact[1] = 'First'          !Assign value to first element
  Contact[1,2] = 'u'            !Change first element 2nd character
  Contact[1,2:3] = Name[5:6]    !Assign slice to slice
```

DATE (four-byte date)

label **DATE** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

DATE A four-byte date.

```
Format:     year       mm     dd
          | ........ | .... | .... |
Bits:   31          15      7      0
Range:    year: 1 to 9999
         month: 1 to 12
           day: 1 to 31
```

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

DATE declares a four-byte date variable. This format matches the “DATE” field type used by the Btrieve Record Manager. A DATE used in a numeric expression is converted to the number of days elapsed since December 28, 1800 (Clarion Standard Date - usually stored as a LONG). The valid Clarion Standard Date range is January 1, 1801 through December 31, 2099. Using an out-of-range date produces unpredictable results. DATE fields should be used to achieve compatibility with outside files or procedures.

Example:

```
DueDate     DATE                        !Declare a date field
OtherDate   DATE,OVER(DueDate)          !Declare field over date field
ContactDate DATE,DIM(4)                 !Array of 4 date fields
ExampleFile FILE,DRIVER('Btrieve')      !Declare a file
Record        RECORD
DateRecd        DATE,NAME('DateField')  !Declare with external name
            . .
```

See Also: Standard Date

TIME (four-byte time)

label **TIME** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

TIME A four-byte time.

```
Format:     hh     mm     ss     hs
          | .... | .... | .... | .... |
Bits:   31     23     15     7      0
Range:       hours: 0 to 23
           minutes: 0 to 59
           seconds: 0 to 59
       seconds/100: 0 to 99
```

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, “external” name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable’s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value.*

TIME declares a four byte time variable. This format matches the “TIME” field type used by the Btrieve Record Manager. A TIME used in a numeric expression is converted to the number of hundredths of a second elapsed since midnight (Clarion Standard Time - usually stored as a LONG). TIME fields should be used to achieve compatibility with outside files or procedures.

Example:

```
ChkoutTime  TIME                        !Declare checkout time field
OtherTime   TIME,OVER(CheckoutTime)     !Declare field over time field
ContactTime TIME,DIM(4)                 !Array of 4 time fields
ExampleFile FILE,DRIVER('Btrieve')      !Declare a file
Record        RECORD
TimeRecd        TIME,NAME('TimeField')  !Declare with external name
              . .
```

See Also: Standard Time

GROUP (compound data structure)

```
label   GROUP( [ group ] ) [,PRE( )] [,DIM( )] [,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC]
                           [,THREAD] [,BINDABLE] [, TYPE]
            declarations
        END
```

GROUP	A compound data structure.
group	The label of a previously declared GROUP, QUEUE, or RECORD structure from which it will inherit its structure. This may be a GROUP or QUEUE with the TYPE attribute.
PRE	Declare a label prefix for variables within the structure.
DIM	Dimension the variables into an array.
OVER	Share a memory location with another variable or structure.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
BINDABLE	Specify all variables in the group may be used in dynamic expressions.
TYPE	Specify the GROUP is a type definition for GROUPs passed as parameters.
declarations	Multiple consecutive variable declarations.

A **GROUP** structure allows multiple variable declarations to be referenced by a single label. It may be used to dimension a set of variables, or to assign or compare sets of variables in a single statement. In large complicated programs, a GROUP structure is helpful for keeping sets of related data organized. A GROUP must be terminated by a period or the END statement.

The structure of a GROUP declared with the *group* parameter begins with the same structure as the named *group*; the GROUP inherits the fields of the named *group*. The GROUP may also contain its own *declarations* that follow the inherited fields. If the group parameter names a QUEUE or RECORD structure, only the fields are inherited and not the functionality implied by the QUEUE or RECORD.

When referenced in a statement or expression, a GROUP is treated as a STRING composed of all the variables within the structure. A GROUP structure may be nested within another data structure, such as a RECORD or another GROUP.

Because of their internal storage format, numeric variables (other than DECIMAL) declared in a group do not collate properly when treated as strings. For this reason, building a KEY on a GROUP that contains numeric variables may produce an unexpected collating sequence.

A GROUP with the BINDABLE attribute makes all the variables within the GROUP available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A GROUP with the TYPE attribute is not allocated any memory; it is only a type definition for GROUPs that are passed as parameters to PROCEDUREs or FUNCTIONs. This allows the receiving procedure to directly address component fields in the passed GROUP. The parameter declaration on the PROCEDURE or FUNCTION statement can instantiate a local prefix for the passed GROUP as it names the passed GROUP for ther procedure, however this is not necessary if you use the Field Qualification syntax instead of prefixes. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type definition) to directly address component fields of the GROUP passed as the parameter.

Example:

```
            PROGRAM
PassGroup   GROUP,TYPE                !Type-definition for passed GROUP parameters
F1            STRING(20)              ! first field
F2            STRING(1)               ! middle field
F3            STRING(20)              ! last field
            END

    MAP
      MyProc1(PassGroup)              !Passes a GROUP defined the same as PassGroup
    END

NameGroup   GROUP                     !Name group
First         STRING(20)              ! first name
Middle        STRING(1)               ! middle initial
Last          STRING(20)              ! last name
            END                       !End group declaration

NameGroup2  GROUP(PassGroup)          !Group that inherits PassGroup's fields
                                      ! resulting in NameGroup2:F1, NameGroup2:F2,
                                      ! and NameGroup2:F3
            END                       ! fields declared in this group

DateTimeGrp GROUP,DIM(10)             !Date/time array
Date          LONG
Time          LONG
            END                       !End group declaration

FileNames   GROUP,BINDABLE            !Bindable group
FileName      STRING(8),NAME('FILE')  !Dynamic name: FILE
Dot           STRING('.')             !Dynamic name: Dot
Extension     STRING(3),NAME('EXT')   !Dynamic name: EXT
            END

  CODE
  MyProc1(NameGroup)                  !Call proc passing NameGroup as parameter
  MyProc1(NameGroup2)                 !Call proc passing NameGroup2 as parameter

MyProc1     PROCEDURE(PassedGroup)    !Proc to receive GROUP parameter
LocalVar    STRING(20)
  CODE
  LocalVar = PassedGroup:F1           !Assign value in the first field to LocalVar
                                      !  from passed parameter
```

See Also: Field Qualification

LIKE (inherited data type)

new declaration **LIKE(***like declaration***)** [,**DIM( )**] [,**OVER( )**] [,**PRE( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**BINDABLE**]

LIKE	Declares a variable whose data type is inherited from another variable.
new declaration	The label of the new data element declaration.
like declaration	The label of the data element declaration whose definition will be used.
DIM	Dimension the variables into an array.
OVER	Share a memory location with another variable or structure.
PRE	Declare a label prefix for variables within the *new declaration* structure (if the *like declaration* is a complex data structure). This is not required, since you may use the *new declaration* in the Field Qualification syntax to directly reference any member of the new structure.
NAME	Specify an alternate, “external” name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable’s memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
BINDABLE	Specify all variables in the group may be used in dynamic expressions.

LIKE tells the compiler to define the *new declaration* using the same definition as the *like declaration*, including all attributes. If the original *like declaration* changes, so does the *new declaration*.

The *new declaration* may use the DIM and OVER attributes. If the *like declaration* has a DIM attribute, the *new declaration* is already an array. If a further DIM attribute is added to the *new declaration*, the array is further dimensioned.

The PRE and NAME attributes may be used, if appropriate. If the *like declaration* already has these attributes, the *new declaration* will inherit them and compiler errors can occur. To correct this, specify a PRE or NAME attribute on the *new declaration* to override the inherited attribute.

Example:

```
Amount        REAL                    !Define a field
QTDAmount     LIKE(Amount)            !Use same definition
YTDAmount     LIKE(QTDAmount)         !Use same definition again
MonthlyAmts   LIKE(Amount),DIM(12)    !Use same definition for array, 12 elements
AmtPrPerson   LIKE(MonthlyAmts),DIM(10)
                          !Use same definition for array of 120 elements (12,10)

Construct   GROUP,PRE(Con)            !Define a group
Field1        LIKE(Amount)            ! con:field1 - real
Field2        STRING(10)              ! con:field2 - string(10)
            END

NewGroup    LIKE(Construct)           !Define new group, containing
                                      ! NewGroup:field1 - real
                                      ! NewGroup:field2 - string(10)

AmountFile  FILE,DRIVER('Clarion'),PRE(Amt)
Record        RECORD
Amount          REAL                  !Define a field
QTDAmount       LIKE(Amount)          !Use same definition
            . .
```

See Also: DIM, OVER, PRE, NAME, Field Qualification

Implicit Variables

Implicit variables are not declared in data declarations. They are created by the compiler when it first encounters them. Implicit variables are automatically initialized to blank or zero; they do not have to be explicitly assigned values before use. You may always assume that they contain blanks or zero before your program's first assignment to them.

Any implicit variable used in the global data declaration area (between the keywords PROGRAM and CODE) is Global data, assigned static memory. Any implicit variable used between the keywords MEMBER and PROCEDURE (or FUNCTION) is Module data, assigned static memory. Any other implicit variable is Local data, assigned dynamic memory on the program's stack.

Since the compiler dynamically creates implicit variables as they are encountered, there is a danger that problems may arise that can be difficult to trace. This is due to the lack of compile-time error and type checking on implicit variables. For example, if you spell incorrectly the name of a previously used implicit variable, the compiler will not tell you, but will simply create a new implicit variable with the new spelling. When your program checks the value in the original implicit variable, it will be incorrect. Therefore, implicit variables should be used with care and caution, and only within a limited scope (or not at all).

Implicit variables are generally used for: array subscripts, true/false switches, intermediate variables in complex calculations, loop control variables, etc. The Clarion language provides three types of implicit variables:

#	Pound sign names an implicit LONG variable, a label terminated by a # character.
$	Dollar sign names an implicit REAL variable, a label terminated by a $ character.
“	Double quote names an implicit 32 byte string, a label terminated by a “ character.

Example:

```
LOOP Counter# = 1 TO 10                      !Implicit LONG loop counter
  ArrayField[Counter#] = Counter# * 2        ! to initialize an array
END

Address" = CLIP(City) & ', ' & State & '  ' & Zip     !Implicit STRING(32)
MESSAGE(Address")                            !Used to display a temporary value

Percent$ = ROUND((Quota / Sales),.1) * 100   !Implicit REAL
MESSAGE(FORMAT(Percent$,@P%<<<.##P))         !Used to display a temporary value
```

See Also: Data Declarations and Memory Allocation

Reference Variables

A reference variable contains a reference to another data declaration (its "target"). You declare a reference variable by prepending an ampersand (&) to the data type of its target (&BYTE, &FILE, &LONG, &WINDOW, etc.). Depending upon the target's data type, the reference variable may contain the target's memory address, or a more complex internal data structure (describing the location and type of target data).

Valid reference variable declarations are: &BYTE, &SHORT, &USHORT, &LONG, &ULONG, &REAL, &SREAL, &BFLOAT8, &BFLOAT4, &DECIMAL, &PDECIMAL, &STRING, &CSTRING, &PSTRING, &QUEUE, &FILE, &BLOB, &VIEW, and &WINDOW. Reference variables may not be declared within GROUP, FILE, QUEUE, or VIEW structures.

The &STRING, &CSTRING, &PSTRING, &DECIMAL, and &PDECIMAL reference variable declarations do not require length parameters, since all necessary information about the specific target data item is contained in the reference. This means a &STRING reference variable may contain a reference to any length STRING variable. A reference variable declared with &WINDOW can target either an APPLICATION, WINDOW, or REPORT structure.

The label of the reference variable is syntactically correct every place in executable code where its target is allowed. When used in a code statement, the reference variable is automatically "dereferenced" to supply the statement the value of its target (except for reference assignment statements). References cross thread boundaries, and so, may be used to reference data items in other execution threads.

The &= operator executes a reference assignment statement (destination &= source). This assigns the source's reference to the destination reference variable.

Example:

```
App1   APPLICATION('Hello')
       END
App2   APPLICATION('Buenos Dias')
       END
AppRef &WINDOW                    !Reference to an APPLICATION, WINDOW, or REPORT
  CODE
  IF CTL:Language = 'English'     !If english language user
    AppRef &= App1                ! reference english application frame
  ELSE
    AppRef &= App2                ! else reference spanish application frame
  END
  OPEN(AppRef)                    !Open the referenced application frame window
```

See Also: Reference Assignment Statements, THREAD

Attributes of Variables

PRE (set group label prefix)

PRE([*prefix*] **)**

PRE Provides a label prefix for complex data structures.

prefix Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alphabet character and must not be a reserved word. By convention, a *prefix* is 1-3 characters, although it can be longer.

The **PRE** attribute provides a label prefix for complex data structures. It is used to distinguish between identical variable names that occur in different structures. When referenced in executable statements, assignments, and parameter lists, a *prefix* is attached to a label by a colon (Pre:Label). The PRE attribute may be used with the following data structures discussed in this chapter: GROUP, and LIKE.

Another more flexible method to distinguish between identical variable names that occur in different structures does not use the PRE attribute, but instead uses the Field Qualification syntax. When referenced in executable statements, assignments, and parameter lists, the label of the structure containing the field is attached to the field label by a colon (GroupName:Label).

Example:

```
G1      GROUP,PRE(Mem)    !Declare some memory variables
Message   STRING(30)      ! with the Mem prefix
Page      LONG
Line      LONG
Device    STRING(30)
        END

G2      LIKE(G1),PRE(Me2)   !Declare second GROUP LIKE the first
                            !Contains same variables with Me2 prefix
  CODE
  Mem:Message = 'Variable in original group'      !Using prefix
  G1:Message = 'Same Variable in original group' !Using Field Qualification
  Me2:Message = 'Variable in LIKE group'
  G2:Message = 'Same Variable in LIKE group'
```

See Also: Reserved Words, Field Qualification

DIM (set array dimensions)

DIM(*dimension,...,dimension***)**

DIM	Declares a variable as an array.
dimension	A numeric constant which specifies the number of elements in this *dimension* of the array.

The **DIM** attribute declares a variable as an array. The variable is repeated the number of times specified by the *dimension* parameters. Multi-dimensional arrays may be thought of as nested. Each *dimension* in the array has a corresponding subscript. Therefore, referencing a variable in a three dimensional array requires three subscripts. There is no limit to the number of dimensions; however, the total size of an array must not exceed 65,520 bytes of data in 16-bit applications (there is no limit in 32-bit applications).

Subscripts identify which element of the array is being referenced. A subscript list contains a subscript for each *dimension* of the array. Each subscript is separated by a comma and the entire list is enclosed in brackets ([ ]). A subscript may be a numeric constant, expression, or function. The entire array may be referenced by the label of the array without a subscript list.

A GROUP structure is a special case. Each level of nesting adds subscripts to the GROUP and the variables within the GROUP. Data declared within the GROUP may be referenced exactly like the GROUP itself.

Example:

```
Scr     GROUP                !Characters on a text-mode screen
Row       GROUP,DIM(25)      !Twenty-five rows
Pos         GROUP,DIM(80)    !Two thousand positions
Attr          BYTE           !Attribute byte
Char          BYTE           !Character byte
          . . .              !Terminate the group structures
  ! In the group above:
  ! Scr is a 4,000 byte GROUP
  ! Row[1] is a 160 byte GROUP
  ! Pos[1,1] is a 2 byte GROUP
  ! Attr[1,1] is a BYTE
  ! Char[1,1] is a BYTE

Month   STRING(10),DIM(12)   !Dimension the month to 12
  CODE
  CLEAR(Month)               !Assign blanks to the entire array
  Month[1] = 'January'       !Load the months into the array
  Month[2] = 'February'
  Month[3] = 'March'
```

See Also: MAXIMUM

EXTERNAL (set variable defined externally)

EXTERNAL

The **EXTERNAL** attribute specifies that the variable on which it is placed is defined in an external library. Therefore, a variable with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory. The memory for the variable is allocated by the external library. This allows the Clarion program access to variables declared as public in external libraries.

The EXTERNAL attribute is valid only on variables declared outside FILE, QUEUE, or GROUP structures.

The variable declarations in all libraries (or .EXEs) that reference common variables must be EXACTLY the same (with the appropriate addition of the EXTERNAL attribute). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

When using EXTERNAL to declare a variable shared by multiple libraries (.OBJs, .LIBs, or .DLLs and .EXE), only one library should define the variable without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the variable with the EXTERNAL attribute. This ensures that there is only one memory allocation for the variable and all the libraries and the .EXE will reference the same memory when referring to that variable.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same variables would have one .DLL containing the actual data definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
TotalCount  LONG,EXTERNAL        !A variable declared in an external library
```

See Also: NAME, DLL

DLL (set variable defined externally in .DLL)

DLL([*flag*] **)**

DLL	Declares a variable defined externally in a .DLL.
flag	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the variable on which it is placed is defined in a .DLL. A variable with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the variable. The DLL attribute is valid only on variables declared outside FILE, QUEUE, or GROUP structures.

The variable declarations in all libraries (or .EXEs) that reference common variables must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL to declare a variable shared by .DLLs and .EXE, only one .DLL should define the variable without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the variable with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the variable and all the .DLLs and the .EXE will reference the same memory when referring to that variable.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same variables would have one .DLL containing the actual data definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
TotalCount  LONG,EXTERNAL,DLL        !A variable declared in an external .DLL
```

See Also: EXTERNAL

NAME (set variable's external name)

NAME([| *constant* | *variable* |]**)**

NAME	Specifies an "external" name for the linker or file driver.
constant	A string constant.
variable	The label of a STRING variable declared in the global data declaration area or a MEMBER module's data declaration area.

The **NAME** attribute specifies an "external" name for the linker or file driver. The NAME attribute is completely independent of the EXTERNAL attribute—there is no required connection between the two, although both attributes may be used on the same variable.

The NAME attribute may be placed on a FUNCTION or PROCEDURE Prototype, FILE, KEY, INDEX, MEMO, any field declared within a FILE, any field declared within a QUEUE structure, or any field not within a structure. The NAME attribute has different implications depending on where it is used.

NAME(*constant*) may be specified on a FUNCTION or PROCEDURE Prototype. The *constant* supplies the external name used by the linker to identify the procedure or function from an external library.

The NAME(*constant*) or NAME(*variable*) attribute on a FILE declaration specifies a DOS directory file specification. If the *constant* or *variable* does not contain a drive and path, the current drive and directory are assumed. If the extension is omitted, the directory entry assumes the file driver's default value. Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files. Therefore, a NAME may also be placed on a KEY, INDEX, or MEMO. A NAME attribute without a *constant* or *variable* defaults to the label of the declaration statement on which it is placed (including any specified prefix).

NAME(*constant*) may be used on any field declared within the RECORD structure. This provides the file driver with the name of a field as it may be used in that driver's file system.

NAME(*constant*) may be used on any field declared within a QUEUE structure. This provides the capability of run time dynamic sorts.

NAME(*constant*) may be used on any variable declared outside of any structure. This provides the linker with an external name to identify a variable declared in an external library. If the variable also has the EXTERNAL attribute, it is declared, and its memory is allocated, as a public variable in the external library. Without the EXTERNAL attribute, it is declared, and its memory is allocated, in the Clarion program, and it is

declared as an external variable in the external library.

Example:

```
  PROGRAM
  MAP
   MODULE('External.Obj')
    AddCount(LONG),LONG,C,NAME('_AddCount')   !C function named '_AddCount'
  . .

Cust    FILE,PRE(Cus),NAME(CustName)            !Filename in CustName variable
CustKey   KEY('Name'),NAME('c:\data\cust.idx')  !Declare key, cust.idx
Record    RECORD
Name        STRING(20)                          !Default NAME to 'Cus:Name'
        . .

SortQue QUEUE,PRE(Que)
Field1    STRING(10),NAME('FirstField')     !QUEUE SORT NAME
Field2    LONG,NAME('SecondField')          !QUEUE SORT NAME
        END

CurrentCnt  LONG,EXTERNAL,NAME('Cur')       !Field declared public in
                                            ! external library as 'Cur'
TotalCnt    LONG,NAME('Tot')                !Field declared external
                                            ! in external library as 'Tot'
```

See Also: FUNCTION and PROCEDURE Prototypes, FILE, KEY, INDEX, QUEUE, EXTERNAL

OVER (set shared memory location)

OVER(*overvariable***)**

OVER Allows one memory address to be referenced two different ways.

overvariable The label of a variable that already occupies the memory to be shared.

The **OVER** attribute allows one memory address to be referenced two different ways. The variable declared with the OVER attribute must not be larger than the *overvariable* it is being declared OVER (it may be smaller, though).

You may declare a variable OVER an *overvariable* which is part of the parameter list passed into a PROCEDURE or FUNCTION.

A field within a GROUP structure cannot be declared OVER a *variable* outside that GROUP structure.

Example:

```
SomeProc  PROCEDURE(PassedGroup)       !Proc receives a GROUP parameter

NewGroup  GROUP,OVER(PassedGroup)      !Redeclare passed GROUP parameter
Field1      STRING(10)                 !Compiler warning issued that
Field2      STRING(2)                  ! NewGroup must not be larger
          END                          ! than PassedGroup

CustNote FILE,PRE(Csn)                 !Declare CustNote file
Notes    MEMO(2000)                    !The memo field
Record    RECORD
CustID      LONG
         . .

CsnMemoRow  STRING(10),DIM(200),OVER(Csn:Notes)
                                       !Csn:Notes memo may be addressed
                                       ! as a whole or in 10-byte chunks
```

See Also: DIM

STATIC (set local variable static)

STATIC

The **STATIC** attribute allows a variable declared within a PROCEDURE or FUNCTION to be allocated static memory instead of stack memory. This makes any value contained in the variable "persistent" from one instance of the procedure to the next.

Example:

```
SomeProc    PROCEDURE
AcctFile    STRING(64),STATIC       !STATIC needed for use as
                                    ! Variable in NAME attribute

Transactions  FILE,DRIVER('Clarion'),PRE(TRA),NAME(AcctFile)
AccountKey      KEY(TRA:Account),OPT,DUP
Record          RECORD
Account           SHORT             !Account code
Date              LONG              !Transaction Date
Amount            DECIMAL(13,2)     !Transaction Amount
                . .
```

See Also: Data Declarations and Memory Allocation

THREAD (set thread-specific static variable)

THREAD

The **THREAD** attribute declares a static variable which is allocated memory separately for each execution thread in the program. This makes the value contained in the variable dependent upon which thread is executing. Whenever a new execution thread is begun, a new instance of the variable, specific to that thread, is created.

The variable must be allocated static memory so it should be declared as Local data with the STATIC attribute. It may also be declared as Global data or Module data.

This attribute creates runtime "overhead," particularly on Global or Module data. Therefore, it should be used only when absolutely necessary.

Example:

```
GlobalVar  LONG,THREAD     !Each execution thread gets its own copy

SomeProc   PROCEDURE
LocalVar   LONG,THREAD     !Local threaded variable (automatically STATIC)
```

See Also: START, Data Declarations and Memory Allocation, STATIC

BINDABLE (set dynamic expression string variables)

BINDABLE

The **BINDABLE** attribute declares a GROUP, QUEUE, FILE, or VIEW whose constituent variables are all available for use in a runtime expression string. The contents of each variable's NAME attribute is the logical name used in the runtime expression string. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

Example:

```
FileNames GROUP,BINDABLE              !Bindable group
FileName    STRING(8),NAME('FILE')    !Dynamic name: FILE
Dot         STRING('.')               !Dynamic name: Dot
Extension   STRING(3),NAME('EXT')     !Dynamic name: EXT
          END       !
```

See Also: BIND, UNBIND, EVALUATE

AUTO (uninitialized local variable)

AUTO

The **AUTO** attribute allows a variable, declared within a PROCEDURE or FUNCTION, to be allocated uninitialized stack memory. Without the AUTO attribute, a numeric variable is initialized to zero and a string variable is initialized to all blanks when its memory is assigned at run-time.

The AUTO attribute is used when you do not need to rely on an initial blank or zero value because you intend to assign some other value to the variable. This saves a small amount of run-time memory by eliminating the internal code necessary to perform the automatic initialization for the variable.

Example:

```
SomeProc    PROCEDURE
SaveCustID  LONG,AUTO       !Non-initialized local variable
```

TYPE (GROUP type definition)

TYPE

The **TYPE** attribute creates a "type definition" for a GROUP. The type definition can then be used in a LIKE statement to define other similar GROUPs. A GROUP with the TYPE attribute is not allocated any memory.

Example:

```
PassGroup   GROUP,TYPE                !Type-definition for passed GROUP parameters
F1            STRING(20)              ! first field
F2            STRING(1)               ! middle field
F3            STRING(20)              ! last field
            END

NameGroup   LIKE(PassGroup),PRE(Nme)  !Name group
```

Data Declarations and Memory Allocation

Global, Local, Static, and Dynamic

Data declarations allocate memory to store the data values. Global, Local, Static, and Dynamic are terms that describe types of memory allocation.

The terms “Global” and “Local” refer to the “visibility” of data:

- “Global” means the data is visible and available to all procedures in the program.
- “Local” means the data has limited visibility. This may be limited to one procedure or function, or limited to a specific set of procedures and/or functions.

The terms “Static” and “Dynamic” refer to the persistence of the data’s memory allocation:

- “Static” means the data is allocated memory that is not released until the entire program is finished executing.
- “Dynamic” means the data is allocated memory on the program’s stack. Stack memory is released when the PROCEDURE or FUNCTION that allocated the stack memory returns to the place in the program from which it was called.

Data Declaration Sections

There are three areas where data can be declared in a Clarion program:

- In the PROGRAM module, after the keyword PROGRAM and before the CODE statement. This is the **Global data** section.
- In a MEMBER module, after the keyword MEMBER and before the first PROCEDURE or FUNCTION statement. This is the **Module data** section.
- In a PROCEDURE or FUNCTION, after the keyword PROCEDURE (or FUNCTION) and before the CODE statement. This is the **Local data** section.

Global data is visible to executable statements and expressions in every PROCEDURE and FUNCTION in the PROGRAM. Global data is allocated in Static memory.

Module data is visible only to the set of PROCEDUREs and FUNCTIONs contained in the MEMBER module. Of course, it may be passed as a parameter to PROCEDUREs or FUNCTIONs in other MEMBER modules, if required. Module data is also allocated Static memory.

Local data is visible only within the PROCEDURE or FUNCTION in which it is declared. Of course, it may be passed as a parameter to any other PROCEDURE or FUNCTION. Local data is allocated Dynamic memory. Thew memory is allocated on the program's stack for variables smaller than the stack threshold (5K default), otherwise they are automatically placed onto the heap. This can be overridden by using the STATIC attribute, making its value persistent between calls to the procedure.

Dynamic memory allocation for Local data allows a FUNCTION or PROCEDURE to be truly recursive, receiving a new copy of its local variables each time it is called.

See Also: FUNCTION and PROCEDURE Prototypes, STATIC

Picture Tokens

Picture tokens provide a masking format for displaying and editing variables. Picture tokens may be used as parameters of STRING, ENTRY, or STRING OPTION declarations in SCREEN structures; as a parameter of STRING statements in a REPORT structure; as a parameter of some Clarion procedures and functions; or, the parameter of STRING, CSTRING and PSTRING variable declarations.

There are seven types of picture tokens: numeric and currency, scientific notation, date, time, pattern, key-in template, and string.

Numeric and Currency Pictures

@**N** [*currency*] [*sign*] [*fill*] *size* [*grouping*] [*places*] [*sign*] [*currency*] [**B**]

@N	All numeric and currency pictures begin with @N.
currency	Either a dollar sign ($) or a string constant enclosed in tildes (~). When it precedes the *sign* indicator and there is no *fill* indicator, the *currency* symbol "floats" to the left of the high order digit. If there is a *fill* indicator, the *currency* symbol remains fixed in the left-most position. If the *currency* indicator follows the *size* and *grouping*, it appears at the end of the number displayed.
sign	Specifies the display format for negative numbers. If a hyphen precedes the *fill* and *size* indicators, negative numbers will display with a leading minus sign. If a hyphen follows the *size*, *grouping*, *places*, and *currency* indicators, negative numbers will display with a trailing minus sign. If parentheses are placed in both positions, negative numbers will be displayed enclosed in parentheses. To prevent ambiguity, a trailing minus *sign* should always have *grouping* specified.
fill	Specifies leading zeros, spaces, or asterisks (*) in any leading zero positions, and suppresses *grouping*. If the *fill* indicator is omitted, leading zeros are suppressed. **0** (zero) Produces leading zeroes **_** (underscore) Produces leading spaces ***** (asterisk) Produces leading asterisks
size	The *size* is required to specify the total number of significant digits to display, including the number of digits in the *places* indicator and any formatting characters.

grouping A *grouping* symbol, other than a comma (the default), can be placed to the right of the *size* indicator to specify a three digit group separator.To prevent ambiguity, a hyphen *grouping* indicator should always have the *sign* specified.

. (period) Produces periods
- (hyphen) Produces hyphens
_ (underscore) Produces spaces

places Specifies the decimal separator symbol and the number of decimal digits. The number of decimal digits must be less than the size indicator. The decimal separator may be a period (.), grave accent (‘ -- produces periods for *grouping* separators, unless overridden), or the letter “v” (used only for STRING field storage declarations—not for display).

. (period) Produces a period
‘ (grave accent) Produces a comma
v Produces no decimal separator

B Specifies that the format displays as blank whenever its value is zero.

The numeric and currency pictures format numeric values for screen display or in reports. If the value is greater than the maximum value the picture can display, a string of asterisks is displayed.

Example:

```
Numeric            Result    Format
@N9              4,550,000    Nine digits, group with commas (default)
@N_9B              4550000    Nine digits, no grouping, leading blanks if zero
@N09             004550000    Nine digits, leading zero
@N*9             ***45,000    Nine digits, asterisk fill, group with commas
@N9_             4 550 000    Nine digits, group with spaces
@N9.             4.550.000    Nine digits, group with periods

Decimal            Result    Format
@N9.2             4,550.75    Two decimal places, period decimal separator
@N_9.2B            4550.75    Two decimal places, period decimal separator, no
                              grouping, blank if zero
@N_9‘2             4550,75    Two decimal places, comma decimal separator
@N9.‘2            4.550,75    Comma decimal separator, group with periods
@N9_‘2            4 550,75    Comma decimal separator, group with spaces,

Signed             Result    Format
@N-9.2B          -2,347.25    Leading minus sign, blank if zero
@N9.2-           2,347.25-    Trailing minus sign
@N(10.2)       (2,347.25)     Enclosed in parens when negative

Dollar Currency    Result    Format
@N$9.2B          $2,347.25    Leading dollar sign, blank if zero
@N$10.2-       $2,347.25-     Leading dollar sign, trailing minus when negative
@N$(11.2)     $(2,347.25)     Leading dollar sign, in parens when negative

Int’l Currency              Result    Format
@N12_‘2~ F~            1 5430,50 F    France
@N~L. ~12‘            L. 1.430.050    Italy
@N~£~12.2                £1,240.50    United Kingdom
@N~kr~12‘2              kr1.430,50    Norway
@N~DM~12‘2              DM1.430,50    Germany
@N12_‘2~ mk~           1 430,50 mk    Finland
@N12‘2~ kr~            1.430,50 kr    Sweden

Storage-Only Pictures:
Variable1 STRING(@N_6v2)              !Declare as 6 bytes stored without decimal
  CODE
  Variable1 = 1234.56                 !Assign value, stores ‘123456’ in file
  MESSAGE(FORMAT(Variable1,@N_7.2))   !Display with decimal point: ‘1234.56’
```

Scientific Notation Pictures

@Em.n[B]

@E	All scientific notation pictures begin with @E.
m	Determines the total number of characters in the format provided by the picture.
n	Indicates the number of digits that appear to the left of the decimal point.
B	Specifies that the format displays as blank when the value is zero.

The scientific notation picture formats very large or very small numbers. The format is a decimal number raised by a power of ten.

Example:

Picture	Value	Result
@E9.0	1,967,865	.20e+007
@E12.1	1,967,865	1.9679e+006
@E12.1B	0	
@E12.1	-1,967,865	-1.9679e+006
@E12.1	.000000032	3.2000e-008

Date Pictures

@Dn[s][B]

@D All date pictures begin with @D.

n Determines the date picture format. Date picture formats range from 1 through 18. A leading zero (0) indicates a zero-filled day or month.

s A separation character between the month, day, and year components. If omitted, the slash (/) characters appears.

. (period) Produces periods
‘ (grave accent) Produces commas
- (hyphen) Produces hyphens
_ (underscore) Produces spaces

B Specifies that the format displays as blank when the value is zero.

Dates may be stored in numeric variables (usually LONG), a DATE field (for Btrieve compatibility), or in a STRING declared with a date picture. A date stored in a numeric variable is called a “Clarion Standard Date.” The stored value is the number of days since December 28, 1800. The date picture token converts the value into one of the 16 date formats.

Example:

```
Picture            Format          Result
@D1                mm/dd/yy        10/31/59
@D01               mm/dd/yy        01/01/95
@D2              mm/dd/yyyy        10/31/1959
@D3            mmm dd, yyyy        OCT 31,1959
@D4   mmmmmmmmm dd, yyyy          October 31, 1959
@D5                dd/mm/yy        31/10/59
@D6              dd/mm/yyyy        31/10/1959
@D7               dd mmm yy        31 OCT 59
@D8             dd mmm yyyy        31 OCT 1959
@D9                yy/mm/dd        59/10/31
@D10             yyyy/mm/dd        1959/10/31
@D11                yymmdd         591031
@D12              yyyymmdd         19591031
@D13                 mm/yy         10/59
@D14                mm/yyyy        10/1959
@D15                  yy/mm        59/10
@D16                yyyy/mm        1959/10
@D17                               Windows Control Panel setting for Short Date
@D18                               Windows Control Panel setting for Long Date

    Alternate separators
@D1.               mm.dd.yy        Period separator
@D2-             mm-dd-yyyy        Dash separator
@D5_               dd mm yy        Underscore produces space separator
@D6‘              dd,mm,yyyy       Grave accent produces comma separator
```

See Also: Standard Date

Time Pictures

@Tn[s][B]

@T All time pictures begin with @T.

n Determines the time picture format. Time picture formats range from 1 through 8. A leading zero (0) indicates zero-filled hours.

s A separation character. By default, colon (:) characters appear between the hour, minute, and second components of certain time picture formats. The following s indicators provide an alternate separation character for these formats.

. (period) Produces periods
‘ (grave accent) Produces commas
- (hyphen) Produces hyphens
_ (underscore) Produces spaces

B Specifies that the format displays as blank when the value is zero.

Times may be stored in a numeric variable (usually a LONG), a TIME field (for Btrieve compatibility), or in a STRING declared with a time picture. A time stored in a numeric variable is called a “Standard Time.” The stored value is the number of hundredths of a second since midnight. The picture token converts the value to one of the six time formats.

Example:

```
Picture          Format          Result
@T1                hh:mm         17:30
@T2                  hhmm        1730
@T3              hh:mmXM         5:30PM
@T03             hh:mmXM         05:30PM
@T4            hh:mm:ss          17:30:00
@T5              hhmmss          173000
@T6          hh:mm:ssXM          5:30:00PM
@T7                              Windows Control Panel setting for Short Time
@T8                              Windows Control Panel setting for Long Time

    Alternate separators
@T1.               hh.mm         Period separator
@T1-               hh-mm         Dash separator
@T3_             hh mmXM         Underscore produces space separator
@T4‘            hh,mm,ss         Grave accent produces comma separator
```

See Also: Standard Time

Pattern Pictures

@P[<][#][x]P[B]

@P	All pattern pictures begin with the @P delimiter and end with the P delimiter. The case of the delimiters must be the same.
<	Specifies an integer position that is blank when zero.
#	Specifies an integer position.
x	Represents optional display characters. These characters appear in the final result string.
P	All pattern pictures must end with P. If a lower case @p delimiter is used, the ending P delimiter must also be lower case.
B	Specifies that the format displays as blank when the value is zero.

Pattern pictures contain optional integer positions and optional edit characters. Any character other than < or # is considered an edit character which will appear in the formatted picture string. The @P and P delimiters are case sensitive. Therefore, an upper case “P” can be included as an edit character if the delimiters are both lower case “p” and vice versa.

Pattern pictures do not recognize decimal points, in order to permit the period to be used as an edit character. Therefore, the value formatted by a pattern picture should be an integer. If a floating piont value is formatted by a pattern picture, only the integer portion of the number will appear in the result.

Example:

```
Picture                      Value          Result
@P###-##-####P             215846377     215-84-6377
@P<#/##/##P                   103159        10/31/59
@P(###)###-####P           3057854555  (305)785-4555
@P###/###-####P               7854555   000/785-4555
@p<#:##PMp                        530         5:30PM
@P<#' <#"P                        506          5' 6"
@P<#lb. <#oz.P                    902      9lb. 2oz.
@P4##A-#P                         112         411A-2
@PA##.C#P                      312.45         A31.C2
```

Key-in Template Pictures

@K[@][#][<][x][\][?][^][_][|]K[B]

@K	All key-in template pictures begin with the @K delimiter and end with the K delimiter. The case of the delimiters must be the same.
@	Specifies only uppercase and lowercase alphabetic characters.
#	Specifies an integer 0 through 9.
<	Specifies an integer that is blank for high order zeros.
x	Represents optional constant display characters (any displayable character). These characters appear in the final result string.
\	Indicates the following character is a display character. This allows you to include any of the picture formatting characters (@,#,<,\,?,^,_,|) within the string as a display character.
?	Specifies any character may be placed in this position.
^	Specifies only uppercase alphabetic characters in this position.
_	Underscore specifies only lowercase alphabetic characters in this position.
|	Allows the operator to "stop here" if there are no more characters to input. Only the data entered and any display characters up to that point will be in the string result.
K	All key-in template pictures must end with K. If a lower case @k delimiter is used, the ending K delimiter must also be lower case.
B	Specifies that the format displays as blank when the value is zero.

Key-in pictures may contain integer positions (# <), alphabet character positions (@ ^ _), any character positions (?), and display characters. Any character other than a formatting indicator is considered a display character, which appears in the formatted picture string. The @K and K delimiters are case sensitive. Therefore, an upper case "K" may be included as a display character if the delimiters are both lower case "k" and vice versa.

Key-in pictures are used specifically with STRING, PSTRING, and CSTRING fields to allow custom field editing control and validation. Using a key-in picture containing any of the alphabet indicators (@ ^ _) on a numeric entry field produces unpredictable results.

Using the Insert typing mode for a key-in picture could produce unpredictable results. Therefore, key-in pictures always receive data entry in Overwrite mode, even if the INS attribute is present.

Example:

```
Picture                        Value Entered       Result String
@K###-##-####K                     215846377         215-84-6377
@K######|-####K                         33064               33064
@K######|-####K                    330643597          33064-3597
@K<# ^^^ ##K                         10AUG59           10 AUG 59
@K(###)@@@-##\@##K                305abc4555       (305)abc-45@55
@K###/?##-####K                      7854555        000/785-4555
@k<#:##^Mk                             530P              5:30PM
@K<#' <#"K                              506               5' 6"
@K4#_#A-#K                             1g12             41g1A-2
```

String Pictures

*@**S**length*

@S	All string pictures begin with @S.
length	Determines the number of characters in the picture format.

A string picture describes an unformatted string of a specific *length*.

Example:

```
Name  STRING(@S20)    !A 20 character string field
```

Compiler Directives

EQUATE (assign label)

```
                | label    |
label  EQUATE(  | constant | )
                | picture  |
                | type     |
```

EQUATE Assigns a label to another label or constant.

label The *label* of any statement preceding the EQUATE statement. This is used to declare an alternate statement label.

constant A numeric or string *constant*. This is used to declare a shorthand label for a constant value. It also makes a constant easy to locate and change.

picture A *picture* token. This is used to declare a shorthand label for a picture token. However, the screen and report formatter in the Clarion Editor will not recognize the equated label as a valid picture.

type A data type. This is usually used to declare a single method of declaring a variable as one of several data types. depending upon compiler settings (like a C++ typedef for a simple data type).

The **EQUATE** directive assigns a label to another label or constant. It does not use any run-time memory. The label of an EQUATE directive cannot be the same as its parameter.

Example:

```
Init      EQUATE(SetUpProg)                !Set alias label
Off       EQUATE(0)                        !Off means zero
On        EQUATE(1)                        !On means one
PI        EQUATE(3.1415927)                !The value of PI
EnterMsg  EQUATE('Press Ctrl-Enter to SAVE')
SocSecPic EQUATE(@P###-##-####P)           !Soc-sec number picture

  OMIT('End16BitChk',Flag32Bit = 0)        !OMIT if 32-bit compile is turned off
SIGNED    EQUATE(LONG)                     !SIGNED = LONG in a 32-bit compile
End16BitChk
  OMIT('End32BitChk',Flag32Bit = 1)        !OMIT if 32-bit compile is turned on
SIGNED    EQUATE(SHORT)                    !SIGNED = SHORT in a 16-bit compile
End32BitChk
```

See Also: Reserved Words

SIZE (memory size in bytes)

```
        | variable |
SIZE(   | constant | )
        | picture  |
```

SIZE Supplies the amount of memory used for storage.

variable The label of a previously declared variable.

constant A numeric or string constant.

picture A picture token.

SIZE directs the compiler to supply the amount of memory (in bytes) used to store the *variable*, *constant*, or *picture*.

Example:

```
SavRec    STRING(1),DIM(SIZE(Cus:Record)
                                    !Dimension the string to size of record

StringVar STRING(SIZE('Clarion Software, Inc.'))
                                    !A string long enough for the constant

LOOP I# = 1 TO SIZE(ParseString)    !Loop for number of bytes in the string

PicLen = SIZE(@P(###)###-####P)     !Save size of the picture
```

Expressions

Contents

An expression is a mathematical, string, or logical formula that produces a value. An expression may be the source variable of an assignment statement, a parameter of a procedure or function, a subscript of an array (a dimensioned variable), or the condition of an IF, CASE, LOOP, or EXECUTE structure. Expressions may contain constant values, variables, and function calls connected by logical and/or arithmetic or string operators.

Expression Evaluation

Expressions are evaluated in the standard algebraic order of operations. The precedence of operations is controlled by operator type and placement of parentheses. Each operation produces an (internal) intermediate value used in subsequent operations. Parentheses may be used to group operations within expressions. Expressions are evaluated beginning with the inner-most set of parentheses and working through to the outer-most set.

Precedence levels for expression evaluation, from highest to lowest, are:

Level 1	( )	Parenthetical Grouping
Level 2	-	Unary Minus (Negative sign)
Level 3	function call	Gets the RETURN value
Level 4	^	Exponentiation
Level 5	* / %	Multiplication, Division, Modulus Division
Level 6	+ -	Addition, Subtraction
Level 7	&	Concatenation
Level 8	= <>	Logical Comparisons
Level 9	AND, NOT, OR	Boolean expressions

Expressions may produce numeric values, string values, or logical values (true/false evaluation). An expression may contain no operators at all; it may be a single variable, constant value, or function call.

Arithmetic Operators

An arithmetic operator combines two operands arithmetically to produce an intermediate value. The operators are:

+	Addition (A + B gives the sum of A and B)
-	Subtraction (A - B gives the difference of A and B)
*	Multiplication (A * B multiples A by B)
/	Division (A / B gives divides A by B)
^	Exponentiation (A ^ B gives A raised to power of B)
%	Modulus Division (A % B gives the remainder of A divided by B)

Logical Operators

A logical operator compares two operands or expressions and produces a true or false condition. There are two types of logical operators: conditional and Boolean. Conditional operators compare two values or expressions. Boolean operators connect string, numeric, or logical expressions together to determine true-false logic. Operators may be combined to create complex operators.

Conditional Operators	=	Equal sign
	<	Less than
	>	Greater than
Boolean Operators	**NOT**	Boolean NOT
	~	Tilde (Logical NOT)
	AND	Boolean AND
	OR	Boolean OR
	XOR	Boolean XOR (eXclusive OR)
Combined operators	<>	Not equal
	~=	Not equal
	NOT =	Not equal
	<=	Less than or equal to
	=<	Less than or equal to
	~>	Not greater than
	NOT >	Not greater than
	>=	Greater than or equal to
	=>	Greater than or equal to
	~<	Not less than
	NOT <	Not less than

During logical evaluation, any non-zero value indicates a true condition, and a null (blank) string or zero value indicates a false condition.

Example:

```
Logical Expression              Result
    A = B                       True when A is equal to B
    A < B                       True when A is less than B
    A > B                       True when A is greater than B
    A <> B, A ~= B, A NOT = B   True when A is not equal to B
    A ~< B, A >= B, A NOT < B   True when A is not less than B
    A ~> B, A <= B, A NOT > B   True when A is not greater than B
    ~ A, NOT A                  True when A is null or zero
    A AND B                     True when A is true and B is true
    A OR B                      True when A is true, or B is true, or both are true
    A XOR B                     True when A is true or B is true, but not both.
```

Numeric Constants

Numeric constants are fixed numeric values. They may occur in data declarations, in expressions, and as parameters of procedures, functions, or attributes. A numeric constant may be represented in decimal (base 10—the default), binary (base 2), octal (base 8), hexadecimal (base 16), or scientific notation formats. Formatting characters, such as dollar signs and commas, are not permitted in numeric constants.

Decimal (base ten) numeric constants may contain an optional leading minus sign (hyphen character), an integer, and an optional decimal with a fractional component. Binary (base two) numeric constants may contain an optional leading minus sign, the digits 0 and 1, and a terminating B or b character. Octal (base eight) numeric constants contain an optional leading minus sign, the digits 0 through 7, and a terminating O or o character. Hexadecimal (base sixteen) numeric constants contain an optional leading minus sign, the digits 0 through 9, alphabet characters A through F (representing the numbers 10 through 15) and a terminating H or h character. If the left-most character is a letter A through F, a leading zero must be used.

Example:

```
      -924            !Decimal constants
    76.346
     1011b            !Binary constants
-1000110B
     3403o            !Octal constants
-70413120
    -1FFBh            !Hexadecimal constants
0CD1F74FH
```

Numeric Expressions

Numeric expressions may be used as parameters of procedures or functions, the condition of IF, CASE, LOOP, or EXECUTE structures, or as the source portion of an assignment statement where the destination is a numeric variable. A numeric expression may contain arithmetic operators and the concatenation operator, but they may not contain logical operators. When used in a numeric expression, string constants and variables are converted to numeric intermediate values. If the concatenation operator is used, the intermediate value is converted to numeric after the concatenation occurs.

Example:

```
Count + 1           !Add 1 to Count
(1 - N * N) / R     !N times N subtracted from 1 then divided by R
305 & 7854555       !Concatenate area code with phone number
```

See Also: Data Conversion Rules

String Constants

A string constant is a set of characters enclosed in single quotes (apostrophes). The maximum length of a string constant is 255 characters. Characters that cannot be entered from the keyboard may be inserted into a string constant by enclosing their ASCII character codes in angle brackets (<>). ASCII character codes may be represented in decimal or hexadecimal numeric constant format.

In a string constant, a left angle bracket (<) initiates a scan for a right angle bracket. Therefore, to include a left angle bracket in a string constant requires two left angle brackets in succession. To include an apostrophe as part of the value inside a string constant requires two apostrophes in succession. Two apostrophes (''), with no characters (or just spaces) between them, represents a null, or blank, string. Consecutive occurrences of the same character within a string constant may be represented by *repeat count* notation. The number of times the character is to be repeated is placed within curly braces ({ }) immediately following the character to repeat. To include a left curly brace ({) as part of the value inside a string constant requires two left curly braces ({{) in succession.

Example:

```
'string constant'     !A string constant
'It''s a girl!'       !With embedded apostrophe
'<27,15>'             !Using decimal ASCII codes
'A << B'              !With embedded left angle, A < B
'*{20}'               !Twenty asterisks, repeat-count notation
''                    !A null (blank) string
```

The Concatenation Operator

The concatenation operator (&) is used to append one string or variable to another. The length of the result string is the sum of the lengths of the two values being concatenated. Numeric data types may be concatenated with strings or other numeric variables or constants. In many cases, the CLIP function should be used to remove any trailing spaces from a string being concatenated to another string.

Example:

```
CLIP(FirstName) & ' ' Initial & '. ' & LastName     !Concatenate full name
'Clarion Software' & ', Inc.'                       !Concatenate two constants
```

See Also: CLIP, Numeric Expressions, Data Conversion Rules

String Expressions

String expressions may be used as parameters of procedures, functions, and attributes, or as the source portion of an assignment statement when the destination is a string variable. String expressions may contain a single string or numeric variable, or a complex combination of sub-expressions, functions, and operations.

Example:

```
StringVar   STRING(30)
Name        STRING(10)
Weight      STRING(3)
Phone       LONG
  CODE
  StringVar = ‘Address:’ & Cus:Address      !Concatenate a constant and variable

  StringVar = ‘Phone:’ & ‘ 305-’ & FORMAT(Phone,@P###-####P)
                                            !Concatenate constant valuess
                                            ! and FORMAT function’s return value

  StringVar = Weight & ‘lbs.’               !Concatenate a constant and variable
```

Implicit String Arrays and String Slicing

In addition to their explicit declaration, all STRING, CSTRING and PSTRING variables have an implicit array declaration of one character strings, dimensioned by the length of the string. This is directly equivalent to declaring a second variable as:

```
StringVar     STRING(10)
StringArray   STRING(1),DIM(SIZE(StringVar)),OVER(StringVar)
```

This implicit array declaration allows each character in the string to be directly addressed as an array element, without the need of the second declaration.

If the string also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions). The MAXIMUM function does not operate on the implicit dimension, you should use SIZE instead.

You may also directly address multiple characters within a string using the "string slicing" technique. This technique performs a similar function to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used as either the *destination* or *source* sides of an assignment statement, while the SUB function can only be used as the source. It is more efficient because it takes less memory than either individual character assignments or the SUB function.

To take a "slice" of the string, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the string. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Example:

```
Name      STRING(15)
CONTACT   STRING(15),DIM(4)
  CODE
  Name = 'Tammi'              !Assign a value
  Name[5] = 'y'               ! then change fifth letter
  Name[6] = 's'               ! then add a letter
  Name[0] = '<6>'             ! and handle length byte
  Name[5:6] = 'ie'            ! and change a "slice"
                              !  -- the fifth and sixth letters
  Contact[1] = 'First'        !Assign value to first element
  Contact[1,2] = 'u'          !Change first element 2nd character
  Contact[1,2:3] = Name[5:6]  !Assign slice to first element 2nd & 3rd characters
```

See Also: STRING, CTRING, PSTRING

Logical Expressions

Logical expressions evaluate true-false conditions in IF, LOOP UNTIL, and LOOP WHILE control structures. Control is determined by the final result (true or false) of the expression. Logical expressions are evaluated from left to right. The right operand of an AND, OR, or XOR logical expression will only be evaluated if it could affect the result. Parentheses should be used to eliminate ambiguous evaluation and to control evaluation precedence. The level or precedence for the logical operators is as follows:

Level 1	Conditional operators
Level 2	~, NOT
Level 3	AND
Level 4	OR, XOR

Example:

```
LOOP UNTIL KEYBOARD()          !True when user presses any key
 !some statements
END

IF A = B THEN RETURN.          !RETURN if A is equal to B

LOOP WHILE ~ Done#             !Loop while false (Done# = 0)
 !some statements
END

IF A >= B OR (C > B AND E = D) THEN RETURN.
                               !True if a >= b, also true if
                               ! both c > b and e = d.
                               !The second part of the expression
                               ! (after OR) is evaluated only if the
                               ! first part is not true.
```

Runtime Expression Strings

Clarion for Windows has the ability to evaluate Clarion language expressions dynamically created at runtime, rather than at development time. This allows a Clarion program to contruct expressions "on the fly." This also makes it possible to allow an end-user to enter the expression to evaluate.

An expression is a mathematical or logical formula that produces a value; it is not a complete Clarion language statement. Expressions may only contain constant values, variables, or function calls connected by logical and/or arithmetic operators. An expression may be used as the source side of an assignment statement, a parameter of a procedure or function, a subscript of an array (a dimensioned variable), or the conditions of IF, CASE, LOOP, or EXECUTE structures.

Any program variable, and most of the internal Clarion functions, can be used as part of a runtime expression string. User-defined functions that fall within certain specific guidelines (described in the BIND statement documentation) may also be used in runtime expression strings.

All of the standard Clarion expression syntax is available for use in runtime expression strings. This includes parenthetical grouping and all the arithmetic, logical, and string operators. Dynamic expressions are evaluated just as any other Clarion expression and all the standard operator precedence level rules described in the Expression Evaluation section (see page 3) apply.

It takes three steps to use runtime expression strings:

- The variables that are allowed to be used in the expressions must be explicitly declared with the BIND statement.
- The expression must be built. This may involve concatenating user choices or allowing the user to directly type in their own expression.
- The expression is passed to the EVALUATE function which returns the result. If the expression is not a valid Clarion expression, ERRORCODE is set.

Once the expression is evaluated, its result is used just as the result of any hard-coded expression would be. For example, a runtime expression string could provide a filter expression to eliminate certain records when viewing or printing a database (the FILTER expression of a VIEW structure is an implicit runtime expression string).

BIND (declare runtime expression string variable)

```
BIND( | name,variable | )
      | name,function |
      | group         |
```

BIND Identifies variables allowed to be used in dynamic expressions.

name A string constant containing the identifier used in the dynamic expression. This may be the same as the *variable* or *function* label.

variable The label of any variable (including fields in FILE, GROUP, or QUEUE structures) or passed parameter. If it is an array, it must have only one dimension.

function The label of a Clarion language FUNCTION that returns a STRING, REAL, or LONG value. If parameters are passed to the function, they must be STRING value-parameters (passed by value, not by address).

group The label of a GROUP, RECORD, or QUEUE structure declared with the BINDABLE attribute.

The **BIND** statement declares the logical name used to identify a variable or user-defined function in runtime expression strings. A variable or user-defined function must be identified with the BIND statement before it can be used in an expression string.

BIND(*name,variable*)
The specified *name* is used in the expression in place of the label of the *variable*.

BIND(*name,function*)
The specified *name* is used in the expression in place of the label of the *function*.

BIND(*group*) Declares all the variables within the GROUP, RECORD, or QUEUE (with the BINDABLE attribute) available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used.

A GROUP, RECORD, or QUEUE structure declared with the BINDABLE attribute has space allocated in the .EXE for the names of all of the data elements in the structure. This creates a larger program that uses more memory than it normally would. Also, the more variables that are bound at one time, the slower the EVALUATE function will work. Therefore, BIND(*group*) should only be used when a large proportion of the constituent fields are going to be used.

Example:

```
  PROGRAM
  MAP
    AllCapsFunc(STRING),STRING                  !Clarion function
  END

Header      FILE,DRIVER('Clarion'),PRE(Hea)     !Declare header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey      KEY(Hea:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
ShipToCity      STRING(20)
ShipToState     STRING(20)
ShipToZip       STRING(20)
            . .

Detail      FILE,DRIVER('Clarion'),PRE(Dtl),BINDABLE   !Bindable RECORD structure
OrderKey      KEY(Dtl:OrderNumber)
Record        RECORD
OrderNumber     LONG
Item            LONG
Quantity        SHORT
            . .

  CODE
  BIND('ShipName',Hea:ShipToName)       !BIND a single variable
  BIND(Dtl:Record)                      !BIND a RECORD structure
  BIND('SomeFunc',AllCapsFunc)          !BIND a Clarion language function
  IF EVALUATE('ShipName = SomeFunc(ShipName)')
    MESSAGE('Name is in ALL CAPS')
  END

AllCapsFunc  FUNCTION(PassedString)
  CODE
  RETURN(UPPER(PassedString))
```

See Also: UNBIND, EVALUATE

UNBIND (free runtime expression string variable)

UNBIND([*name*] **)**

UNBIND Frees variables from use in runtime expression strings.

name A string constant that specifies the identifier used by the dynamic expression evaluator. If omitted, all bound variables are unbound.

The **UNBIND** statement frees logical names previously bound by the BIND statement. The more variables that are bound at one time, the slower the EVALUATE function works. Therefore, UNBIND should be used to free all variables and user-defined functions not currently available for use in runtime expression strings.

Example:

```
 PROGRAM
 MAP
   AllCapsFunc(STRING),STRING              !Clarion function
 END

Header      FILE,DRIVER('Clarion'),PRE(Hea)   !Declare header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey      KEY(Hea:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
ShipToCity      STRING(20)
ShipToState     STRING(20)
ShipToZip       STRING(20)
            . .

Detail      FILE,DRIVER('Clarion'),PRE(Dtl),BINDABLE   !Bindable RECORD structure
OrderKey      KEY(Dtl:OrderNumber)
Record        RECORD
OrderNumber     LONG
Item            LONG
Quantity        SHORT
            . .

  CODE
  BIND('ShipName',Hea:ShipToName)
  BIND(Dtl:Record)
  BIND('SomeFunc',AllCapsFunc)
  UNBIND('ShipName')                !UNBIND the variable
  UNBIND('SomeFunc')                !UNBIND the Clarion language function
  UNBIND                              !UNBIND all bound variables

AllCapsFunc  FUNCTION(PassedString)
  CODE
  RETURN(UPPER(PassedString))
```

See Also: BIND, EVALUATE

EVALUATE (return runtime expression string result)

EVALUATE(*expression***)**

EVALUATE Evaluates runtime expression strings.

expression A string constant or variable containing the expression to evaluate.

The **EVALUATE** function returns the result of the *expression* as a STRING value. If the *expression* does not meet the rules of a valid Clarion expression, the result will be a null string, and the ERRORCODE function is set.

The more variables are bound at one time, the slower the EVALUATE function works. Therefore, BIND(*group*) should only be used when most of the *group's* fields are needed, and UNBIND should be used to free all variables and user-defined functions not currently required for use in dynamic expressions.

Return Data Type: STRING

Errors Posted: 800 Illegal Expression
801 Variable Not Found

Example:

```
 PROGRAM
 MAP
   AllCapsFunc(STRING),STRING                          !Clarion function
 END
Header      FILE,DRIVER('Clarion'),PRE(Hea),BINDABLE   !Declare header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey      KEY(Hea:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
ShipToCity      STRING(20)
ShipToState     STRING(20)
ShipToZip       STRING(20)
            . .
StringVar   STRING(20)
  CODE
  BIND('ShipName',Hea:ShipToName)
  BIND('SomeFunc',AllCapsFunc)
  StringVar = 'SMITH'
  IF EVALUATE('StringVar = SomeFunc(ShipName)')
    DO SmithProcess
  END
AllCapsFunc  FUNCTION(PassedString)
  CODE
  RETURN(UPPER(PassedString))
```

See Also: BIND, UNBIND

Assignment Statements

Simple Assignment Statements

destination **=** *source*

destination — The label of a variable or data structure property.

source — A numeric or string constant, variable, function, expression, or data structure property.

The = sign assigns the value of *source* to the *destination*; it copies the value of the *source* expression into the *destination* variable. If *destination* and *source* are different data types, the value the *destination* receives from the *source* is dependent upon the Data Conversion Rules.

Example:

```
Name = 'JONES'                          !Variable = string constant
PI = 3.14159                            !Variable = numeric constant
Cosine = SQRT(1 - Sine * Sine)          !Variable = function return value
A = B + C + 3                           !Variable = numeric expression
Name = CLIP(FirstName) & ' ' Initial & '. ' & LastName
                                        !Variable = string expression
```

See Also: Data Conversion Rules

Operating Assignment Statements

destination += *source*
destination -= *source*
destination *= *source*
destination /= *source*
destination ^= *source*
destination %= *source*

destination — Must be the label of a variable. This may not be any type property (window, control, report, etc.).

source — A constant, variable, function, or expression.

Operating assignment statements perform their operation on the *destination* and *source*, assigning the result to the *destination*. Operating assignment statements are more efficient than their equivalent operations.

Example:

```
Operating Assignment    Functional Equivalent
  A += 1                    A = A + 1
  A -= B                    A = A - B
  A *= -5                   A = A * -5
  A /= 100                  A = A / 100
  A ^= I + 1                A = A ^ (I + 1)
  A %= 7                    A = A % 7
```

Deep Assignment Statements

destination **:=:** *source*

destination The label of a GROUP, RECORD, or QUEUE data structure, or an array.

source The label of a GROUP, RECORD, or QUEUE data structure, or a numeric or string constant, variable, function, or expression.

The **:=:** sign executes a deep assignment statement which performs multiple individual component variable assignments from one data structure to another. The assignments are only performed between the variables within each structure that have exactly matching labels, ignoring all prefixes. The compiler looks within nested GROUP structures to find matching labels. Any variable in the *destination* which does not have a label exactly matching a variable in the *source*, is not changed.

Deep assignments are performed just as if each matching variable were individually assigned to its matching variable. This means that all normal data conversion rules apply to each matching variable assignment. For example, the label of a nested *source* GROUP may match a nested *destination* GROUP or simple variable. In this case, the nested *source* GROUP is assigned to the *destination* as a STRING, just as normal GROUP assignment is handled.

The name of a *source* array may match a *destination* array. In this case, each element of the *source* array is assigned to its corresponding element in the *destination* array. If the *source* array has more or fewer elements than the *destination* array, only the matching elements are assigned to the *destination*.

If the *destination* is an array variable that is not part of a GROUP, RECORD, or QUEUE, and the *source* is a constant, variable, or expression, then each element of the *destination* array is initialized to the value of the *source*. This is a much more efficient method of initializing an array to a specific value than using a LOOP structure and assigning each element in turn.

Example:

```
Group1  GROUP
S         SHORT
L         LONG
        END

Group2  GROUP
L         SHORT
S         REAL
T         LONG
        END

ArrayField  SHORT,DIM(1000)

  CODE
  Group2 :=: Group1     !Is equivalent to:
                        !  Group2:S = Group1:S
                        !  Group2:L = Group1:L
                        ! and performs all necessary data conversion

  ArrayField :=: 7      !Is equivalent to:
                        !   LOOP I# = 1 to 1000
                        !     ArrayField[I#] = 7
                        !   END
```

Reference Assignment Statements

destination &= *source*

destination The label of a reference variable.

source The label of another reference variable of the same type as the *destination*, or the label of a variable or data structure of the type referenced by the *destination*. This cannot be an expression, only a data label.

The **&=** sign executes a reference assignment statement which assigns to the *destination* reference variable the reference to the *source* variable. Depending upon the data type, the *destination* reference variable may receive the *source's* memory address, or a more complex internal data structure (describing the location and type of *source* data).

The declarations of the *destination* reference variable and its *source* must match exactly; reference assignment does not perform automatic type conversion. For example, a reference assignment statement to a *destination* declared as &QUEUE must have a *source* that is either another &QUEUE reference variable, or the label of a QUEUE structure.

Example:

```
Queue1    QUEUE
ShortVar    SHORT
LongVar1    LONG
LongVar2    LONG
          END

QueueRef  &QUEUE      !Reference a QUEUE, only
LongRef   &LONG       !Reference a LONG, only

  CODE
  QueueRef &= Queue1            !Assign QUEUE reference
  IF SomeCondition              !Evaluate some condition
    LongRef &= Queue1:LongVar1  ! and reference an appropriate variable
  ELSE
    LongRef &= Queue1:LongVar2
  END
  LongRef += 1                  !Increment either LongVar1 or LongVar2
                                ! depending upon which variable is referenced
```

See Also: ReferenceVariables

CLEAR (clear a variable)

CLEAR(*label* [,*n*]**)**

CLEAR	Clears any value from a variable.
label	The label of a variable.
n	A numeric constant; 1 or -1. This parameter indicates a cleared value other than zero or blank. If *n* is 1, the variable is set to the highest possible value for that data type. For STRING, PSTRING and CSTRING, that is ASCII 255. If *n* is -1, the variable is set to the lowest possible value for that data type. For STRING, PSTRING and CSTRING, that is ASCII 0.

The **CLEAR** statement clears any value from the *label* variable. If *n* is omitted, numeric variables are cleared to zero, and string variables are cleared to spaces. If the *label* parameter is a GROUP, RECORD, or QUEUE structure name, all variables in the structure are cleared. If the variable has a DIM attribute, the entire array is cleared. A single element of an array cannot be CLEARed.

Example:

```
CLEAR(Count)          !Clear a variable
CLEAR(Cus:Record)     !Clear the record structure
CLEAR(Amount,1)       !Clear variable to highest possible value
CLEAR(Amount,-1)      !Clear variable to lowest possible value
```

Data Conversion Rules

The Clarion language provides automatic conversion between data types. However, some assignments can produce an unequal source and destination. Assigning an "out of range" value can produce unpredictable results.

Base Types

To facilitate this automatic data type conversion, Clarion internally uses four Base Types to which all data items are automatically converted when any operation is performed on the data. These types are: STRING, LONG, DECIMAL, and REAL.These are all standard Clarion data types.

The STRING Base Type is used as the intermediate type for all string operations. The LONG, DECIMAL, and REAL Base Types are used in all arithmetic operations. Which numeric type is used, and when, is determined by the original data types of the operands and the type of operation being performed on them.

The "normal" Base Type for each data type is:

```
Base Type LONG:
    BYTE
    SHORT
    USHORT
    LONG
    DATE
    TIME
    Integer Constants

Base Type DECIMAL:
    ULONG
    DECIMAL
    PDECIMAL
    STRING(@Nx.y)
    Decimal Constants

Base Type REAL:
    SREAL
    REAL
    BFLOAT4
    BFLOAT8
    STRING(@Ex.y)
    Scientific Notation Constants
    Untyped (? and *?) Parameters

Base Type STRING:
    STRING
    CSTRING
    PSTRING
    String Constants
```

DATE and TIME data types are first converted to Clarion Standard Date and Clarion Standard Time intermediate values and have a LONG Base Type for

all operations.

For the most part, Clarion's internal use of these Base Types is transparent to the programmer and do not require any consideration when planning applications. However, for business programming with numeric data containing fractional portions (currency, for instance), using data types that have the DECIMAL Base Type has some significant advantages over REAL Base Types.

- DECIMAL supports 31 significant digits of accuracy for data storage while REAL only supports 15.
- DECIMAL automatically rounds to the precision specified by the data declaration, while REAL can create rounding problems due to the transalation of decimal (base 10) numbers to binary (base 2) for processing by the CPU's Floating Point Unit (or Floating Point emulation software).
- On machines without a Floating Point Unit, DECIMAL is substantially faster than REAL.
- DECIMAL operations are closely linked with conventional (decimal) arithmetic.

BCD Operations and Functions

Clarion has a Binary Coded Decimal (BCD) library of operations and functions that execute in a manner similar to the manner in which decimal arithmetic is performed on paper. These operations use internal intermediate values with 31 digits accuracy on both sides of the decimal point.

The big advantage of the BCD operations is that it is very easy to "see" what is happening because they execute just as you would with pencil and paper. Simply imagine doing the computation long hand and throwing away numbers that go off the end of the page (rounding to the right).

Having 31 fixed decimal places either side of the decimal point there are numbers that cannot be represented in a BCD system which can be represented by a REAL. Therefore, understanding what is going on is useful.

Generally, the only cases where underflow will affect you is in division operations, usually when dividing by a multiple of 3. For example:

```
100000/3 = 33333.3333333333333333333333333333333
(100000/3)-INT(100000/3)*100000 = 33333.33333333333333333333333300000
```

BCD computation times are very data sensitive; the time taken is proportional to how long the computation would take you by hand. Therefore, the longer the numbers involved, the longer the execution times. However, standard "tricks of the trade" (such as multiplying by a power of

ten by shifting the decimal point) are spotted, making the BCD libraries fast in real world applications.

The following operations may execute as BCD operations:

Addition (+), **Subtraction** (-), **Multiplication** (*)
: Performed as a BCD operation when neither operand has a REAL Base Type (both are LONG or DECIMAL) and one has the DECIMAL Base Type. Any digits appearing to the right of 1e31 disappear (wrap), and any to the left of 1e-30 are rounded up.

Division (/)
: Performed as a BCD operation when neither operand has a REAL Base Type (both are LONG or DECIMAL). Any digits appearing to the right of 1e31 disappear (wrap), and any to the left of 1e-30 are rounded up.

Exponentiation (^)
: Performed as a BCD operation when the first operand is a DECIMAL or LONG Base Type and the second operand is a LONG Base Type. Any digits appearing to the right of 1e31 disappear (wrap), and any to the left of 1e-30 are rounded.

ABS()
: Removes the sign from a DECIMAL variable or intermediate value and returns the DECIMAL value.

INT()
: Truncates a DECIMAL intermediate value and returns a DECIMAL value.

ROUND()
: If the second parameter is a LONG or DECIMAL Base Type, then rounding is performed as a BCD operation which returns a DECIMAL value. ROUND is very efficient as a BCD operation and should be used to compare REALs to DECIMALs at decimal width.

Type Conversion and Intermediate Results

Internally, a BCD intermediate result may have up to 31 digits of accuracy on both sides of the decimal point, so any two DECIMALs can be added with complete accuracy. Therefore, storage from BCD intermediate results to a data type can result in loss of precision. This is handled as follows :

Decimal(x,y) = BCD
: First the BCD value is rounded to y decimal places. If the result overflows x digits then leading digits are removed (this corresponds to “wrapping around” a decimal counter).

Integer = BCD
: Any digits to the right of the decimal point are ignored. The decimal is then converted to an integer with complete accuracy and then taken modulo 2^32.

String(@Nx.y) = BCD
The BCD value is rounded to y decimal places, the result is fitted into the pictured string. If overflow occurs, an invalid picture (####) results.

Real = BCD The most significant 15 digits are taken and the decimal point 'floated' accordingly.

For those operations and functions that do not support DECIMAL types, the DECIMAL is converted to REAL first. In cases where more than 15 digits were available in the DECIMAL value, there is a loss of accuracy.

Note: Untyped parameters have an implicit REAL Base Type, therefore DECIMAL Base Type data passed as an Untyped Parameterswill only have 15 digits of precision. DECIMAL Base Types can be passed as *DECIMAL parameters with no loss of precision.

When EVALUATEing a expression (or processing a VIEW FILTER) the REAL Base Type is used.

Simple Assignment Data Conversion

The rules of simple assignment data conversion from source into destination are as follows:

BYTE = **(SHORT, USHORT, LONG, or ULONG)**
The destination receives the low-order 8 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 8 bits of the LONG.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no formatting characters. The source is converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 8 bits of the LONG.

SHORT = **BYTE** The destination receives the value of the source.

(USHORT, LONG, or ULONG)
The destination receives the low-order 16 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

USHORT = **BYTE** The destination receives the value of the source.

(SHORT, LONG, or ULONG)
The destination receives the low-order 16 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

LONG = **(BYTE, SHORT, USHORT, or ULONG)**
The destination receives the value and the sign of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The destination receives the value of the source, including the sign, up to 2^{31}. If the number is greater than 2^{31}, the destination receives the result of modulo 2^{31}. Any decimal portion is truncated.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no embedded formatting characters. The source is first converted to a REAL, which is then converted to the LONG.

DATE = **(BYTE, SHORT, USHORT, or ULONG)**
The destination receives the Btrieve format for the Clarion Standard Date for the value of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG as a Clarion Standard Date, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Date.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG as a Clarion Standard Date, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Date.

TIME = **(BYTE, SHORT, USHORT, or ULONG)**
The destination receives the Btrieve format for the Clarion Standard Time for the value of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG as a Clarion Standard Time, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Time.

(STRING, CSTRING, PSTRING)
The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG as a Clarion Standard Time, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Time.

ULONG = **(BYTE, SHORT, or USHORT)**
The source is first converted to a LONG, then the destination receives the entire 32 bits of the LONG.

LONG The destination receives the entire 32 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the entire 32 bits of the LONG.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the entire 32 bits of the LONG.

REAL = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the full integer portion and the sign of the source.

(DECIMAL, PDECIMAL, SREAL, BFLOAT8, or BFLOAT4)
The destination receives the sign, integer portion, and the decimal portion of the source.

(STRING, CSTRING, PSTRING)
The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

SREAL = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)
The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)
The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

BFLOAT8 = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)
The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)
The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

BFLOAT4 = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)
The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)
The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

DECIMAL = **(BYTE, SHORT, USHORT, LONG, ULONG, or PDECIMAL)**
The destination receives the sign and the value of the source, wrapping or rounding as appropriate.

(REAL, or SREAL)
The destination receives the sign, integer, and the high order part of the fraction from the source. The high order fractional portion is rounded in the destination.

(STRING, CSTRING, PSTRING)
The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

PDECIMAL = **(BYTE, SHORT, USHORT, LONG, ULONG, or DECIMAL)**
The destination receives the sign and the value of the source, wrapping or rounding as appropriate.

(REAL, SREAL, BFLOAT8, or BFLOAT4)
The destination receives the sign, integer, and the high order part of the fraction from the source. The high order fractional portion is rounded in the destination.

(STRING, CSTRING, or PSTRING)
The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

STRING = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The destination receives the sign, integer, and fractional portion of the source (rounded into the string's picture format). The value is left justified in the destination.

CSTRING = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The destination receives the sign, integer, and fractional portion of the source (rounded into the string's picture format). The value is left justified in the destination.

PSTRING = **(BYTE, SHORT, USHORT, LONG, or ULONG)**
The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The destination receives the sign, integer, and fractional portion of the source (rounded into the string's picture format). The value is left justified in the destination.

Control Structures

Contents

CASE (selective execution structure)

```
CASE condition
OF expression [ TO expression ]
  statements
[ OROF expression [ TO expression ] ]
  statements
[ ELSE ]
  statements
END
```

CASE	Initiates a selective execution structure.
condition	A numeric or string variable or expression.
OF	The *statements* following an **OF** are executed when the *expression* following the OF option is equal to the *condition* of the CASE. There may be many OF options in a CASE structure.
expression	A numeric or string constant, variable, or expression.
TO	**TO** allows a range of values in an OF or OROF. The *statements* following the OF (or OROF) are executed if the value of the *condition* falls within the inclusive range specified by the *expressions*. The *expression* following OF (or OROF) must contain the lower limit of the range. The *expression* following TO must contain the upper limit of the range.
OROF	The *statements* following an **OROF** are executed when either the *expression* following the OROF or the OF option is equal to the *condition* of the CASE. There may be many OROF options associated with one OF option. An OROF may optionally be put on a separate line. An OROF does not terminate preceding *statements* groups, so control "falls into" the OROF *statements*.
ELSE	The *statements* following **ELSE** are executed when all preceding OF and OROF options have been evaluated as not equivalent. ELSE is not required; however, when used, it must be the last option in the CASE structure.
statements	Any valid Clarion executable source code.

A **CASE** structure selectively executes *statements* based on equivalence between the *condition* and *expression* or range of *expressions*. CASE structures may be nested within other executable structures and other executable structures may be nested within CASE structures. The CASE structure must terminate with an END statement (or period).

Example:

```
CASE ACCEPTED()                       !Evaluate field edit routine
OF ?Name                              !If field is Name
  ERASE(?Address,?Zip)                ! erase Address through Zip
  GET(NameFile,NameKey)               ! get the record

  CASE Action                         !Evaluate Action
  OF 1                                ! adding record - does not exist
    IF NOT ERRORCODE()                ! should be a file error
      ErrMsg = 'ALREADY ON FILE'      !   otherwise display error message
      DISPLAY(?Address,?Zip)          !   display address through zipcode
      SELECT(?Name)                   !   re-enter the name
    END
  OF 2 OROF 3                         ! change or delete - record exists
    DISPLAY(?Address,?Zip)            !   display address through zipcode
  END                                 ! end case action

  CASE Name[1]                        !Get first letter of name
  OF 'A' TO 'M'                       !Process first half of alphabet
  OROF 'a' TO 'm'
    DO FirstHalf
  OF 'N TO 'Z' OROF 'n' TO 'z'        !Process second half of alphabet
    DO SecondHalf
  END                                 !End case sub(name

OF ?Address                           !If field is address
  DO AddressVal                       ! call validation routine
END                                   !End case accepted()
```

EXECUTE (statement execution structure)

```
EXECUTE expression
  statement 1
  statement 2
  [ BEGIN
        statements
  END ]
  statement n
END
```

EXECUTE	Initiates a single statement execution structure.
expression	A numeric expression or a variable that contains a numeric integer.
statement 1	A single statement that executes only when the *expression* is equal to 1.
statement 2	A single statement that executes only when the *expression* is equal to 2.
BEGIN	BEGIN marks the beginning of a structure containing a number of lines of code. The BEGIN structure will be treated as a single statement by the EXECUTE structure. The BEGIN structure is terminated by a period or the keyword END.
statement n	A single statement that executes only when the *expression* is equal to *n*.

An **EXECUTE** structure selects a single executable statement (or executable code structure) based on the value of the *expression*. The EXECUTE structure must terminate with an END statement (or period).

If the *expression* equals 1, the first statement (*statement 1*) executes. If *expression* equals 2, the second statement (*statement 2*) executes, and so on. If the value of the *expression* is zero, or greater than the total number of statements (or structures) within the EXECUTE structure, program execution continues with the next statement following the EXECUTE structure.

EXECUTE structures may be nested within other executable structures. Other executable structures (IF, CASE, LOOP, EXECUTE, and BEGIN) may be nested within an EXECUTE.

Example:

```
EXECUTE Transact          !Evaluate Transact
  ADD(Customer)           !Execute if Transact = 1
  PUT(Customer)           !Execute if Transact = 2
  DELETE(Customer)        !Execute if Transact = 3
END                       !End execute

EXECUTE CHOICE()          !Evaluate CHOICE() function
  OrderPart               !Execute if CHOICE() = 1
  BEGIN                   !Execute if CHOICE() = 2
    SavVendor" = Vendor
    UpdVendor
    IF Vendor <> SavVendor"
      Mem:Message = 'VENDOR NAME CHANGED'
  . .
  CASE VendorType         !Execute if CHOICE() = 3
  OF 1
    UpdPartNo1
  OF 2
    UpdPartNo2
  END
  RETURN                  !Execute if CHOICE() = 4
END                       !End execute
```

See Also: BEGIN

IF (conditional execution structure)

```
IF logical expression [ THEN ]
  statements
[ ELSIF logical expression  [ THEN ]
  statements ]
[ ELSE
  statements ]
END
```

IF	Initiates a conditional statement execution structure.
logical expression	A numeric or string variable, expression, or function. A *logical expression* evaluates a condition. Control is determined by the result (true or false) of the expression. A zero (or blank) value evaluates as false, anything else is true.
THEN	The *statements* following THEN are executed when the preceding *logical expression* is evaluated as true. If used, THEN must only be placed on the same line as the **IF** or **ELSIF**.
statements	An executable statement, or a sequence of executable statements.
ELSIF	The *logical expression* following an **ELSIF** is evaluated only when all preceding **IF** or **ELSIF** conditions were evaluated as false.
ELSE	The *statements* following **ELSE** are executed when all preceding **IF** and **ELSIF** options were evaluated as false. ELSE is not required, however, when it is used, it must be the last option in the IF structure.

An **IF** structure controls program execution based on the outcome of one or more *logical expressions*. IF structures may have any number of ELSIF statement groups. IF structures may be "nested" within other executable structures. Other executable structures may be nested within an IF structure. Each IF structure must terminate with an END statement (or period).

Example:

```
IF Cus:TransCount = 1                                  !If new customer
  AcctSetup                                            ! call account setup procedure
ELSIF Cus:TransCount > 10 AND Cus:TransCount < 100     !If regular customer
  DO RegularAcct                                       ! process the account
ELSIF Cus:TransCount > 100                             !If special customer
  DO SpecialAcct                                       ! process the account
ELSE                                                   !Otherwise
  DO NewAcct                                           ! process the account
  IF Cus:Credit THEN CheckCredit ELSE CLEAR(Cus:CreditStat).
                                                       ! verify credit status
END

IF ERRORCODE() THEN ErrHandler(Cus:AcctNumber,Trn:InvoiceNbr). !Handle errors
```

LOOP (iteration structure)

```
LOOP [ | count TIMES                    | ]
       | i = initial TO limit [ BY step ] |
       |  UNTIL logical expression        |
       | WHILE logical expression         |
  statements
END
```

LOOP	Initiates an iterative statement execution structure.
count	A numeric constant, variable, or expression that determines the number of TIMES the *statements* in the LOOP are executed.
TIMES	Executes *count* number of iterations of the *statements*.
i	The label of a variable which is automatically incremented on each iteration of the LOOP.
=	Assigns a new value to the increment (*i*) variable for each cycle of the LOOP.
initial	A numeric constant, variable, or expression that specifies the initial value assigned to the increment variable (*i*) on the first pass through the LOOP structure.
TO	A syntax conjunctive for the *limit* parameter.
limit	When *i* is greater than *limit*, the LOOP structure control sequence terminates.
BY	A syntax conjunctive for the *step* parameter.
step	A numeric constant, variable, or expression. The *step* determines the quantity by which the *i* variable increments on each iteration of the LOOP. If the BY *step* parameter is omitted, *i* increments by 1.
UNTIL	Evaluates the *logical expression* before each iteration of the LOOP. If the *logical expression* evaluates to true, the LOOP control sequence terminates.
WHILE	Evaluates the *logical expression* before each iteration of the LOOP. If the *logical expression* evaluates to false, the LOOP control sequence terminates.
logical expression	A numeric or string variable, expression, or function. A *logical expression* evaluates a condition. Control is determined by the result (true or false) of the expression. A zero (or blank) value evaluates as false, anything else is true.
statements	An executable statement, or a sequence of executable statements.

A **LOOP** structure repetitively executes the *statements* within its structure.

LOOP conditions are always evaluated at the top of the LOOP, before the LOOP is executed. LOOP structures may be nested within other executable code structures. Other executable code structures may be nested within a LOOP structure. Each LOOP structure must terminate with an END statement (or period).

A LOOP with no parameters iterates continuously, unless a BREAK or RETURN statement is executed. BREAK discontinues the LOOP and continues program execution with the statement following the LOOP structure. All statements within a LOOP structure are executed unless a CYCLE statement is executed. CYCLE immediately sends program execution back to the top of the LOOP for the next iteration, without executing any statements following the CYCLE in the LOOP.

Example:

```
LOOP                                !Continuous loop
  Char = GetChar()                  ! get a character
  IF Char <> CarrReturn             ! if it's not a carriage return
    Field = CLIP(Field) & Char      !   append the character
  ELSE                              ! otherwise
    BREAK                           !   break out of the loop
. .                                 !End if, end loop

IF ERRORCODE()                      !On error
  LOOP 3 TIMES                      ! loop three times
    BEEP                            !   sound the alarm
. .                                 !End loop, end if

LOOP I# = 1 TO 365 BY 7             !Loop, increment I# by 7 each time
  GET(DailyTotal,I#)                ! read every 7th record
  DO WeeklyJob                      ! do the routine
END                                 !End loop

SET(MasterFile)                     !Point to first record
LOOP UNTIL EOF(MasterFile)          !Process all the records
  NEXT(MasterFile)                  ! read a record
  ProcMaster                        ! call the procedure
END

LOOP WHILE KEYBOARD()               !Empty the keyboard buffer
  ASK                               ! without processing keystrokes
END
```

See Also: BREAK, CYCLE

Control Statements

BREAK (immediately leave loop)

BREAK

The **BREAK** statement immediately terminates the LOOP or ACCEPT loop and transfers control to the first statement following the LOOP or ACCEPT loop structure. BREAK may only be used in a LOOP or ACCEPT loop structure.

Example:

```
LOOP                        !Loop
  ASK                       ! wait for a keystroke
  IF KEYCODE() = 256        ! if Esc key pressed
    BREAK                   !   break out of the loop
  ELSE                      ! otherwise
    BEEP                    !   sound the alarm
  END
END

ACCEPT                      !ACCEPT loop structure
  CASE ACCEPTED()
  OF ?Ok
    CallSomeProc
  OF ?Cancel
    BREAK                   !   break out of the loop
  END
END
```

See Also: LOOP, CYCLE, ACCEPT

CHAIN (execute another program)

CHAIN(*program***)**

CHAIN	Terminates the current program and executes another.
program	A string constant or variable containing the name of the program to execute. This may be any .EXE or .COM program.

CHAIN terminates the current program, closing all files and returning its memory to the operating system, and executes another *program*.

Example:

```
PROGRAM                  !MainMenu program code
 CODE
 EXECUTE CHOICE()
   CHAIN('Ledger')       !Execute LEDGER.EXE
   CHAIN('Payroll')      !Execute PAYROLL.EXE
   RETURN                !Return to DOS
 END

  PROGRAM                !Ledger program code
 CODE
 EXECUTE CHOICE()
   CHAIN('MainMenu')     !Return to MainMenu program
   RETURN                !Return to DOS
 END

 PROGRAM                  !Payroll program code
 CODE
 EXECUTE CHOICE()
   CHAIN('MainMenu')     !Return to MainMenu program
   RETURN                !Return to DOS
 END
```

See Also: RUN

CYCLE (go to top of loop)

CYCLE

The **CYCLE** statement passes control immediately back to the top of the LOOP or ACCEPT loop, where the LOOP condition is evaluated. CYCLE may only be used in a LOOP or ACCEPT loop structure.

In an ACCEPT loop, for certain events, CYCLE terminates an automatic action before it is performed (such as EVENT:Move). This behavior is documented for each event so affected.

Example:

```
SET(MasterFile)                !Point to first record
LOOP                           !Process all the records
  NEXT(MasterFile)             ! read a record
  IF ERRORCODE() THEN BREAK.   !Get out of loop at end of file
  DO MatchMaster               ! check for a match
  IF NoMatch                   ! if match not found
    CYCLE                      !   jump to top of loop
  END
  DO TransVal                  ! validate the transaction
  PUT(MasterFile)              ! write the record
END
```

See Also: LOOP, BREAK, ACCEPT

DO (call a ROUTINE)

DO *label*

DO Executes a ROUTINE.

label The label of a ROUTINE statement.

The **DO** statement is used to execute a ROUTINE local to a PROGRAM, PROCEDURE, or FUNCTION. When a ROUTINE completes execution, program control reverts to the statement following the DO statement. A ROUTINE may only be called within the CODE section containing the ROUTINE's source code.

Example:

```
DO NextRecord     !Call the next record routine
DO CalcNetPay     !Call the calc net pay routine
```

See Also: EXIT, ROUTINE

EXIT (leave a ROUTINE)

EXIT

The **EXIT** statement immediately leaves a ROUTINE and returns program control to the statement following the DO statement that called it. This is different from RETURN, which completely exits the PROCEDURE or FUNCTION even when called from within a ROUTINE.

An EXIT statement is not required. A ROUTINE with no EXIT statement terminates automatically when the entire sequence of statements in the ROUTINE is complete.

Example:

```
CalcNetPay  ROUTINE
  IF GrossPay = 0       !If no pay
    EXIT                ! exit the routine
  END
  NetPay = GrossPay - FedTax - Fica
  QtdNetPay += NetPay
  YtdNetPay += NetPay
```

See Also: DO, RETURN

GOTO (go to a label)

GOTO *label*

GOTO	Unconditionally transfers program control to another statement.
label	The label of another executable statement within the PROGRAM, PROCEDURE, FUNCTION, or ROUTINE.

The **GOTO** statement unconditionally transfers control from one statement to another. The target *label* of a GOTO must not be the label of a ROUTINE, PROCEDURE, or FUNCTION.

The scope of GOTO is limited to the currently executing ROUTINE, PROCEDURE, or FUNCTION; it may not target a *label* outside the ROUTINE, PROCEDURE, or FUNCTION in which it is used.

Example:

```
ComputeIt FUNCTION(Level)
  CODE
  IF Level = 0 THEN GOTO PassCompute.   !Skip rate calculation if no Level
  Rate = Level * MarkUp                 !Compute Rate
  RETURN(Rate)                          ! and return it
PassCompute RETURN(999999)              !Return bogus number
```

HALT (exit program)

HALT([*errorlevel*] [,*message*]**)**

HALT	Immediately terminates the program.
errorlevel	A positive integer constant or variable (range: 0 - 250) which is the exit code to pass to DOS, setting the DOS ERRORLEVEL. If omitted,the default is zero.
message	A string constant or variable which is typed on the screen after program termination.

The **HALT** statement immediately returns to the operating system, setting the *errorlevel* and optionally displaying a *message* after the program terminates.

If the program being HALTed was launched by a RUN statement within another Clarion program, the *errorlevel* exit code HALT sets may be determined by using the RUNCODE function in the launching program.

Example:

```
PasswordProc PROCEDURE
Password  STRING(10)
Window WINDOW,CENTER
        ENTRY(@s10),AT(5,5),USE(Password),HIDE
       END
  CODE
  OPEN(Window)
  ACCEPT
    CASE ACCEPTED()
    OF ?Password)
      IF Password <> 'Pay$MeMoRe'
        HALT(0,'Incorrect Password entered.')
      END
    END
  END
```

See Also: RUN, RUNCODE, STOP

IDLE (arm periodic procedure)

IDLE([*procedure*] [,*separation*]**)**

IDLE	Arms a *procedure* that periodically executes.
procedure	The label of a PROCEDURE. The *procedure* may not take any parameters.
separation	An integer that specifies the minimum wait time (in seconds) between calls to the *procedure*. A *separation* of 0 specifies continuous calls. If *separation* is omitted, the default value is 1 second.

An **IDLE** procedure is active while ASK or ACCEPT are waiting for user input. Only one IDLE procedure may be active at a time, and it executes on thread zero (0). Naming a new IDLE procedure overrides the previous one. An IDLE statement with no parameters disarms the IDLE process.

An IDLE *procedure* is usually prototyped in the PROGRAM's MAP. If prototyped in a MEMBER MAP, the IDLE statements which activate and de-activate it must be contained in a procedure or function within the same MEMBER module.

Example:

```
IDLE(ShoTime,10)        !Call shotime every 10 seconds
IDLE(CheckNet)          !Check network activity every 1 second
IDLE                    !Disarm idle procedure
```

See Also: ASK, ACCEPT, PROCEDURE, MAP

RETURN (return to caller)

RETURN([*expression*] **)**

RETURN Terminates a PROGRAM, PROCEDURE, or FUNCTION.

expression The *expression* passes the return value of a FUNCTION back to the expression in which the FUNCTION was used. The *expression* is required for a FUNCTION and may not be used in a PROCEDURE or PROGRAM.

The **RETURN** statement terminates a PROGRAM, PROCEDURE, or FUNCTION, and passes control back to the caller. When RETURN is executed from the CODE section of a PROGRAM, the program is terminated, all files and windows are closed, and control is passed to the operating system.

RETURN is required in a FUNCTION and optional in a PROCEDURE or PROGRAM. If RETURN is not used in a PROCEDURE or PROGRAM, an implicit RETURN occurs at the end of the executable code. The end of executable code is defined as the end of the source file, or the beginning of another PROCEDURE, FUNCTION, or ROUTINE.

RETURN from a PROCEDURE or FUNCTION (whether explicit or implicit) automatically closes any local APPLICATION, WINDOW, REPORT, or VIEW structure opened in the PROCEDURE or FUNCTION. It does not automatically close any Global or Module Static APPLICATION, WINDOW, REPORT, or VIEW. It also closes and frees any local QUEUE structure declared without the STATIC attribute.

Example:

```
  IF Done# THEN RETURN.       !Quit when done

DayOfWeek FUNCTION(Date)      !Function to return the day of the week
  CODE
  EXECUTE (Date % 7) + 1      !Determine what day of week Date is
    RETURN('Sunday')          ! and RETURN the correct day string
    RETURN('Monday')
    RETURN('Tuesday')
    RETURN('Wednesday')
    RETURN('Thursday')
    RETURN('Friday')
    RETURN('Saturday')
  END
```

RUN (execute command)

RUN(*command* **)**

RUN Executes a *command* as if it were entered on the DOS command line.

command A string constant or variable containing the command to execute. This may include a full path and command line parameters.

The **RUN** statement executes a *command* to execute a DOS or Windows program. When the *command* executes, the new program is loaded as the ontop and active program. Execution control in the launching program returns immediately to the statement following RUN and the program continues executing as a background application. The user can return to the launching program by either terminating the launched program, or switching back to it through the Windows Task List.

If the *command* does not contain a path to the program, the following search sequence is followed:

1. The DOS current directory
2. The Windows directory
3. The Windows system directory
4. Each directory in the DOS PATH
5. Each directory mapped in a network

The successful execution of the *command* may be verified with the RUNCODE function, which returns the DOS exit code of the *command*. If unsuccessful, RUN posts the error to the ERROR and ERRORCODE functions.

Errors Posted: RUN may post any possible error

Example:

```
RUN('notepad.exe readme.txt')   !Run Notepad, automatically loading readme.txt file
RUN(ProgName)                   !Run the command in the ProgName variable
```

See Also: RUNCODE, HALT

SHUTDOWN (arm termination procedure)

SHUTDOWN([*procedure***])**

SHUTDOWN Arms a procedure which is called when the program terminates.

procedure The label of a PROCEDURE. If omitted, the **SHUTDOWN** process is disarmed.

The **SHUTDOWN** statement arms a *procedure* which is called when the program terminates. The shutdown *procedure* is called by normal program termination or by an abnormal-end/run-time halt. It may not be able to execute for an abnormal-end/run-time halt, depending upon the state of the system resources at the time of the crash. It is not called if the computer is rebooted or the program is terminated due to power failure. RESTART within a SHUTDOWN *procedure* is not recommended.

The same effect as SHUTDOWN can be more safely achieved by simply calling a procedure to execute on EVENT:CloseDown for the application frame.

Example:

```
SHUTDOWN(CloseSys)           !Arm CloseSys as the shutdown procedure
```

STOP (suspend program execution)

STOP([*message*] **)**

STOP	Suspends program execution and displays a message window.
message	An optional string expression (up to 64K) which displays in the error window.

STOP suspends program execution and displays a message window. It offers the user the option of continuing the program or exiting. When exiting, it closes all files and frees the allocated memory.

Example:

```
PswdScreen  WINDOW
              STRING(‘ Please Enter the Password ‘),AT(5,5)
              ENTRY(@10),AT(20,5),USE(Password),HIDE     !Password storage field
            END
  CODE
  OPEN(PswdScreen)              !Open the password screen
  ACCEPT                        ! and get user input
    CASE ACCEPTED
    OF ?Password)
      IF Password <> ‘PayMe$moRe’                  !Correct password?
        STOP(‘Incorrect Password Entered -- Access Denied’)
        HALT(0,’Incorrect password’)               !If not, throw them out
      END
    END
  END
```

Clarion Windows

Contents

Window Overview

In most Windows programs there are three types of screen windows used: application windows, document windows, and dialog boxes. An application window is the first window opened in a Windows program, and it usually contains the main menu as the entry point to the rest of the program. All other windows in the program are document windows or dialog boxes.

Along with these three screen window types, there are two user interface design conventions that are used in Windows programs: the Single Document Interface (SDI), and the Multiple Document Interface (MDI).

An SDI program usually only contains linear logic that allows the user to take only one execution path (thread) at a time; it does not open separate execution threads which the user may move between. This is the same type of program logic used in most DOS programs. An SDI program would not contain a Clarion APPLICATION structure as its application window. The Clarion WINDOW structure (without an MDI attribute) is used to define an SDI program's application window, and the subsequent document windows or dialog boxes opened on top of it.

An MDI program allows the user to choose multiple execution paths (threads) and change from one to another at any time. This is a very common Windows program user interface. It is used by applications as a way of organizing and grouping windows which present several execution paths for the user to take.

A Clarion APPLICATION structure defines the MDI application window. The MDI application window acts as a parent for all the MDI child windows (document windows and dialog boxes), in that the child windows are clipped to its frame and automatically moved when the application frame is moved. They can also be concealed en masse by minimizing the parent. There may be only one APPLICATION open at any time in a Clarion Windows program.

Document windows and dialog boxes are very similar in that they are both defined as Clarion WINDOW structures. They differ in the conventional context in which they are commonly used and the conventions regarding appearance and attributes. In many cases, the difference is not distinguishable and does not matter. The generic term for both document windows and dialog boxes is "window" and that is the term used throughout this text.

Document windows usually display data. By convention they are movable and resizable. They usually have a title, a system menu, and maximize

button. For example, in the Windows environment, the "Main" program group window that appears when you DOUBLE-CLICK on the "Main" icon in the Program Manager's desktop, is a document window.

Dialog boxes usually request information from the user or alert the user to some condition, usually prior to performing some action requested by the user. They may or may not be movable, and so, may or may not have a system menu and title. By convention, they are not resizable, although they can have a maximize button which gives the dialog two alternate sizes. A dialog box may be system modal (the user must respond before doing anything else in Windows), application modal (the user must respond before doing anything in the application), or modeless. For example, in the Clarion environment, the window that appears from the **F**ile menu's **O**pen selection is an application modal dialog box that requests the name of the file to open.

Control Fields and Input Focus

The objects placed in an APPLICATION or WINDOW structure are "control fields." "Control" is a standard Windows term used to refer to any screen object—command buttons, text entry fields, radio buttons, list boxes, etc. In most DOS programs, the term "field" is usually used to refer to these objects. In this document, the terms "control" and "field" are generally interchangeable.

Controls appear only in MENUBARs, TOOLBARs, or WINDOW structures. Controls are available to the user to select and/or edit the data they contain only when it has "input focus." This occurs when the user uses the TAB key, the mouse, or an accelerator key combination to highlight the control.

A WINDOW also has "input focus" when it is the top WINDOW in the currently active execution thread. Since Clarion for Windows allows multi-threaded programs, the concept of which WINDOW currently has focus is important. Only the thread whose uppermost WINDOW has focus is active. The user may edit data in the WINDOW's control fields only when it has focus.

Field Equate Labels

In WINDOW structures, every control field with a USE variable is assigned a field number by the compiler. By default, these field numbers begin with one (1) and are assigned to controls in the order they appear in the WINDOW structure code. The actual assigned numbers can be overridden by the second parameter of the USE attribute. The order of appearance in code determines the "natural" selection order of control fields for the ACCEPT structure (which may be altered with the SELECT statement). The

order of appearance in code is independent of the control's placement on the screen. Therefore, there is not necessarily any correlation between a control's position on screen and the field number assigned by the compiler.

There are a number of statements that use these field numbers as parameters. It would be very tedious to "hard code" these numbers in order to use these statements. Therefore, Clarion provides a mechanism to address this problem: Field Equate Labels.

Field Equate Labels always begin with a question mark (?) followed by the name of the control's USE variable. The leading question mark indicates to the compiler a Field Equate Label. They are very similar to normal EQUATE compiler directives. The compiler substitutes the field number for the Field Equate Label at compile time. This makes it unnecessary to know field numbers in advance.

Field Equate Labels for USE variables which are array elements always begin with a question mark (?) followed by the name of the USE variable followed by an underscore and a number (?ArrayField_1). Array elements from the same array are incrementally numbered beginning with one (1) for each element used in the same structure (?ArrayField_1, ?ArrayField_2, ...). Multi-dimensioned arrays are treated similarly (?ArrayField_1_1, ?ArrayField_1_2, ...).

Two or more controls with exactly the same USE variable in one WINDOW or APPLICATION structure would create the same Field Equate Label for all. Therefore, when the compiler encounters this condition, all Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference. It also allows you to deliberately create this condition to display the contents of the variable in multiple controls using different display pictures. Some fields may have USE variables that can only be Field Equate Labels (a unique label with a leading question mark). This provides a way of referencing these fields in code statements.

In APPLICATION structures, every menu selection in the MENUBAR, and every control with a USE variable placed in the TOOLBAR, is assigned a number by the compiler. By default, these numbers begin with negative one (-1) and are decremented by one (1) in the order the menu selections and controls appear in the APPLICATION structure code.

Window Structures

APPLICATION (declare an MDI frame window)

```
label   APPLICATION('title') [,AT( )] [,CENTER] [,SYSTEM] [,MAX] [,ICON( )] [,STATUS( )] [,HLP( )]
            [,CURSOR( )] [,TIMER( )] [,ALRT( )] [,ICONIZE] [,MAXIMIZE] [,MASK] [,FONT( )]
            [,MSG( )] [,IMM] [,AUTO] [,  | HSCROLL   | ] [,| DOUBLE   | ]
                                         | VSCROLL   |    | NOFRAME |
                                         | HVSCROLL  |    | RESIZE   |
        [ MENUBAR
            multiple menu and/or item declarations
          END ]
        [ TOOLBAR
            multiple control field declarations
          END ]
        END
```

APPLICATION	Declares a Multiple Document Interface (MDI) frame.
label	A valid Clarion label. A *label* is required on the APPLICATION statement.
title	Specifies the title text for the application window.
AT	Specifies the initial size and location of the application window. If omitted, default values are selected by the runtime library.
CENTER	Specifies that the window's initial position is centered in the screen by default. This attribute takes effect only if at least one parameter of the **AT** attribute is omitted.
SYSTEM	Specifies the presence of a system menu.
MAX	Specifies the presence of a maximize control.
ICON	Specifies the presence of a minimize control, and names a file or standard icon identifier for the icon displayed when the window is minimized.
STATUS	Specifies the presence of a status bar at the base of the application window.
HLP	Specifies the "Help ID" associated with the APPLICATION window and provides the default for any child windows.
CURSOR	Specifies a mouse cursor to be displayed when the mouse is positioned over the APPLICATION window. If omitted, the Windows default cursor is used.
TIMER	Specifies periodic timed event generation.
ALRT	Specifies "hot" keys active for the APPLICATION.
ICONIZE	Specifies the APPLICATION is opened as an icon.

MAXIMIZE	Specifies the APPLICATION is maximized when opened.
MASK	Specifies pattern input editing mode of all ENTRY controls in the TOOLBAR.
FONT	Specifies the default font for all controls in the toolbar.
MSG	Specifies a string constant containing the default text to display in the status bar for all controls in the APPLICATION.
IMM	Specifies the window generates events whenever it is moved or resized.
AUTO	Specifies all toolbar controls' USE variables re-display on screen each time through the ACCEPT loop.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the application frame when any portion of a child window lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the application frame when any portion of a child window lies vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the application frame when any portion of a child window lies outside the visible area.
DOUBLE	Specifies a double-width frame around the window. A window with this type of frame may not be resized.
NOFRAME	Specifies a window with no frame. A window with this type of frame may not be resized.
RESIZE	Specifies a thick frame around the window which does allow window resizing.
MENUBAR	Defines the menu structure (optional). The menu specified in an APPLICATION is the "Global menu."
TOOLBAR	Defines a toolbar structure (optional). The toolbar specified in an APPLICATION is the "Global toolbar."

APPLICATION declares a Multiple Document Interface (MDI) frame window. MDI is a part of the standard Windows interface, and is used by Windows applications to present several "views" in different windows. This is a way of organizing and grouping these. The MDI frame window (APPLICATION structure) acts as a "parent" for all the MDI "child" windows (WINDOW structures with the MDI attribute). These MDI "child" windows are clipped to the APPLICATION frame and automatically moved when the frame is moved, and can be totally concealed by minimizing the parent.

There may be only one APPLICATION window open at any time in a Clarion Windows program, and it must be opened before any MDI "child"

windows may be opened. However, non-MDI windows may be opened before or after the APPLICATION is opened, and may be on the same execution thread as the APPLICATION.

An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START function can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

A "conventional" APPLICATION window would have the ICON, MAX, STATUS, RESIZE, and SYSTEM attributes. This creates an application frame window with minimize and maximize buttons, a status bar, a resizable frame, and a system menu. It would also have a MENUBAR structure containing the global menu items, and may have a TOOLBAR with "shortcuts" to global menu items. These attributes create a standard Windows look and feel for the application frame.

An APPLICATION window may not contain controls except within its MENUBAR and TOOLBAR structures, and cannot be used for any output. For output, document windows or dialog boxes are required (defined using the WINDOW structure).

When the APPLICATION window is first opened, it remains hidden until the first DISPLAY statement or ACCEPT loop is encountered. This enables any changes to be made to the appearance before it is displayed. For example, the caption or size can be adjusted via runtime property assignment.

Events Generated:

EVENT:PreAlertKey
: The user pressed an ALRT attribute hot key.

EVENT:AlertKey
: The user pressed an ALRT attribute hot key.

EVENT:CloseWindow
: The window is closing.

EVENT:CloseDown
: The application is closing.

EVENT:OpenWindow
: The window is opening.

EVENT:LoseFocus
: The window is losing focus to another thread.

EVENT:GainFocus
: The window is gaining focus from another thread.

EVENT:Suspend
: The window still has input focus but is giving control to another thread to process timer events.

EVENT:Resume
: The window still has input focus and is regaining control from an EVENT:Suspend.

EVENT:Timer	The TIMER attribute has triggered.
EVENT:Move	The user is moving the window. CYCLE aborts the move.
EVENT:Moved	The user has moved the window.
EVENT:Size	The user is resizing the window. CYCLE aborts the resize.
EVENT:Sized	The user has resized the window.
EVENT:Restore	The user is restoring the window's previous size. CYCLE aborts the resize.
EVENT:Restored	The user has restored the window's previous size.
EVENT:Maximize	The user is maximizing the window. CYCLE aborts the resize.
EVENT:Maximized	The user has maximized the window.
EVENT:Iconize	The user is minimizing the window. CYCLE aborts the resize.
EVENT:Iconized	The user has minimized the window.
EVENT:Completed	AcceptAll (non-stop) mode has finished processing all the window's controls.
EVENT:DDErequest	A client has requested a data item from this Clarion DDE server application.
EVENT:DDEadvise	A client has requested continuous updates of a data item from this Clarion DDE server application.
EVENT:DDEexecute	A client has executed a DDEEXECUTE statement to this Clarion DDE server application.
EVENT:DDEpoke	A client has sent unsolicited data to this Clarion DDE server application.
EVENT:DDEdata	A DDE server has supplied an updated data item to this Clarion client application.
EVENT:DDEclose	A DDE server has terminated the DDE link to this Clarion client application.

Example:

```
  !An MDI application frame window with system menu, minimize and maximize
  ! buttons, a status bar, scroll bars, and a resizable frame, containing the
  ! main menu and toolbar for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          MENU('&File'),USE(?FileMenu)
            ITEM('&Open...'),USE(?OpenFile)
            ITEM('&Close'),USE(?CloseFile),DISABLE
            ITEM('E&xit'),USE(?MainExit)
          END
          MENU('&Edit'),USE(?EditMenu)
            ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
            ITEM('&Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
            ITEM('&Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
          END
          MENU('&Window'),STD(STD:WindowList),LAST
            ITEM('&Tile'),STD(STD:TileWindow)
            ITEM('&Cascade'),STD(STD:CascadeWindow)
            ITEM('&Arrange Icons'),STD(STD:ArrangeIcons)
          END
          MENU('&Help'),USE(?HelpMenu)
            ITEM('&Contents'),USE(?HelpContents),STD(STD:HelpIndex)
            ITEM('&Search...'),USE(?HelpSearch),STD(STD:HelpSearch)
            ITEM('&How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
            ITEM('&About MyApp...'),USE(?HelpAbout)
          END
        END
        TOOLBAR
          BUTTON('E&xit'),USE(?MainExitButton)
          BUTTON('&Open'),USE(?OpenButton),ICON(ICON:Open)
        END
      END
  CODE
  OPEN(MainWin)              !Open APPLICATION
  ACCEPT                     !Display APPLICATION and accept user input
    CASE ACCEPTED()          !Which control was chosen?
    OF ?OpenFile             !Open... menu selection
    OROF ?OpenButton         !Open button on toolbar
      START(OpenFileProc)    !Start new execution thread
    OF ?MainExit             !Exit menu selection
    OROF ?MainExitButton     !Exit button on toolbar
      BREAK                  !Break ACCEPT loop
    OF ?HelpAbout            !About... menu selection
      HelpAboutProc          !Call application information procedure
    END
  END
  CLOSE(MainWin)             !Close APPLICATION
```

WINDOW (declare a dialog window)

```
label   WINDOW('title') [,AT( )] [,CENTER] [,SYSTEM] [,MAX] [,ICON( )] [,STATUS( )] [,HLP( )]
            [,CURSOR( )] [,MDI] [,MODAL] [,MASK] [,FONT( )] [,GRAY][,TIMER( )] [,ALRT( )]
            [,ICONIZE] [,MAXIMIZE] [,MSG( )] [,TOOLBOX][,PALETTE( )] [,DROPID( )] [,IMM]
            [,AUTO] [, | HSCROLL    | ] [, | DOUBLE   | ]
                       | VSCROLL    |    | NOFRAME |
                       | HVSCROLL   |    | RESIZE   |
          [ MENUBAR
              menus and/or items
            END ]
          [ TOOLBAR
              controls
            END ]
            controls
        END
```

WINDOW	Declares a document window or dialog box.
label	A valid Clarion label. A *label* is required.
title	A string constant containing the window's title text.
AT	Specifies the initial size and location of the window. If omitted, default values are selected by the runtime library.
CENTER	Specifies that the window's initial position is centered on screen relative to its parent window, by default. This attribute takes effect only if at least one parameter of the AT attribute is omitted.
SYSTEM	Specifies the presence of a system menu.
MAX	Specifies the presence of a maximize control.
ICON	Specifies the presence of a minimize control, and names a file or standard icon identifier for the icon displayed when the window is minimized.
STATUS	Specifies the presence of a status bar for the window.
HLP	Specifies the "Help ID" associated with the window.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the window. This cursor is inherited by the WINDOW's controls unless overridden.
MDI	Specifies that the window conforms to normal MDI child-window behavior.
MODAL	Specifies the window is "system modal" and must be closed before the user may do anything else.
MASK	Specifies pattern input editing mode of all ENTRY controls in this window.
FONT	Specifies the default font for all controls in this window.

GRAY	Specifies that the window has a gray background for use with 3-D look controls.
TIMER	Specifies periodic timed event generation.
ALRT	Specifies "hot" keys active when the window has focus.
ICONIZE	Specifies the window is opened as an icon.
MAXIMIZE	Specifies the window is maximized when opened.
MSG	Specifies a string constant containing the default text to display in the status bar for all controls in the window.
TOOLBOX	Specifies the window is "always on top" and its controls never retain focus.
PALETTE	Specifies the number of hardware colors used for graphics in the window.
DROPID	Specifies the window may serve as a drop target for drag-and-drop actions.
IMM	Specifies the window generates events whenever it is moved or resized.
AUTO	Specifies all window controls' USE variables re-display on screen each time through the ACCEPT loop.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the window when any scrollable portion of the window lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the window when any scrollable portion of the window lies vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the window when any scrollable portion of the window lies outside the visible area.
DOUBLE	Specifies a double-width frame around the window. A window with this type of frame may not be resized.
NOFRAME	Specifies a window with no frame. A window with this type of frame may not be resized.
RESIZE	Specifies a thick frame around the window, which does allow window resizing.
MENUBAR	Defines a menu structure (optional).
menus and/or items	MENU and/or ITEM declarations that define the menu selections.
TOOLBAR	Defines a toolbar structure (optional).
controls	Control field declarations that define tools available on the TOOLBAR, or the control fields in the WINDOW.

A **WINDOW** declares a document window or dialog box which may contain controls, and may be used to display output to the user. When the WINDOW is first opened, it remains hidden until the first DISPLAY statement or ACCEPT loop is encountered. This enables any changes to be made to the appearance before it is displayed. For example, the caption or size can be adjusted via runtime property assignment. Any previously opened WINDOW on the same execution thread is disabled.

A WINDOW automatically receives a single-width border frame unless one of the DOUBLE, NOFRAME, or RESIZE attributes are specified. Screen coordinates are measured in dialog units. A dialog unit is defined as one-quarter the average character width and one-eighth the average character height of the font specified in the WINDOW's FONT attribute (or the system font, if no FONT attribute is specified on the WINDOW).

A WINDOW with the MODAL attribute is system modal; it takes exclusive control of the computer. This means that any other progam running in the background halts its execution until the MODAL WINDOW is closed. Therefore, the MODAL attribute should be used only when absolutely necessary. Also, the RESIZE attribute is ignored, and the WINDOW cannot be moved when the MODAL attribute is present.

A WINDOW without the MDI attribute, when opened in an MDI program on an MDI execution thread, is application modal. This means that the user must respond before moving to any other window in the application. The user may, however, move to any other program running in Windows at the time. Non-MDI windows may be opened either before or after an APPLICATION is opened, and may be on the same execution thread as the APPLICATION or any MDI child window (application modal) or their own thread (not application modal).

A WINDOW with the MDI attribute is an MDI "child" window. MDI "child" windows are clipped to the APPLICATION frame and automatically moved when the frame is moved, and can be totally concealed by minimizing the parent APPLICATION. MDI "child" windows are modeless; the user may change to the top window of another execution thread, within the same application or any other application running in Windows, at any time. An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START function can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

The MENUBAR specified in a WINDOW with the MDI attribute is automatically merged into the "Global menu" (from the APPLICATION) when the WINDOW receives focus unless either the WINDOW's or APPLICATION's MENUBAR has the NOMERGE attribute. A MENUBAR specified in a WINDOW without the MDI attribute is never merged into the "Global menu"—it always appears in the window itself.

The TOOLBAR specified in a WINDOW with the MDI attribute is automatically merged into the "Global toolbar" (from the APPLICATION) when the WINDOW receives focus, unless either the WINDOW's or APPLICATION's TOOLBAR has the NOMERGE attribute. The toolbar specified in a WINDOW without the MDI attribute is never merged into the "Global toolbar"—it always appears in the window itself.

A WINDOW with the TOOLBOX attribute is automatically "always on top" and its controls do not retain focus (just as if they all had the SKIP attribute). This creates a window whose controls all behave in the same manner as controls in the toolbar. Normally, a WINDOW with the TOOLBOX attribute would be executed in its own thread.

Events Generated:

EVENT:PreAlertKey
: The user pressed an ALRT attribute hot key.

EVENT:AlertKey
: The user pressed an ALRT attribute hot key.

EVENT:CloseWindow
: The window is closing.

EVENT:CloseDown
: The application is closing.

EVENT:OpenWindow
: The window is opening.

EVENT:LoseFocus
: The window is losing focus to another thread.

EVENT:GainFocus
: The window is gaining focus from another thread.

EVENT:Suspend
: The window still has input focus but is giving control to another thread to process timer events.

EVENT:Resume
: The window still has input focus and is regaining control from an EVENT:Suspend.

EVENT:Timer
: The TIMER attribute has triggered.

EVENT:Move
: The user is moving the window. CYCLE aborts the move.

EVENT:Moved
: The user has moved the window.

EVENT:Size
: The user is resizing the window. CYCLE aborts the resize.

EVENT:Sized
: The user has resized the window.

EVENT:Restore
: The user is restoring the window's previous size. CYCLE aborts the resize.

EVENT:Restored
: The user has restored the window's previous size.

EVENT:Maximize
: The user is maximizing the window. CYCLE aborts the resize.

EVENT:Maximized	The user has maximized the window.
EVENT:Iconize	The user is minimizing the window. CYCLE aborts the resize.
EVENT:Iconized	The user has minimized the window.
EVENT:Completed	AcceptAll (non-stop) mode has finished processing all the window's controls.
EVENT:DDErequest	A client has requested a data item from this Clarion DDE server application.
EVENT:DDEadvise	A client has requested continuous updates of a data item from this Clarion DDE server application.
EVENT:DDEexecute	A client has executed a DDEEXECUTE statement to this Clarion DDE server application.
EVENT:DDEpoke	A client has sent unsolicited data to this Clarion DDE server application.
EVENT:DDEdata	A DDE server has supplied an updated data item to this Clarion client application.
EVENT:DDEclose	A DDE server has terminated the DDE link to this Clarion client application.

Example:

```
 !MDI child window with system menu, minimize and maximize buttons, status bar,
 ! scroll bars, a resizable frame, with menu and toolbar which are merged into the
 !application's menubar and toolbar:
MDIChild  WINDOW('Child  One'),MDI,SYSTEM,MAX,ICON('Icon.ICO'),STATUS,HVSCROLL,RESIZE
            MENUBAR
              MENU('File'),USE(?FileMenu)
                ITEM('Close'),USE(?CloseFile)
              END
              MENU('Edit'),USE(?EditMenu)
                ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
                ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
                ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
                ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
              END
            END
            TOOLBAR
              BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut)
              BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy)
              BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste)
            END
            TEXT,HVSCROLL,USE(Pre:Field)
            BUTTON('&OK'),USE(?Exit),DEFAULT
          END

 !Non-MDI, system menu, maximize button, status bar, non-resizable frame,
NonMDI  WINDOW('Dialog  Window'),SYSTEM,MAX,STATUS
          TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
          BUTTON('&OK'),USE(?Exit),DEFAULT
        END

 !System-modal window with non-resizable frame, with only a message and Ok button:
ModalWin  WINDOW('Modal  Window'),MODAL
            IMAGE(ICON:Exclamation)
            STRING('An ERROR has occurred')
            BUTTON('&OK'),USE(?Exit),DEFAULT
          END
```

APPLICATION and WINDOW Attributes

ALRT (set window "hot" keys)

ALRT(*keycode***)**

ALRT Specifies a "hot" key active while the APPLICATION or WINDOW has focus.

keycode A numeric constant keycode or keycode EQUATE.

The **ALRT** attribute specifies a "hot" key active while the APPLICATION or WINDOW has focus. When the user presses an ALRT "hot" key for the APPLICATION or WINDOW, two field-independent events, EVENT:PreAlertKey and EVENT:AlertKey, are generated. If the code executes a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the EVENT:AlertKey, preventing library's default action on the alerted keypress for the window.

You may have multiple ALRT attributes on one APPLICATION or WINDOW. The ALERT statement and the ALRT attribute of a window or control are completely separate. This means that clearing ALERT keys has no effect on any keys alerted by ALRT attributes.

Example:

```
Screen  WINDOW,ALRT(F10Key),ALRT(F9Key)            !F10 and F9 alerted
          LIST,AT(109,48,50,50),USE(?List),FROM(Que),IMM
          BUTTON('&Ok'),AT(111,108,,),USE(?Ok)
          BUTTON('&Cancel'),AT(111,130,,),USE(?Cancel)
        END
  CODE
  OPEN(Screen)
  ACCEPT
    CASE EVENT()
    OF EVENT:PreAlertKey            !Pre-check alert events
      IF FOCUS() <> ?LIST             !Allow execution only on the list
        CYCLE                         !Terminate alert processing on other controls
      END
    OF EVENT:AlertKey               !Alert processing
      CASE KEYCODE()
      OF F9Key                      !Check for F9
        F9HotKeyProc                !Call hot key procedure
      OF F10Key                     !Check for F10
        F10HotKeyProc               !Call hot key procedure
      END
    END
  END
```

AT (set window position and size)

AT([*x*] [,*y*] [,*width*] [,*height*]**)**

AT	Specifies the initial position and size of the window.
x	An integer constant or constant expression that specifies the initial horizontal position of the top left corner. If omitted, the runtime library provides a default value.
y	An integer constant or constant expression that specifies the initial vertical position of the top left corner. If omitted, the runtime library provides a default value.
width	An integer constant or constant expression that specifies the initial width. If omitted, the runtime library provides a default value.
height	An integer constant or constant expression that specifies the initial height. If omitted, the runtime library provides a default value.

The **AT** attribute defines the initial position and size of an APPLICATION or WINDOW. If any parameter is omitted, the runtime library provides a default value. The *x* and *y* parameters are relative to the top left hand corner of the video screen when the AT attribute is placed on an APPLICATION structure, or a WINDOW without the MDI attribute that is opened before any APPLICATION structure is opened by the program. They are relative to the top left hand corner of the APPLICATION when the AT attribute is placed on a WINDOW with the MDI attribute, or a WINDOW without the MDI attribute opened after an APPLICATION structure has been opened.

The *width* and *height* parameters specify the size of the "client area" or "workspace" of an APPLICATION. This is the area below the MENUBAR and above the status bar which defines the area in which the TOOLBAR is placed and MDI "child" windows are opened. On a WINDOW, they specify the size of the "workspace" which may contain control fields.

The values contained in the *x*, *y*, *width*, and *height* parameters are mreasured in to dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
WinOne  WINDOW,AT(0,0,380,200),MDI   !top left corner, relative to app frame
        END

WinTwo  WINDOW,AT(0,0,380,200)       !Top left corner, relative to video screen
        END
```

AUTO (set USE variable automatic re-display)

AUTO

The **AUTO** attribute specifies all window and toolbar controls' USE variables re-display on screen each time through the ACCEPT loop. This incurs some overhead, but ensures the data displayed is current, without requiring explicit DISPLAY statements.

Example:

```
WinOne  WINDOW,AT(,,380,200),MDI,CENTER,AUTO   !All controls values always display
          !controls
        END
  CODE
  ACCEPT                  !ACCEPT automatically re-dislays changed USE variables
  END
```

CENTER (set position and size)

CENTER

The **CENTER** attribute indicates that the window's default width and height are centered. A WINDOW structure with the MDI attribute is centered on the APPLICATION. An APPLICATION structure is centered on the screen. A non-MDI WINDOW is centered on its parent (the window currently with focus when the non-MDI WINDOW is opened).

This attribute has no meaning unless at least one parameter of the AT attribute is omitted. This means that the CENTER attribute provides a default value for any omitted AT parameter.

Example:

```
 !Window centered relative to application frame:
WinOne  WINDOW,AT(,,380,200),MDI,CENTER
        END

 !Window centered relative to its parent:
WinTwo  WINDOW,AT(,,380,200),CENTER
        END
```

CURSOR (set mouse cursor type)

CURSOR(*file*)

CURSOR Specifies a mouse cursor to display for the window.

file A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the window. This cursor is inherited by the controls in the window unless overridden.

The Windows standard mouse cursors contained in EQUATES.CLW are:

CURSOR:None	No mouse cursor
CURSOR:Arrow	The normal windows arrow cursor
CURSOR:IBeam	A capital “I” like a steel I-beam
CURSOR:Wait	An hourglass
CURSOR:Cross	A large plus sign
CURSOR:UpArrow	A vertical arrow
CURSOR:Size	A four-headed arrow
CURSOR:Icon	A box within a box
CURSOR:SizeNWSE	A double-headed arrow slanting left
CURSOR:SizeNESW	A double-headed arrow slanting right
CURSOR:SizeWE	A double-headed horizontal arrow
CURSOR:SizeNS	A double-headed vertical arrow
CURSOR:DragWE	A double-headed horizontal arrow

Example:

```
 !Window with Windows-standard large plus sign cursor
WinOne  WINDOW,CURSOR(CURSOR:Cross)
        END

 !Window with custom cursor
WinTwo  WINDOW,CURSOR('CUSTOM.CUR')
        END
```

DOUBLE, NOFRAME, RESIZE (set window border)

DOUBLE
NOFRAME
RESIZE

The **DOUBLE**, **NOFRAME**, and **RESIZE** attributes specify a WINDOW or APPLICATION border frame style other than the default single-width border. The **DOUBLE** attribute places a double-width border around the window and the **NOFRAME** attribute places no border on the window. A window with these frame types may not be resized.

The **RESIZE** attribute places a thick border frame around the window. This is the only type that allows the user to dynamically resize the window. RESIZE is ignored on any WINDOW with the MODAL attribute.

The RESIZE frame type is normally used on APPLICATION structures and WINDOW structures used as document windows, not dialog boxes. NOFRAME is usually used on “hidden” windows used only to activate an ACCEPT loop. DOUBLE is a common dialog bix frame type.

Example:

```
 !A Window with a single-width border:
Win1 WINDOW
     END

 !A resizable Window:
Win2 WINDOW,RESIZE
     END

 !A Window with a double-width border:
Win3 WINDOW,DOUBLE
     END

 !A Window without a border:
Win4 WINDOW,NOFRAME
     END
```

FONT (set window default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default display font for the window.
typeface	A string constant containing the name of the font. If omitted, the system font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on a WINDOW or APPLICATION structure specifies the default display font for all controls in the WINDOW or APPLICATION that do not have a FONT attribute. This is also the default font for newly created controls on the window, and is the font used by the SHOW and TYPE statements when writing to the window.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

```
FONT:thin        EQUATE (100)
FONT:regular     EQUATE (400)
FONT:bold        EQUATE (700)
FONT:italic      EQUATE (01000H)
FONT:underline   EQUATE (02000H)
FONT:strikeout   EQUATE (04000H)
```

Example:

```
 !A Window using 14 point Times New Roman
Win1 WINDOW,FONT('Times New Roman',14,00H)
     END

 !A Window using 14 point Times New Roman, Bold and Italic
Win2 WINDOW,FONT('Times New Roman',14,00H,FONT:italic+FONT:bold)
     END
```

GRAY (set 3-D look background)

GRAY

The **GRAY** attribute indicates that the WINDOW has a gray background, suitable for use with three-dimensional dialog controls. All controls on a WINDOW with the GRAY attribute are automatically given a three-dimensional appearance. Controls in a TOOLBAR are always automatically given a three-dimensional appearance, without the GRAY attribute.

This attribute is not valid on an APPLICATION structure.

The three-dimensional look may be disabled by SET3DLOOK.

Example:

```
 !A Window with 3-D controls
Win1 WINDOW,GRAY
     END
```

HLP (set window's on-line help identifier)

HLP(*helpID***)**

HLP	Specifies the *helpID* for the APPLICATION, WINDOW, or control.
helpID	A string constant specifying the key used to access the Help system. This may be either a Help keyword or a "context string."

The **HLP** attribute specifies the *helpID* for the APPLICATION or WINDOW. Help, if available, is automatically displayed by Windows whenever the user presses F1.

If the user presses F1 to request help when the APPLICATION window is foremost and no menus are active, the APPLICATION's *helpID* is used to locate the Help text. Otherwise, the library automatically uses the *helpID* of the active menu of uppermost control or window, searching up the hierarchy until an object with that *helpID* is found. The *helpID* of the APPLICATION is at the top of the hierarchy.

The *helpID* may contain a Help keyword or a "context string." A Help keyword is a keyword or phrase that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *helpID*, followed by a unique identifier (no spaces allowed) associated with exactly one help topic. When the user presses F1, the help file is opened at the specific topic associated with that "context string." If the tilde is missing, the *helpID* is assumed to be a help keyword.

Example:

```
 !A Window with a help context string:
Win1  WINDOW,HLP('~Win1Help')
      END

 !A Window with a help keyword:
Win2  WINDOW,HLP('Window One Help')
      END
```

HSCROLL, VSCROLL, HVSCROLL (set window scroll bars)

HSCROLL
VSCROLL
HVSCROLL

The **HSCROLL**, **VSCROLL**, and **HVSCROLL** parameters place scroll bars on an APPLICATION or WINDOW. HSCROLL adds a horizontal scroll bar to the bottom, VSCROLL adds a vertical scroll bar on the right side, and HVSCROLL adds both.

The vertical scroll bar allows a mouse to scroll up or down. The horizontal scroll bar allows a mouse to scroll left or right. The scroll bars appear whenever any scrollable portion of the APPLICATION or WINDOW lies outside the visible area on screen.

Example:

```
 !A Window with a horizontal scroll bar:
Win1 WINDOW,HSCROLL
     END

 !A Window with a vertical scroll bar:
Win2 WINDOW,VSCROLL
     END

 !A Window with both scroll bars:
Win2 WINDOW,HVSCROLL
     END
```

ICON (set window icon)

ICON([*file*] **)**

ICON Specifies an icon to display for the APPLICATION or WINDOW.

file A string constant containing the name of an .ICO file, or an EQUATE for the Windows-standard icon to display. The .ICO file is automatically linked into the .EXE as a resource.

The **ICON** attribute specifies an icon to display for the APPLICATION or WINDOW. On an APPLICATION or WINDOW, ICON also specifies the presence of a minimize control. The minimize control appears in the top right corner of the window as a downward pointing triangle (usually). When the user clicks the mouse on it, the window shrinks to an icon without halting its execution. When an APPLICATION or non-MDI WINDOW is minimized, the icon *file* is displayed in the operating system's desktop; when a WINDOW with the MDI attribute is minimized, the icon *file* is displayed in the APPLICATION.

EQUATE statements for the Windows-standard icons are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

```
ICON:None           No icon
ICON:Application
ICON:Question       ?
ICON:Exclamation    !
ICON:Asterisk       *
ICON:VCRtop         >>|
ICON:VCRrewind      <<
ICON:VCRback        <
ICON:VCRplay                >
ICON:VCRfastforward >>
ICON:VCRbottom      |<<
ICON:VCRlocate      ?
```

Example:

```
 !A Window with a minimize button:
WinOne  WINDOW,ICON('MyIcon.ICO')
        END

 !A Window with a minimize button:
WinTwo  WINDOW,ICON(ICON:Application)
        END
```

ICONIZE (set window open as icon)

ICONIZE

The **ICONIZE** attribute specifies the APPLICATION or WINDOW is opened minimized as the icon specified by the ICON attribute. When an APPLICATION or non-MDI WINDOW is minimized, the icon *file* is displayed in the operating system's desktop; when a WINDOW with the MDI attribute is minimized, the icon *file* is displayed in the APPLICATION.

Example:

```
 !A Window with a minimize button, opened as the icon:
Win2  WINDOW,ICON('MyIcon.ICO'),ICONIZE
      END
```

IMM (set immediate resize event notification)

IMM

The **IMM** attribute on a WINDOW or APPLICATION specifies immediate event generation whenever the user moves or resizes the window. It generates one the following events before the action is executed:

EVENT:Move
EVENT:Size
EVENT:Restore
EVENT:Maximize
EVENT:Iconize

If the code that handles these events executes a CYCLE statement, the action is not performed. This allows you to prevent the user from moving or resizing the window. Once the action has been performed, one or more of the following events are generated:

EVENT:Moved
EVENT:Sized
EVENT:Restored
EVENT:Maximized
EVENT:Iconized

Multiple post-action events are generated because some of the actions have multiple results. For example, if the user CLICKS on the maximize button, EVENT:Maximize is generated. If there is no CYCLE statement executed as a result of this event, the action is performed, then EVENT:Maximized, EVENT:Moved, and EVENT:Sized are generated. This occurs because the window has been maximized, which also moves and resizes it at the same time.

Example:

```
Win2  WINDOW('Some  Window'),AT(58,11,174,166),MDI,DOUBLE,MAX,IMM
        LIST,AT(109,48,50,50),USE(?List),FROM('Que'),IMM
        BUTTON('&Ok'),AT(111,108,,),USE(?Ok)
        BUTTON('&Cancel'),AT(111,130,,),USE(?Cancel)
      END
  CODE
  OPEN(Win2)
  ACCEPT
    CASE EVENT()
    OF EVENT:Move                 !Prevent user from moving window
      CYCLE
    OF EVENT:Maximized            !When Maximized
      ?List{PROP:Height} = 100    ! resize the list
    OF EVENT:Restored             !When Restored
      ?List{PROP:Height} = 50     ! resize the list
    END
  END
```

MASK (set pattern editing data entry)

MASK

The **MASK** attribute specifies pattern input editing mode of all controls in this window. This means that, as the user types in data, each character is automatically validated against the control's picture for proper input (numbers only in numeric pictures, etc.). This forces the user to enter data in the format specified by the control's display picture.

If omitted, Windows free-input is allowed in the controls. Free-input means the user's data is formatted to the control's picture only after entry. This allows users to enter data as they choose and it is automatically formatted to the control's picture after entry. If the user types in data in a format different from the control's picture, the libraries attempt to determine the format the user used, and convert the data to the control's display picture. For example, if the user types "January 1, 1995" into a control with a display picture of @D1, the runtime library formats the user's input to "1/1/95." This action occurs only after the user completes data entry and moves to another control. If the runtime library cannot determine what format the user used, it will not update the USE variable. It then beeps and leaves the user on the same control with the data they entered, to allow them to try again.

Example:

```
 !A Window with pattern input editing enabled
Win2 WINDOW,MASK
     END
```

MAX (set maximize control)

MAX

The **MAX** attribute specifies a maximize control on the APPLICATION or WINDOW. The maximize control appears in the top right corner of the window as a box containing either an upward pointing triangle, or an upward pointing triangle above a downward pointing triangle (in Windows 3.1). When the user clicks the mouse on it, an APPLICATION or non-MDI WINDOW expands to occupy the full screen, an MDI WINDOW expands to occupy the entire APPLICATION. Once expanded, the maximize control appears as an upward pointing triangle above a downward pointing triangle. Click the mouse on it again, and the window returns to its previous size and the maximize control appears as an upward pointing triangle.

Example:

```
 !A Window with a maximize button:
Win2 WINDOW,MAX
     END
```

MAXIMIZE (set window open maximized)

MAXIMIZE

The **MAXIMIZE** attribute specifies the APPLICATION or WINDOW is opened maximized.When maximized, an APPLICATION or non-MDI WINDOW expands to occupy the full screen, and an MDI WINDOW expands to occupy the entire APPLICATION. Once expanded, the maximize control appears as an upward pointing triangle above a downward pointing triangle (in Windows 3.1).

Example:

```
 !A Window with a maximize button, opened maximized:
Win2  WINDOW,MAX,MAXIMIZE
      END
```

MDI (set MDI child window)

MDI

The **MDI** attribute specifies a WINDOW structure that acts as a "child" window to the APPLICATION. MDI "child" windows are clipped to the APPLICATION frame—they scroll only within the boundaries set by the display size of the APPLICATION. MDI "child" windows are automatically moved when the APPLICATION frame is moved, and can be totally concealed by minimizing the APPLICATION. A WINDOW with the MDI attribute cannot be opened unless there is a currently open APPLICATION.

MDI "child" windows are modeless; the user may change to the top window of another execution thread, within the same application or any other application running in Windows, at any time. An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START function can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

A non-MDI WINDOW operates independently of any previously opened APPLICATION. It will, however, disable an APPLICATION if it or any of its MDI "child" windows are on the same execution thread. This makes a non-MDI window opened in an MDI program an "application modal" window which effectively disables the application while the user has the window open (unless it is opened in its own execution thread). It does not, however, prevent the user from changing to another application running under Windows.

Example:

```
 !An MDI child Window:
Win2  WINDOW,MDI
      END
```

MODAL (set system modal window)

MODAL

The **MODAL** attribute specifies the WINDOW is "system modal." This means that no other window (in the same or any other concurrent program) can receive focus while the MODAL window has focus—the MODAL window has exclusive control of the computer. MODAL windows are usually used for error messages, or messages which require immediate attention by the user, such as: "Please insert a disk in drive A:."

A WINDOW without the MODAL attribute, may be "application-modal" or "modeless." An application-modal window is a non-MDI window opened as the top window of an MDI execution thread. An application-modal window restricts the user from moving to another execution thread in the same application, but does not restrict them from changing to another Windows program.

A modeless window is an MDI "child" WINDOW (with the MDI attribute) without the MODAL attribute. From a modeless window, The top window on other execution threads may be selected by the mouse, keyboard, or menu commands. If so, the other window takes focus and becomes uppermost on the video display. Any window not on the top of its execution thread may not be selected to receive focus, even from a modeless window.

Example:

```
Win2 WINDOW,MODAL  !A system-modal Window
     END
```

MSG (set window status bar message)

MSG(*text***)**

MSG	Specifies *text* to display in the status bar.
text	A string constant containing the message to display in the status bar.

The **MSG** attribute on an APPLICATION or WINDOW structure specifies the *text* to display in the first zone of the status bar when the control with focus has no MSG attribute of its own.

Example:

```
WinOne  WINDOW,AT(0,0,160,400),MSG('Enter Data')          !Default MSG to use
          COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),MSG('Enter or Select')
          TEXT,AT(20,0,40,40),USE(E2)                     !Default MSG used
          ENTRY(@S8),AT(100,200,20,20),USE(E2)            !Default MSG used
          CHECK('&A'),AT(0,120,20,20),USE(?C7),MSG('On or Off')
          OPTION('Option 1'),USE(OptVar),MSG('Pick One or Two')
            RADIO('Radio 1'),AT(120,0,20,20),USE(?R1)
            RADIO('Radio 2'),AT(140,0,20,20),USE(?R2)
          END
        END
```

PALETTE (set number of hardware colors)

PALETTE(*colors***)**

PALETTE	Specifies the number of hardware colors displayed in the window.
colors	An integer constant specifying the number of hardware colors displayed in the window.

The **PALETTE** attribute on a WINDOW specifies how many colors in the hardware palette you want this window to use when it is the foreground window. This is only applicable in hardware modes where a palette is in use and spare colors (not reserved by the system) are available - in practice this means 256 color mode. This enforces a particular set of colors for the graphics. 24-bit color (16.7M) does not use a hardware palette. Values of PALETTE above 256 are not recommended.

Example:

```
WinOne  WINDOW,AT(0,0,160,400),PALETTE(256)           !Display 256-color
          IMAGE,AT(120,120,20,20),USE(ImageField)
        END
```

STATUS (set status bar)

STATUS([*widths*] **)**

STATUS	Specifies the presence of a status bar.
widths	A list of integer constants (separated by commas) specifying the size of each zone in the status bar. If omitted, the status bar has one zone the width of the window.

The **STATUS** attribute specifies the presence of a status bar at the base of the APPLICATION or WINDOW. The status bar of an MDI WINDOW is always displayed at the bottom of the APPLICATION. A WINDOW without the MDI attribute displays its status bar at the base of the WINDOW. If the STATUS attribute is not present on the APPLICATION or WINDOW, there is no status bar.

The status bar may be divided into multiple zones specified by the *widths* parameters. The size of each zone is specified in dialog units. A negative value indicates the zone is expandable, but has a minimum width indicated by the parameter's absolute value. If no *widths* parameters are specified, a single expanding zone with no minimum width is created, which is equivalent to a STATUS(-1).

The first zone of the status bar is always used to display MSG attributes. The MSG attribute string is displayed in the status bar as long as its control field still has input focus. A control or menu item without a MSG attribute causes the status bar to revert to its former state (either blank or displaying the text previously displayed in the zone).

Text may be placed in, or retrieved from, any zone of the status bar using the runtime property assignment syntax. The text remains present until replaced. The status bar configuration can also be changed dynamically by using the runtime property assignment syntax.

Example:

```
 !An APPLICATION with a one-zone status bar:
MainWin  APPLICATION,STATUS
        END

 !A WINDOW with a two-zone status bar:
Win1  WINDOW,STATUS(160,160)
     END
```

SYSTEM (set system menu)

SYSTEM

The **SYSTEM** attribute specifies the presence of a Windows system menu (also called the control menu) on the APPLICATION or WINDOW. This menu contains standard Windows menu selections, such as: Close, Minimize, Maximize (the window), and Switch To (another window). The actual selections available on a given window depend upon the attributes set for that window.

Example:

```
 !An APPLICATION with a system menu:
MainWin  APPLICATION,SYSTEM
         END
 !A WINDOW with a system menu:
Win1  WINDOW,SYSTEM
      END
```

TOOLBOX (set toolbox window behavior)

TOOLBOX

The **TOOLBOX** attribute specifies a WINDOW that is “always on top.” Neither the WINDOW nor its controls retain input focus. This creates control behavior as if all the controls in the WINDOW had the SKIP attribute. Normally, a WINDOW with the TOOLBOX attribute executes in its own execution thread to provide a set of tools to the window with input focus. The MSG attributes of the controls in the window appear in the status bar when the mouse cursor is positioned over the control.

Example:

```
 PROGRAM
MainWin APPLICATION('My Application')
          MENUBAR
            MENU('File'),USE(?FileMenu)
              ITEM('E&xit'),USE(?MainExit),LAST
            END
            MENU('Edit'),USE(?EditMenu)
              ITEM('Use Tools'),USE(?UseTools)
        . . .
Pre:Field       STRING(400)
UseToolsThread  BYTE
ToolsThread     BYTE
  CODE
  OPEN(MainWin)
  ACCEPT
    CASE ACCEPTED()
    OF ?MainExit
      BREAK
    OF ?UseTools
      UseToolsThread = START(UseTools)
  . .

UseTools PROCEDURE                      !A procedure that uses a toolbox
MDIChild WINDOW('Use Tools Window'),MDI
           TEXT,HVSCROLL,USE(Pre:Field)
           BUTTON('&OK'),USE(?Exit),DEFAULT
         END
  CODE
  OPEN(MDIChild)                       !Open the window
  DISPLAY                              ! and display it
  ToolsThread = START(Tools)           !Pop up the toolbox
  ACCEPT
    CASE EVENT()                               !Check for user-defined events
    OF 401h                                    ! posted by toolbox controls
      Pre:Field += ' ' & FORMAT(TODAY(),@D1)   ! append date to end of field
    OF 402h
      Pre:Field += ' ' & FORMAT(CLOCK(),@T1)   ! append time to end of field
    END
    CASE ACCEPTED()
    OF ?Exit
      POSTEVENT(400h,,ToolsThread)             !Signal to close tools window
      BREAK
  . .
  CLOSE(MDIChild)

Tools PROCEDURE   !The toolbox procedure
Win1  WINDOW('Tools'),TOOLBOX
        BUTTON('Date'),USE(?Button1)
        BUTTON('Time'),USE(?Button2)
      END
  CODE
  OPEN(Win1)
  ACCEPT
    IF EVENT() = 400h THEN BREAK.              !Check for close window signal
    CASE ACCEPTED()
    OF ?Button1
      POSTEVENT(401h,,UseToolsThread)          !Post datestamp signal
    OF ?Button2
      POSTEVENT(402h,,UseToolsThread)      !Post timestamp signal
  . .
  CLOSE(Win1)
```

TIMER (set periodic event)

TIMER(*period***)**

TIMER	Specifies a periodic event.
period	An integer constant or constant expression specifying the interval between timed events, in hundredths of a second. The maximum *period* you can specify is 6553 (a Windows limitation).

The **TIMER** attribute specifies generation of a periodic field-independent event whenever the time *period* passes. EQUATES.CLW contains EVENT:Timer which equates the timer-generated event. The FOCUS() function returns the number of the control that currently has focus at the time of the event.

Example:

```
RunClock PROCEDURE
ShowTime LONG

 !A WINDOW with a timed event occurring every second:
Win1  WINDOW,TIMER(100)
        STRING(@T4),USE(ShowTime)
      END
  CODE
  OPEN(Win1)
  ShowTime = CLOCK()
  ACCEPT
    CASE EVENT()
    OF EVENT:Timer
      ShowTime = CLOCK()
      DISPLAY
    END
  END
  CLOSE(Win1)
```

MENUBAR and TOOLBAR Structures

MENUBAR (declare a pulldown menu)

```
MENUBAR [, NOMERGE ]
   [ MENU( )
      [ ITEM( ) ]
      [ MENU( )
         [ ITEM( ) ]
       END ]
    END ]
   [ ITEM( ) ]
END
```

MENUBAR	Declares the menu for an APPLICATION or WINDOW.
NOMERGE	Specifies menu merging behavior.
MENU	A menu item with an associated drop box containing other menu selections.
ITEM	A menu item for selection.

The **MENUBAR** structure declares the pulldown menu selections displayed for an APPLICATION or WINDOW. MENUBAR must appear in the source code before any TOOLBAR or controls.

On an APPLICATION, the MENUBAR defines the Global menu selections for the program. These are active and available on all MDI “child” windows (unless the window’s own MENUBAR structure has the NOMERGE attribute). If the NOMERGE attribute is specified on the APPLICATION’s MENUBAR, then the menu is a local menu displayed only when no MDI child windows are open and there is no global menu.

On an MDI WINDOW, the MENUBAR defines menu selections that are automatically merged with the Global menu. Both the Global and the window’s menu selections are then active while the MDI “child” window has input focus. Once the window loses focus, its specific menu selections are removed from the Global menu. If the NOMERGE attribute is specified on an MDI WINDOW’s MENUBAR, the menu overwrites and replaces the Global menu.

On a non-MDI WINDOW, the MENUBAR is never merged with the Global menu. A MENUBAR on a non-MDI WINDOW always appears in the WINDOW, not on any APPLICATION which may have been previously opened.

Events generated by local menu items are sent to the WINDOW’s ACCEPT loop in the normal way. Events generated by global menu items are sent to the active event loop of the thread which opened the APPLICATION (in a normal multi-thread application this means the APPLICATION’s own

ACCEPT loop).

Dynamic changes to menu items which reference the currently active window affect only the currently displayed menu, even if global items are changed. Changes made to the Global menu items when the APPLICATION is the current window, or which reference the global APPLICATION window affect the global portions of all menus, whether already open or not.

When a WINDOW's MENUBAR is merged into an APPLICATION's MENUBAR, the global menu selections appear first, followed by the local menu selections, unless the FIRST or LAST attributes are specified on individual menu selections.

Example:

```
 !An MDI application frame window with main menu for the application:
MainWin APPLICATION('My Application')
          MENUBAR
            MENU('File'),USE(?FileMenu)
              ITEM('Open...'),USE(?OpenFile)
              ITEM('Close'),USE(?CloseFile),DISABLE
              ITEM('E&xit'),USE(?MainExit),LAST
            END
            MENU('Edit'),USE(?EditMenu)
              ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
              ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
              ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
            END
            MENU('Help'),USE(?HelpMenu),LAST
              ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
              ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
              ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
              ITEM('About MyApp...'),USE(?HelpAbout)
            END
          END
        END

 !An MDI child window with menu for the window, merged into the
 ! application's menubar:
MDIChild WINDOW('Child One'),MDI
           MENUBAR
             MENU('File'),USE(?FileMenu)          !Merges into File menu
               ITEM('Close'),USE(?CloseFile)      !Supercedes main menu selection
               ITEM('Pick...'),USE(?PickFile)     !Added to menu selections
             END
             MENU('Edit'),USE(?EditMenu)          !Merges into Edit menu
               ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo) !Added to menu
                                  !These items supercede main menu selections:
               ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
               ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
               ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
             END
             MENU('Window'),STD(STD:WindowList),LAST
               ITEM('Tile'),STD(STD:TileWindow)
               ITEM('Cascade'),STD(STD:CascadeWindow)
             END
           END
           TEXT,HVSCROLL,USE(Pre:Field)
           BUTTON('&OK'),USE(?Exit),DEFAULT
         END
```

```
 !An MDI window with its own menu, overwriting the main menu:
MDIChild2 WINDOW('Dialog Window'),MDI,SYSTEM,MAX,STATUS
            MENUBAR,NOMERGE
              MENU('File'),USE(?FileMenu)
                ITEM('Close'),USE(?CloseFile)
              END
              MENU('Edit'),USE(?EditMenu)
                ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
                ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
                ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
                ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
              END
            END
          TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
            BUTTON('&OK'),USE(?Exit),DEFAULT
          END

 !A non-MDI window with its own menu:
NonMDI  WINDOW('Dialog Window'),SYSTEM,MAX,STATUS
          MENUBAR
            MENU('File'),USE(?FileMenu)
              ITEM('Close'),USE(?CloseFile)
            END
            MENU('Edit'),USE(?EditMenu)
              ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
              ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
              ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
              ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
            END
          END
          TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
          BUTTON('&OK'),USE(?Exit),DEFAULT
        END
```

TOOLBAR (declare a tool bar)

```
TOOLBAR [,AT( )] [,CURSOR( )] [,FONT( )] [,NOMERGE]
        controls
END
```

TOOLBAR	Declares tools for an APPLICATION or WINDOW.
AT	Specifies the initial size of the toolbar. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the TOOLBAR. If omitted, the WINDOW or APPLICATION structure's CURSOR attribute is used, else the Windows default cursor is used.
FONT	Specifies the default display font for the controls in the TOOLBAR.
NOMERGE	Specifies tools merging behavior.
controls	Control field declarations that define the available tools.

The **TOOLBAR** structure declares the tools displayed for an APPLICATION or WINDOW. On an APPLICATION, the TOOLBAR defines the Global tools for the program. If the NOMERGE attribute is specified on the APPLICATION's TOOLBAR, the tools are local and are displayed only when no MDI child windows are open; there are no global tools. Global tools are active and available on all MDI "child" windows unless an MDI "child" window's TOOLBAR structure has the NOMERGE attribute. If so, the "child" window's tools overwrite the Global tools.

On an MDI WINDOW, the TOOLBAR defines tools that are automatically merged with the Global toolbar. Both the Global and the window's tools are then active while the MDI "child" window has input focus. Once the window loses focus, its specific tools are removed from the Global toolbar. If the NOMERGE attribute is specified on an MDI WINDOW's TOOLBAR, the tools overwrite and replace the Global toolbar. On a non-MDI WINDOW, the TOOLBAR is never merged with the Global menu. A TOOLBAR on a non-MDI WINDOW always appears in the WINDOW, not on any APPLICATION which may have been previously opened.

Events generated by local tools are sent to the WINDOW's ACCEPT loop in the normal way. Events generated by global tools are sent to the active event loop of the thread which opened the APPLICATION. In a normal multi-thread application, this means the APPLICATION's own ACCEPT loop.

TOOLBAR controls generate events in the normal manner. However, they do not keep the focus, and cannot be operated from the keyboard unless accelerator keys are provided. As soon as user interaction with a TOOLBAR control is done, focus returns to the window and local control which previously had it.

Dynamic changes to tools which reference the currently active window affect only the currently displayed toolbar, even if global tools are changed. Changes made to the Global toolbar when the APPLICATION is the current window, or which reference the global APPLICATION's window affect the global portions of all toolbars, whether already open or not.

When a WINDOW's TOOLBAR is merged into an APPLICATION's TOOLBAR, the global tools appear first, followed by the local tools. The toolbars are merged so that the fields in the WINDOW's toolbar begin just right of the position specified by the value of the width parameter of the APPLICATION TOOLBAR's AT attribute. The height of the displayed toolbar is the maximum height of the "tallest" tool, whether global or local. If any part of a control falls below the bottom, the height is increased accordingly.

Example:

```
 !An MDI application frame window containing the
 ! main menu and toolbar for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
          MENUBAR
            ITEM('E&xit'),USE(?MainExit)
          END
          TOOLBAR
            BUTTON('Exit'),USE(?MainExitButton)
          END
        END
 !An MDI child window with toolbar for the window, merged into the
 ! application's toolbar:
MDIChild  WINDOW('Child One'),MDI
            TOOLBAR
              BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
              BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
              BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
            END
            TEXT,HVSCROLL,USE(Pre:Field)
            BUTTON('&OK'),USE(?Exit),DEFAULT
          END
 !An MDI window with its own toolbar, overwriting the main toolbar:
MDIChild2 WINDOW('Dialog Window'),MDI,SYSTEM,MAX,STATUS
            TOOLBAR,NOMERGE
              BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
              BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
              BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
            END
            TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
            BUTTON('&OK'),USE(?Exit),DEFAULT
          END
 !A non-MDI window with its own toolbar:
NonMDI  WINDOW('Dialog Window'),SYSTEM,MAX,STATUS
          TOOLBAR
            BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
            BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
            BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
          END
          TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
          BUTTON('&OK'),USE(?Exit),DEFAULT
        END
```

MENUBAR and TOOLBAR Attributes

CURSOR (set toolbar mouse cursor type)

CURSOR(*file*)

CURSOR Specifies a mouse cursor to display for the TOOLBAR.

file A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the TOOLBAR. This cursor is inherited by the controls in the toolbar unless overridden.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital “I” like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow
CURSOR:DragWE	Double-headed horizontal arrow

Example:

```
 !Toolbar with large plus sign cursor
WinOne  WINDOW
          TOOLBAR,CURSOR('CURSOR:Cross')
            BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
            BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
            BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
          END
        END
```

FONT (set toolbar default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default display font for the TOOLBAR.
typeface	A string constant containing the name of the font. If omitted, the system font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on a TOOLBAR structure specifies the default display font for all controls in the TOOLBAR that do not have a FONT attribute. The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
Win1  WINDOW    !A toolbar using 14 point Times New Roman
        TOOLBAR,FONT('Times New Roman',14,00H)
          BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
          BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
        END
      END

Win2  WINDOW  !14 point Times New Roman, Bold and Italic
        TOOLBAR,FONT('Times New Roman',14,00H,FONT:italic+FONT:bold)
          BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
          BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
        END
      END
```

NOMERGE (set merging behavior)

NOMERGE

The **NOMERGE** attribute indicates that the MENUBAR or TOOLBAR on a WINDOW should not be merged with the Global menu or toolbar.

The NOMERGE attribute on an APPLICATION's MENUBAR indicates that the menu is local and to be displayed only when no MDI "child" windows are open and that there is no Global menu. The NOMERGE attribute on an APPLICATION's TOOLBAR indicates that the tools are local and to be displayed only when no MDI "child" windows are open and that there are no Global tools.

Without the NOMERGE attribute, an MDI WINDOW's menu and toolbar are automatically merged with the global menu and toolbar, and then displayed in the APPLICATION menu and toolbar. When NOMERGE is specified, the WINDOW's menu and toolbar overwrite the Global menu and toolbar. The menu and toolbar displayed when the WINDOW has focus are only the WINDOW's own menu and toolbar. However, they are still displayed on the APPLICATION.

A MENUBAR or TOOLBAR specified in a non-MDI WINDOW is never merged with the Global menu or toolbar—they appear in the WINDOW.

Example:

```
 !An MDI application frame window with local-only menu and toolbar:
MainWin APPLICATION('My  Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS
          MENUBAR,NOMERGE
            ITEM('E&xit'),USE(?MainExit)
          END
          TOOLBAR,NOMERGE
            BUTTON('Exit'),USE(?MainExitButton)
          END
        END

 !MDI window with its own menu and toolbar, overwriting the application's:
MDIChild  WINDOW('Dialog  Window'),MDI,SYSTEM,MAX,STATUS
            MENUBAR,NOMERGE
              MENU('Edit'),USE(?EditMenu)
                ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
                ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
                ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
              END
            END
            TOOLBAR,NOMERGE
              BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
              BUTTON('Copy'),USE(?CopyButton),STD(STD:Copy)
              BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
            END
            TEXT,HVSCROLL,USE(Pre:Field),MSG('Enter some text here')
            BUTTON('&OK'),USE(?Exit),DEFAULT
          END
```

MENUBAR Controls

MENU (declare a menu box)

```
MENU(text) [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,STD( )] [,RIGHT] [,DISABLE]
                [, | FIRST | ]
                   | LAST  |
```

MENU	Declares a menu box within a MENUBAR.
text	A string constant containing the display text for the menu selection.
USE	A field equate label to reference the menu selection in executable code.
KEY	Specifies an integer constant or keycode equate that immediately opens the menu.
MSG	Specifies a string constant containing the text to display in the status bar when the menu is pulled down.
HLP	Specifies a string constant containing the help system identifier for the menu.
STD	Specifies an integer constant or equate that identifies a "Windows standard behavior" for the menu.
RIGHT	Specifies the MENU appears at the far right of the action bar.
FIRST	Specifies the MENU appears at the left or top of the menu when merged.
LAST	Specifies the MENU appears at the right or bottom of the menu when merged.
DISABLE	Specifies the menu appears dimmed when the WINDOW or APPLICATION is first opened.

MENU declares a drop-down or cascading menu box structure within a MENUBAR structure. When the MENU is selected, the MENU and/or ITEM statements within the structure are displayed in a menu box. A MENU is not required to have any MENUs or ITEMs in it. A menu box usually appears (drops down) immediately below its *text* on the menu bar (or above, if there is no room below). When selected with ENTER or RIGHT ARROW, any subsequent menu drop-box appears (cascades) immediately to the right of the MENU *text* in the preceding menu box (or left, if there is no room to the right). LEFT ARROW backs up to the preceding menu. The KEY attribute designates a separate accelerator key for the field. This may be any valid Clarion keycode to immediately pull down the MENU.

The *text* string may contain an ampersand (&) which designates the following character as the accelerator "hot" key which is automatically

underlined. If the MENU is on the menu bar, pressing the Alt key together with the accelerator key highlights and displays the MENU. If the MENU is within another MENU, pressing the accelerator key, alone, highlights and executes the MENU. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the accelerator key for the MENU, but it will not be underlined. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

Example:

```
 !An MDI application frame window with main menu for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
         MENUBAR
           MENU('File'),USE(?FileMenu),FIRST
             ITEM('Open...'),USE(?OpenFile)
             ITEM('Close'),USE(?CloseFile),DISABLE
             ITEM('E&xit'),USE(?MainExit)
           END
           MENU('Edit'),USE(?EditMenu),KEY(CtrlE),HLP('EditMenuHelp')
             ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo),DISABLE
             ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
             ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
             ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
           END
           MENU('Window'),STD(STD:WindowList),MSG('Arrange or Select Window'),LAST
             ITEM('Tile'),STD(STD:TileWindow)
             ITEM('Cascade'),STD(STD:CascadeWindow)
             ITEM('Arrange Icons'),STD(STD:ArrangeIcons)
           END
           MENU('Help'),USE(?HelpMenu),LAST,RIGHT
             ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
             ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
             ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
             ITEM('About MyApp...'),USE(?HelpAbout)
           END
         END
       END
```

ITEM (declare a menu item)

```
ITEM(text) [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,STD( )] [,CHECK] [,DISABLE]
                [, | FIRST | ] [,SEPARATOR]
                   | LAST  |
```

ITEM	Declares a menu choice within a MENUBAR or MENU structure.
text	A string constant containing the display text for the menu item.
USE	A field equate label to reference the menu item in executable code, or the variable used with CHECK.
KEY	Specifies an integer constant or keycode equate that immediately executes the menu item.
MSG	Specifies a string constant containing the text to display in the status bar when the menu item is highlighted.
HLP	Specifies a string constant containing the help system identifier for the menu item.
STD	Specifies an integer constant or equate that identifies a "Windows standard action" the menu item executes.
CHECK	Specifies an on/off ITEM.
DISABLE	Specifies the menu item appears dimmed when the WINDOW or APPLICATION is first opened.
FIRST	Specifies the ITEM appears at the top of the menu when menus are merged.
LAST	Specifies the ITEM appears at the bottom of the menu when menus are merged.
SEPARATOR	Specifies the ITEM displays a solid horizontal line across the menu box at run-time to delimit groups of menu selections. No other attributes may be specified with SEPARATOR.

ITEM declares a menu choice within a MENUBAR or MENU structure. The *text* string may contain an ampersand (&) which designates the following character as an accelerator "hot" key which is automatically underlined. If the ITEM is on the menu bar, pressing the Alt key together with the accelerator key highlights and executes the ITEM. If the ITEM is in a MENU, pressing the accelerator key, alone, when the menu is displayed, highlights and executes the ITEM. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the accelerator key for the ITEM, which will not be underlined. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

The KEY attribute designates a separate “hot” key for the field. This may be any valid Clarion keycode to immediately execute the ITEM’s action.

A cursor bar highlights individual ITEMs within the MENU structure. Each ITEM is usually associated with some code to be executed upon selection of that ITEM, unless the STD attribute is present. The STD atribute specifies a standard Windows action the menu item performs, such as Tile or Cascade the windows.

The SEPARATOR attribute creates an ITEM which serves only to delimit groups of menus selections so it should not have a *text* parameter, nor any other attributes. It creates a solid horizontal line across the menu box.

An ITEM that is not within a MENU structure is placed on the menu bar. This creates a menu bar selection which has no related drop-down menu. The normal convention to indicate this to the user is to terminate the *text* displayed for the item with an exclamation point (!). For example, the *text* for the ITEM might contain ‘Exit!’ to alert the user to the executable nature of the menu choice.

Events Generated:

EVENT:Accepted The control has been pressed by the user.

Example:

```
 !An MDI application frame window with main menu for the application:
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          ITEM('E&xit!'),USE(?MainExit),FIRST
          MENU('File'),USE(?FileMenu),FIRST
            ITEM('Open...'),USE(?OpenFile) ,HLP('OpenFileHelp') ,FIRST
            ITEM('Close'),USE(?CloseFile),HLP('CloseFileHelp'),DISABLE
            ITEM('Auto Increment'),USE(ToggleVar),CHECK
          END
          MENU('Edit'),USE(?EditMenu),KEY(CtrlE),HLP('EditMenuHelp')
            ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo),DISABLE
            ITEM,SEPARATOR
            ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
            ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
            ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
          END
          MENU('Window'),STD(STD:WindowList),MSG('Arrange or Select Window'),LAST
            ITEM('Tile'),STD(STD:TileWindow)
            ITEM('Cascade'),STD(STD:CascadeWindow)
            ITEM('Arrange Icons'),STD(STD:ArrangeIcons)
            ITEM,SEPARATOR
          END
          MENU('Help'),USE(?HelpMenu),LAST,RIGHT
            ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
            ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
            ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
            ITEM('About MyApp...'),USE(?HelpAbout),MSG('Copyright Info'),LAST
          END
        END
      END
```

TOOLBAR and WINDOW Control Fields

BOX (declare a window box control)

BOX ,**AT( )** [,**USE( )**] [,**DISABLE**] [,**COLOR( )**] [,**FILL( )**] [,**ROUND**] [,**FULL**] [,**SCROLL**] [,**HIDE**]

BOX	Places a rectangular box on the window.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW (or APPLICATION) is first opened.
COLOR	Specifies the color for the border of the control. If omitted, the border is black.
FILL	Specifies the fill color for the control. If omitted, the box is not filled with color.
ROUND	Specifies the box corners are rounded. If omitted, the corners are square.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

The **BOX** control places a rectangular box on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           BOX,FILL(COLOR:MENU),FULL          !Filled, full screen, black border
           BOX,AT(0,0,20,20)                  !Unfilled, black border
           BOX,AT(0,20,20,20),USE(?Box1),DISABLE
                                              !Unfilled, black border, dimmed
           BOX,AT(20,20,20,20),ROUND          !Unfilled, rounded, black border
           BOX,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                              !Filled, black border
           BOX,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
                                              !Unfilled, active border color border
           BOX,AT(480,180,20,20),SCROLL       !Scrolls with screen
         END
```

BUTTON (declare a pushbutton control)

```
BUTTON(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
    [,STD( )] [,FONT( )] [,ICON( )] [,DEFAULT] [,IMM][,REQ] [,FULL] [,SCROLL] [,ALRT( )]
    [,HIDE] [DROPID( )] [TIP( )] [, | LEFT  | ]
                                    | RIGHT |
```

BUTTON	Places a command button on the WINDOW or TOOLBAR.
text	A string constant containing the text to display on the button face (along with any ICON specified). This may contain an ampersand (&) to indicate the "hot" letter (accelerator key) for the button.
AT	Specifies the initial size and location of the control. If omitted, default values are set by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to and presses the button.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
STD	Specifies an integer constant or equate that identifies a "Windows standard action" the control executes.
FONT	Specifies the display font for the control.
ICON	Specifies an .ICO file or standard icon to display on the button face.
DEFAULT	Specifies the BUTTON is automatically pressed when the user presses the ENTER key.
IMM	Specifies the control generates an event when the left mouse button is pressed, continuing as long as it is depressed. If omitted, an event is generated only when the left mouse button is pressed and released on the control.

REQ	Specifies that when the BUTTON is pressed, the runtime library automatically checks all ENTRY controls in the same WINDOW with the REQ attribute to ensure they contain data other than blanks or zeroes.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
LEFT	Specifies that the *text* appears to the left of the icon.
RIGHT	Specifies that the *text* appears to the right of the icon.

The **BUTTON** control places a pushbutton on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

A BUTTON with the IMM attribute generates an event as soon as the left mouse button is pressed on the control and continues to do so until it is released. This allows the BUTTON control's executable code to execute continuously until the mouse button is released. A BUTTON without the IMM attribute generates an event only when the left mouse button is pressed and released on the control.

A BUTTON with the REQ attribute is a "required control fields check" button. REQ attributes of ENTRY or TEXT control fields are not checked until a BUTTON with the REQ attribute is pressed or the INCOMPLETE function is called. Focus is given to the first required control which is blank or zero.

A BUTTON with an ICON attribute displays the icon on the button face in addition to its *text* parameter (which appears below the icon, by default). The *text* parameter also serves for accelerator "hot" key definition.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The control has been pressed by the user.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            BUTTON('1'),AT(0,0,20,20),USE(?B1)
            BUTTON('2'),AT(20,0,20,20),USE(?B2),KEY(F10Key)
            BUTTON('3'),AT(40,0,20,20),USE(?B3),MSG('Button 3')
            BUTTON('4'),AT(60,0,20,20),USE(?B4),HLP('Button4Help')
            BUTTON('5'),AT(80,0,20,20),USE(?B5),STD(STD:Cut)
            BUTTON('6'),AT(100,0,20,20),USE(?B6),FONT('Arial',12)
            BUTTON('7'),AT(120,0,20,20),USE(?B7),ICON(ICON:Question)
            BUTTON('8'),AT(140,0,20,20),USE(?B8),DEFAULT
            BUTTON('9'),AT(160,0,20,20),USE(?B9),IMM
            BUTTON('10'),AT(180,0,20,20),USE(?B10),CURSOR(CURSOR:Wait)
            BUTTON('11'),AT(200,0,20,20),USE(?B11),REQ
            BUTTON('12'),AT(220,0,20,20),USE(?B12),ALRT(F10Key)
            BUTTON('13'),AT(240,0,20,20),USE(?B13),SCROLL
          END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?B1
      !Perform some action
    END
  END
```

CHECK (declare a window checkbox control)

```
CHECK(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
            [,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,DROPID( )]
            [,TIP( )] [,  | LEFT  | ]
                          | RIGHT |
```

CHECK	Places a check box on the WINDOW or TOOLBAR.
text	A string constant containing the text to display for the check box. This may contain an ampersand (&) to indicate the "hot" letter for the check box.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a numeric variable to receive the value of the check box, zero (0 = OFF) or one (1 = ON).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to and toggles the box.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FONT	Specifies the display font for the control.
ICON	Specifies an .ICO file or standard icon to display on the button face of a "latching" pushbutton.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-

and-drop actions.

TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
LEFT	Specifies that the text appears to the left of the check box.
RIGHT	Specifies that the text appears to the right of the check box (the default position).

The **CHECK** control places a check box on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. A CHECK with an ICON attribute appears as a "latched" button with the icon displayed on the button face. When the button appears "up" the CHECK is off and the USE variable receives a zero (0); when it appears "down" the CHECK is on and the USE variable receives a one (1). The PROP:TrueValue and PROP:FalseValue runtime properties can be used to automatically set the USE variable to values other than zero and one.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The control has been toggled by the user.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            CHECK('1'),AT(0,0,20,20),USE(C1)
            CHECK('2'),AT(0,20,20,20),USE(C2),KEY(F10Key)
            CHECK('3'),AT(0,40,20,20),USE(C3),MSG('Button 3')
            CHECK('4'),AT(0,60,20,20),USE(C4),HLP('Check4Help')
            CHECK('5'),AT(20,80,20,20),USE(C5),LEFT
            CHECK('6'),AT(0,100,20,20),USE(C6),FONT('Arial',12)
            CHECK('7'),AT(0,120,20,20),USE(C7),ICON(ICON:Question)
            CHECK('8'),AT(0,140,20,20),USE(C8),DEFAULT
            CHECK('9'),AT(0,160,20,20),USE(C9),IMM
            CHECK('10'),AT(0,180,20,20),USE(C10),CURSOR(CURSOR:Wait)
            CHECK('11'),AT(0,200,20,20),USE(C11),ALRT(F10Key),DISABLE
          END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?C1
      IF C1 = 1
        ENABLE(?C11)
      ELSE
        DISABLE(?C11)
      END
    END
  END
```

COMBO (declare an entry/list control)

```
COMBO(picture) ,FROM( ) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
       [,HLP( )] [,SKIP] [,FONT( )] [,FORMAT( )] [,DROP] [,COLUMN] [,VCR] [,FULL]
       [,SCROLL] [,ALRT( )] [,HIDE] [,READONLY] [,REQ] [,NOBAR] [DROPID( )] [,TIP( )]
       [, | MARK() |][, | HSCROLL   |][, | LEFT    |][,  | INS  |][,  | UPR |]
          | IMM     |   | VSCROLL   |   | RIGHT   |      | OVR |      | CAP |
                        | HVSCROLL  |   | CENTER  |
                                        | DECIMAL |
```

COMBO	Places a data entry field with an associated list of data items on the WINDOW or TOOLBAR.
picture	A display picture token that specifies the input format for the data entered into the control.
FROM	Specifies the origin of the data displayed in the list.
AT	Specifies the initial size and location of the control. If omitted, the runtime library chooses a value.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code or the label of the variable that receives the value selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
FORMAT	Specifies the display format of the data.
DROP	Specifies a drop-down list box and the number of elements the drop-down portion contains.
COLUMN	Specifies a field-by-field highlight bar on multi-column list boxes.
VCR	Specifies a VCR-type control to the left of the horizontal scroll bar (if present).

FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry.
NOBAR	Specifies the highlight bar is displayed only when the LIST has focus.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
REQ	Specifies the control may not be left blank or zero.
MARK	Specifies multiple item selection mode.
IMM	Specifies generation of an event whenever the user presses any key.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the list box when any portion of the data item lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the list box when any data items lie vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the list box when any portion of the data items lies outside the visible area.
LEFT	Specifies that the data is left justified within the list.
RIGHT	Specifies that the data is right justified within the list.
CENTER	Specifies that the data is centered within the list.
DECIMAL	Specifies that the data is aligned on the decimal point within the list.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) entry.

The **COMBO** control places a data entry field with an associated list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute (a combination of an ENTRY and LIST control). The user

may type in data or select an item from the list. The entered data is not automatically validated against the entries in the list. The data entry portion of the COMBO acts as an "incremental locator" to the list—as the user types each character, the highlight bar is positioned to the closest matching entry.

A COMBO with the DROP attribute displays only the currently selected data item on screen until the control has focus and the user presses the down arrow key, or CLICKS ON the the icon to the right of the displayed data item. When either of these occurs, the selection list appears ("drops down") to allow the user to select an item.

A COMBO with the IMM attribute generates an event every time the user moves the highlight bar to another selection, or pressed any key that causes the displayed entries to scroll. This allows an opportunity for the source code to re-fill the display QUEUE, or get the currently highlighted record to display other fields from the record. A COMBO with the VCR attribute has scroll control buttons like a **V**ideo **C**assette **R**ecorder to the left of the horizontal scroll bar (if there is one). These buttons allow the user to use the mouse to scroll through the list.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has selected an entry.
EVENT:Rejected	The user has entered an invalid value for the entry picture.
EVENT:NewSelection	The current selection in the list has changed (highlight has moved up or down).
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

A COMBO with the IMM attribute also generates the following events:

EVENT:ScrollUp	The highlight bar has attempted to move off the top of the LIST.
EVENT:ScrollDown	The highlight bar has attempted to move off the bottom of the LIST.
EVENT:PageUp	The user pressed PgUp.
EVENT:PageDown	The user pressed PgDn.

EVENT:ScrollTop The user pressed Ctrl-PgUp.

EVENT:ScrollBottom
The user pressed Ctrl-PgDn.

EVENT:PreAlertKey
The user pressed a printable character or an ALRT attribute hot key.

EVENT:AlertKey The user pressed a printable character or an ALRT attribute hot key.

EVENT:Locate The user pressed the locator VCR button.

A COMBO with the DROP attribute also generates the following events:

EVENT:DroppingDown
The user pressed the down arrow button.

EVENT:DroppedDown
The list has dropped.

Example:

```
MDIChild  WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            COMBO(@S8),AT(0,0,20,20),USE(C1),FROM(Que)
            COMBO(@S8),AT(20,0,20,20),USE(C2),FROM(Que),KEY(F10Key)
            COMBO(@S8),AT(40,0,20,20),USE(C3),FROM(Que),MSG('Button  3')
            COMBO(@S8),AT(60,0,20,20),USE(C4),FROM(Que),HLP('Check4Help')
            COMBO(@S8),AT(80,0,20,20),USE(C5),FROM(Q)  |
               ,FORMAT('5C~List~15L~Box~'),COLUMN
            COMBO(@S8),AT(100,0,20,20),USE(C6),FROM(Que),FONT('Arial',12)
            COMBO(@S8),AT(120,0,20,20),USE(C7),FROM(Que),DROP(8)
            COMBO(@S8),AT(140,0,20,20),USE(C8),FROM(Que),HVSCROLL,VCR
            COMBO(@S8),AT(160,0,20,20),USE(C9),FROM(Que),IMM
            COMBO(@S8),AT(180,0,20,20),USE(C10),FROM(Que),CURSOR(CURSOR:Wait)
            COMBO(@S8),AT(200,0,20,20),USE(C11),FROM(Que),ALRT(F10Key)
            COMBO(@S8),AT(220,0,20,20),USE(C12),FROM(Que),LEFT
            COMBO(@S8),AT(240,0,20,20),USE(C13),FROM(Que),RIGHT
            COMBO(@S8),AT(260,0,20,20),USE(C14),FROM(Que),CENTER
            COMBO(@N8.2),AT(280,0,20,20),USE(C15),FROM(Que),DECIMAL
            END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?C1
      LOOP X# = 1 to RECORDS(Que)     !Check for user's entry in Que
        GET(Que,X#)
        IF C1 = Que THEN BREAK.       !Break loop if present
      END
      IF X# > RECORDS(Que)            !Check for BREAK
        Que = C1                      ! and add the entry
        ADD(Que)
  . . .
```

See Also: LIST, ENTRY

CUSTOM (declare a window .VBX custom control)

CUSTOM(*text***) ,AT( ) [,CLASS( )] [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]**
[,HLP( )] [,SKIP] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,FONT( )] [DROPID( )]
[,TIP( )] [,*property(value)***]**

CUSTOM	Places a Visual Basic .VBX control on the WINDOW or TOOLBAR.
text	A string constant containing the title for the control.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the control.
CLASS	Specifies the .VBX filename and type of control.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a variable to receive the value of the control.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
FONT	Specifies the display font for the control.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.

property	A string constant containing the name of a custom property setting for the control.
value	A string constant containing the property value number or EQUATE for the *property*.

The **CUSTOM** control places a Visual Basic .VBX control on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

The *property* attribute allows you to specify any additional property settings the .VBX control may require. These are properties that need to be set for the .VBX control to properly function, and are not standard Clarion properties (such as AT, CURSOR, or USE). The custom control should only receive values for these properties that are defined for that control. Valid properties and values for those properties would be defined in the custom control's documentation. You may have multiple *property* attributes on a single CUSTOM control.

Events Generated:

EVENT:VBXevent	A VBX-specific event occurred. Interrogate the PROP:VBXEvent and PROP:VBXEventArg properties for the event.
EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has completed using the control.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           CUSTOM,AT(0,0,120,320),USE(C1), |
                 CLASS('graph.vbx','graph'),'graphstyle'('2')
         END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?C1
      ?C1{'graphstyle'} = '3'   !Change graphstyle property "on the fly"
                                ! using runtime property access syntax
    END
  END
```

ELLIPSE (declare a window ellipse control)

ELLIPSE ,AT( ) [,USE( )] [,DISABLE] [,COLOR( )] [,FILL( )] [,FULL] [,SCROLL] [,HIDE]

ELLIPSE	Places a “circular” figure on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
COLOR	Specifies the color for the border of the ellipse. If omitted, the ellipse has a black border.
FILL	Specifies the fill color for the control. If omitted, the ellipse is not filled with color.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

The **ELLIPSE** control places a “circular” figure on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The ellipse is drawn inside a “bounding box” defined by the *x*, *y*, *width*, and *height* parameters of it sAT attribute. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the “bounding box.” This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           ELLIPSE,FILL(COLOR:MENU),FULL        !Filled, full screen, black border
           ELLIPSE,AT(0,0,20,20)                        !Unfilled, black border
           ELLIPSE,AT(0,20,20,20),USE(?Box1),DISABLE    !Dimmed
           ELLIPSE,AT(20,20,20,20),ROUND        !Unfilled, rounded, black border
           ELLIPSE,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                                !Filled, black border
           ELLIPSE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDeR)
                                          !Unfilled, active border color border
           ELLIPSE,AT(480,180,20,20),SCROLL !Scrolls with screen
         END
```

ENTRY (declare a data entry control)

```
ENTRY(picture) ,AT() [,CURSOR()] [,USE()] [,DISABLE] [,KEY()] [,MSG()] [,HLP()] [,SKIP]
      [,FONT()] [,IMM] [,PASSWORD] [,REQ] [,FULL] [,SCROLL][,ALRT()] [,HIDE] [,TIP( )]
      [,READONLY] [DROPID( )] [,  | INS     | ] [, | CAP | ] [, | LEFT    | ]
                                  | OVR     |    | UPR |    | RIGHT   |
                                                            | CENTER  |
                                                            | DECIMAL |
```

ENTRY	Places a data entry field on the WINDOW or TOOLBAR.
picture	A display picture token that specifies the input format for the data entered into the control.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable that receives the value entered into the control by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
IMM	Specifies immediate event generation whenever the user presses any key.
PASSWORD	Specifies non-display of the data entered (password mode).
REQ	Specifies the control may not be left blank or zero.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.

ALRT	Specifies “hot” keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
TIP	Specifies the text that displays as “balloon help” when the mouse cursor pauses over the control.
READONLY	Specifies the control does not allow data entry.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) entry.
LEFT	Specifies that the data entered is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the data entered is right justified within the area specified by the AT attribute.
CENTER	Specifies that the data entered is centered within the area specified by the AT attribute.
DECIMAL	Specifies that the data entered is aligned on the decimal point within the area specified by the AT attribute.

The **ENTRY** control places a data entry field on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. Data entered is formatted according to the *picture*, and the variable specified in the USE attribute receives the data entered when the user has completed data entry and moves on to another control. Data entry scrolls horizontally to allow the user to enter data to the full length of the variable. Therefore, the right and left arrow keys move within the data in the ENTRY control.

An ENTRY control with the PASSWORD attribute displays asterisks when the user enters data. This is useful for password-type variables. An ENTRY control with the SKIP attribute is used for seldom-used data entry. Display-only data should be declared with the READONLY attribute.

The MASK attribute on a WINDOW specifies pattern input editing mode of all controls in the window. This means that, as the user types in data, each character is automatically validated against the control’s picture for proper input (numbers only in numeric pictures, etc.). This forces the user to enter data in the format specified by the control’s display picture. If omitted, Windows free-input is allowed in the controls. This is Windows’ default data entry mode. Free-input means the user’s data is formatted to the control’s picture only after entry. This allows users to enter data as they choose and it is automatically formatted to the control’s picture after entry. If the user types in data in a format different from the control’s picture, the libraries

attempt to determine the format the user used, and convert the data to the control's display picture. For example, if the user types "January 1, 1995" into a control with a display picture of @D1, the runtime library formats the user's input to "1/1/95." This action occurs only after the user completes data entry and moves to another control. If the runtime library cannot determine what format the user used, it will not update the USE variable. It then beeps and leaves the user on the same control with the data they entered, to allow them to try again.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has completed data entry in the control.
EVENT:Rejected	The user has entered an invalid value for the entry picture.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

An ENTRY with the IMM attribute also generates the following events:

EVENT:NewSelection	The user has pressed a key.

Example:

```
MDIChild  WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            ENTRY(@S8),AT(0,0,20,20),USE(E1)
            ENTRY(@S8),AT(20,0,20,20),USE(E2),KEY(F10Key)
            ENTRY(@S8),AT(40,0,20,20),USE(E3),MSG('Button  3')
            ENTRY(@S8),AT(60,0,20,20),USE(E4),HLP('Entry4Help')
            ENTRY(@S8),AT(80,0,20,20),USE(E5),DISABLE
            ENTRY(@S8),AT(100,0,20,20),USE(E6),FONT('Arial',12)
            ENTRY(@S8),AT(120,0,20,20),USE(E7),REQ,INS,CAP
            ENTRY(@S8),AT(140,0,20,20),USE(E8),SCROLL,OVR,UPR
            ENTRY(@S8),AT(160,0,20,20),USE(E9),IMM
            ENTRY(@S8),AT(180,0,20,20),USE(E10),CURSOR(CURSOR:Wait)
            ENTRY(@S8),AT(200,0,20,20),USE(E11),ALRT(F10Key)
            ENTRY(@S8),AT(220,0,20,20),USE(E12),LEFT
            ENTRY(@S8),AT(240,0,20,20),USE(E13),RIGHT
            ENTRY(@S8),AT(260,0,20,20),USE(E14),CENTER
            ENTRY(@N8.2),AT(280,0,20,20),USE(E15),DECIMAL
          END
```

GROUP (declare a group of window controls)

```
GROUP(text) ,AT() [,CURSOR()] [,USE()] [,DISABLE] [,KEY()] [,MSG()]  [,HLP()] [,FONT()]
            [,BOXED] [,FULL] [,SCROLL] [,HIDE] [,ALRT( )] [,SKIP] [,TIP( )] [,DROPID( )]
   controls
END
```

GROUP	Declares a group of controls that may be referenced as one entity.
text	A string constant containing the prompt for the group of controls. This may contain an ampersand (&) to indicate the "hot" letter for the prompt. The *text* is displayed on screen only if the BOXED attribute is also present.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (or any control within the GROUP). If omitted, the window's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the GROUP control and the controls in the GROUP appear dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the first control in the GROUP.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the GROUP has focus.
HLP	Specifies a string constant containing the default help system identifier for any control in the GROUP.
FONT	Specifies the display font for the control and the default for all the controls in the GROUP.
BOXED	Specifies a single-track border around the group of controls with the text at the top of the border.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the GROUP control and the controls in the GROUP scroll with the window.
HIDE	Specifies the GROUP control and the controls in the GROUP do not appear when the WINDOW or APPLI-

CATION is first opened. UNHIDE must be used to display them.

ALRT	Specifies "hot" keys active for the controls in the GROUP.
SKIP	Specifies the controls in the GROUP do not receive input focus and may only be accessed with the mouse or accelerator key.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
controls	Control declarations that may be referenced as the GROUP.

The **GROUP** control declares a group of controls that may be referenced as one entity. GROUP allows the user to use the cursor keys instead of the TAB key to move between the *controls* in the GROUP, and provides default MSG and HLP attributes for all controls in the GROUP. This control cannot receive input focus.

Events Generated:

EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           GROUP('Group 1'),USE(?G1),KEY(F10Key)
             ENTRY(@S8),AT(0,0,20,20),USE(?E1)
             ENTRY(@S8),AT(20,0,20,20),USE(?E2)
           END
           GROUP('Group 2'),USE(?G2),MSG('Group 2')
             ENTRY(@S8),AT(40,0,20,20),USE(?E3)
             ENTRY(@S8),AT(60,0,20,20),USE(?E4)
           END
           GROUP('Group 3'),USE(?G3),AT(80,0,20,20),BOXED
             ENTRY(@S8),AT(80,0,20,20),USE(?E5)
             ENTRY(@S8),AT(100,0,20,20),USE(?E6)
           END
           GROUP('Group 4'),USE(?G4),FONT('Arial',12)
             ENTRY(@S8),AT(120,0,20,20),USE(?E7)
             ENTRY(@S8),AT(140,0,20,20),USE(?E8)
           END
           GROUP('Group 5'),USE(?G5),CURSOR(CURSOR:Wait)
             ENTRY(@S8),AT(160,0,20,20),USE(?E9)
             ENTRY(@S8),AT(180,0,20,20),USE(?E10)
           END
           GROUP('Group 6'),USE(?G6),SCROLL,HLP('Group6Help')
             ENTRY(@S8),AT(200,0,20,20),USE(?E11)
             ENTRY(@S8),AT(220,0,20,20),USE(?E12)
           END
         END
```

IMAGE (declare a window graphic image control)

IMAGE(*file***) ,AT( ) [,USE( )] [,DISABLE] [,FULL] [,SCROLL] [,HIDE] [, | HSCROLL |]**
| VSCROLL |
| HVSCROLL |

IMAGE	Places a graphic image on the WINDOW or TOOLBAR.
file	A string constant containing the name of the file to display. The file is linked into the .EXE as a resource.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the IMAGE control when the graphic image is wider than the area specified for display.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the IMAGE control when the graphic image is taller than the area specified for display.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the IMAGE control when the graphic image is larger than the display area.

The **IMAGE** control places a graphic image on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. This may be a bitmap (.BMP), icon (.ICO), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or Windows metafile (.WMF). This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           IMAGE('PIC.BMP'),AT(0,0,20,20),USE(?I1)
           IMAGE('PIC.WMF'),AT(40,0,20,20),USE(?I3),SCROLL
         END
```

LINE (declare a window line control)

LINE ,**AT( )** [,**USE( )**] [,**DISABLE**] [,**COLOR( )**] [,**FULL**] [,**SCROLL**] [,**HIDE**]

LINE	Places a straight line on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
COLOR	Specifies the color for the line. If omitted, the color is black.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

The **LINE** control places a straight line on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The *x* and *y* parameters of the AT attribute specify the starting point of the line. The *width* and *height* parameters of the AT attribute specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* is negative, the line slopes left; if the *height* is negative, the line slopes left. If either the *width* or *height* is zero, the line is horizontal or vertical. This control cannot receive input focus and does not generate events.

Width	Height	Result
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           LINE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)    !Border color
           LINE,AT(480,180,20,20),SCROLL                     !Scrolls with screen
         END
```

LIST (declare a window list control)

```
LIST ,FROM( ) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
     [,FONT( )] [,FORMAT( )] [,DROP] [,COLUMN] [,VCR] [,FULL] [,SCROLL] [,NOBAR]
     [,ALRT( )] [,HIDE] [,DRAGID( )] [,DROPID( )] [,TIP( )]
     [, | MARK( ) | ] [, | HSCROLL   | ] [, | LEFT    | ]
        | IMM      |    | VSCROLL   |    | RIGHT   |
                        | HVSCROLL  |    | CENTER  |
                                         | DECIMAL |
```

LIST	Places a scrolling list of data items on the WINDOW or TOOLBAR.
FROM	Specifies the origin of the data displayed in the list.
AT	Specifies the initial size and location of the control. If omitted, the runtime library chooses a value.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code, or the label of the variable that receives the value selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FONT	Specifies the display font for the control.
FORMAT	Specifies the display format of the data. This can include icons, colors, and tree controls.
DROP	Specifies a drop-down list box and the number of elements the drop-down portion contains.
COLUMN	Specifies cell-by-cell highlighting on multi-column lists.
VCR	Specifies a VCR-type control to the left of the horizontal scroll bar (if present).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.

SCROLL	Specifies the control scrolls with the window.
NOBAR	Specifies the highlight bar is displayed only when the LIST has focus.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DRAGID	Specifies the control may serve as a drag host for drag-and-drop actions.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
MARK	Specifies multiple items selection mode.
IMM	Specifies generation of an event whenever the user presses any key.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the list box when any portion of the data item lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the list box when any data items lie vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the list box when any portion of the data items lies outside the visible area.
LEFT	Specifies that the data is left justified within the LIST.
RIGHT	Specifies that the data is right justified within the LIST.
CENTER	Specifies that the data is centered within the LIST.
DECIMAL	Specifies that the data is aligned on the decimal point within the LIST.

The **LIST** control places a scrolling list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The data items displayed in the LIST come from a QUEUE or STRING specified by the FROM attribute and are formatted by the parameters specified in the FORMAT attribute (which can include colors, icons, and tree control parameters).

The CHOICE function returns the QUEUE entry number (the value returned by POINTER(queue)) of the selected item when the EVENT:Accepted event has been generated by the LIST. The data displayed in the LIST is automatically refreshed every time through the ACCEPT loop, whether the AUTO attribute is present or not.

A LIST with the DROP attribute displays only the currently selected data item on screen until the control has focus and the user presses the down arrow key, or CLICKS ON the the icon to the right of the displayed data item. When either of these occurs, the selection list appears ("drops down") to allow the user to select an item.

A LIST with the IMM attribute generates an event every time the user moves the highlight bar to another selection, or pressed any key that causes the displayed entries to scroll. This allows an opportunity for the source code to re-fill the display QUEUE, or get the currently highlighted record to display other fields from the record. If VSCROLL is also present, the vertical scroll bar is always displayed and when the end-user CLICKS on the scroll bar, events are generated but the list does not move (executable code should perfrom this action). You can interrogate the PROP:VscrollPos property to determine the scroll thumb's position from 0 (top) to 255 (bottom).

A LIST with the VCR attribute has scroll control buttons like a **V**ideo **C**assette **R**ecorder to the left of the horizontal scroll bar (if there is one). These buttons allow the user to use the mouse to scroll through the list.

A LIST with the DRAGID attribute can serve as a drag-and-drop host, providing information to be moved or copied to another control. A LIST with the DROPID attribute can serve as a drag-and-drop target, receiving information from another control. These attributes work together to specify drag-and-drop "signatures" that define a valid target for the operation. The DRAGID() and DROPID() functions, along with the SETDROPID procedure, are used to perform the data exchange.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has selected an entry from the control.
EVENT:NewSelection	The current selection in the list has changed (the highlight bar has moved up or down).
EVENT:ScrollUp	The highlight bar has attempted to move off the top of the LIST (only with the IMM attribute).
EVENT:ScrollDown	The highlight bar has attempted to move off the bottom of the LIST (only with the IMM attribute).
EVENT:PageUp	The user pressed PGUP (only with the IMM attribute).
EVENT:PageDown	The user pressed PGDN (only with the IMM attribute).
EVENT:ScrollTop	The user pressed CTRL+PGUP (only with IMM attribute).
EVENT:ScrollBottom	The user pressed CTRL+PGDN (only with IMM attribute).

EVENT:Locate The user pressed the locator VCR button (only with the IMM attribute).

EVENT:ScrollDrag The scroll bar "thumb" is being moved (only with the IMM attribute).

EVENT:PreAlertKey
The user pressed a printable character (only with the IMM attribute) or an ALRT attribute hot key.

EVENT:AlertKey The user pressed a printable character (only with the IMM attribute) or an ALRT attribute hot key.

EVENT:Dragging The mouse cursor is over a potential drag target (only with the DRAGID attribute).

EVENT:Drag The mouse cursor has been released over a drag target (only with the DRAGID attribute).

EVENT:Drop The mouse cursor has been released over a drag target (only with the DROPID attribute).

EVENT:DroppingDown
The user has requested the droplist drop down (only with the DROP attribute). CYCLE aborts the dropdown.

EVENT:DroppedDown
The user has dropped the droplist (only with the DROP attribute).

EVENT:Expanding The user has clicked on a tree expansion box (only with the T in the FORMAT attribute string). CYCLE aborts the expansion.

EVENT:Expanded The user has clicked on a tree expansion box (only with the T in the FORMAT attribute string).

EVENT:Contracting
The user has clicked on a tree contraction box (only with the T in the FORMAT attribute string). CYCLE aborts the contraction.

EVENT:Contracted The user has clicked on a tree contraction box (only with the T in the FORMAT attribute string).

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           LIST,AT(0,0,20,20),USE(?L1),FROM(Que),IMM
           LIST,AT(20,0,20,20),USE(?L2),FROM(Que),KEY(F10Key)
           LIST,AT(40,0,20,20),USE(?L3),FROM(Que),MSG('Button 3')
           LIST,AT(60,0,20,20),USE(?L4),FROM(Que),HLP('Check4Help')
           LIST,AT(80,0,20,20),USE(?L5),FROM(Q),FORMAT('5C~List~15L~Box~'),COLUMN
           LIST,AT(100,0,20,20),USE(?L6),FROM(Que),FONT('Arial',12)
           LIST,AT(120,0,20,20),USE(?L7),FROM(Que),DROP(6)
           LIST,AT(140,0,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR
           LIST,AT(180,0,20,20),USE(?L10),FROM(Que),CURSOR(CURSOR:Wait)
           LIST,AT(200,0,20,20),USE(?L11),FROM(Que),ALRT(F10Key)
           LIST,AT(220,0,20,20),USE(?L12),FROM(Que),LEFT
           LIST,AT(240,0,20,20),USE(?L13),FROM(Que),RIGHT
           LIST,AT(260,0,20,20),USE(?L14),FROM(Que),CENTER
           LIST,AT(280,0,20,20),USE(?L15),FROM(Que),DECIMAL
         END
```

See Also: COMBO, DRAGID, DROPID, SETDROPID

OPTION (declare a group of window RADIO controls)

```
OPTION(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,BOXED]
             [,FULL] [,SCROLL] [,HIDE] [,FONT( )] [,ALRT( )] [,SKIP] [DROPID( )] [,TIP( )]
  radios
END
```

OPTION	Declares a group of RADIO controls.
text	A string constant containing the prompt for the group of controls. This may contain an ampersand (&) to indicate the "hot" letter for the prompt. The *text* is displayed on screen only if the BOXED attribute is also present.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a variable to receive the choice. If this is a string variable, it receives the value of the RADIO string (with any accelerator key ampersand stripped out) selected by the user. If a numeric variable, it receives the number of the RADIO button selected by the user (the value returned by the CHOICE() function).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the currently selected RADIO in the OPTION control.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the OPTION has focus.
HLP	Specifies a string constant containing the default help system identifier for any control in the OPTION.
BOXED	Specifies a single-track border around the RADIO controls with the text at the top of the border.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

FONT	Specifies the display font for the control and the default for all the controls in the OPTION.
ALRT	Specifies "hot" keys active for the controls in the OPTION.
SKIP	Specifies the controls in the OPTION do not receive input focus and may only be accessed with the mouse or accelerator key.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
radios	Multiple RADIO control declarations.

The **OPTION** control declares a group of RADIO controls that offer the user a list of choices. The multiple RADIO controls in the OPTION structure define the choices offered to the user.

Input focus changes between the OPTION's RADIO controls are signalled only to the individual RADIO controls affected. This means the events generated when the user changes input focus within an OPTION structure are field-specific events for the affected RADIO controls, not the OPTION structure which contains them.

A string variable as the OPTION structure's USE attribute receives the text of the RADIO control selected by the user, and the CHOICE(*?Option*) function returns the number of the selected RADIO button. If the OPTION structure's USE attribute is a numeric variable, it receives the number of the RADIO button selected by the user (the value returned by the CHOICE function).

No RADIO button selected is a valid option, which occurs only when the OPTION structure's USE variable does not contain a value related to one of its component RADIO controls. This condition only lasts until the user has selected one of the RADIOs.

Events Generated:

EVENT:Selected	One of the OPTION's RADIO controls has received input focus.
EVENT:Accepted	One of the OPTION's RADIO controls has been selected by the user.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
             RADIO('Radio 1'),AT(0,0,20,20),USE(?R1)
             RADIO('Radio 2'),AT(20,0,20,20),USE(?R2)
            END
            OPTION('Option 2'),USE(OptVar2),MSG('Option 2'),SCROLL
              RADIO('Radio 3'),AT(40,0,20,20),USE(?R3)
              RADIO('Radio 4'),AT(60,0,20,20),USE(?R4)
            END
            OPTION('Option 3'),USE(OptVar3),AT(80,0,20,20),BOXED
              RADIO('Radio 5'),AT(80,0,20,20),USE(?R5)
              RADIO('Radio 6'),AT(100,0,20,20),USE(?R6)
            END
           OPTION('Option 4'),USE(OptVar4),FONT('Arial',12),CURSOR(CURSOR:Wait)
             RADIO('Radio 7'),AT(120,0,20,20),USE(?R7)
             RADIO('Radio 8'),AT(140,0,20,20),USE(?R8)
          END
        END
```

See Also: RADIO

PROMPT (declare a prompt control)

```
PROMPT(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,FONT( )] [,FULL] [,SCROLL]
      [,HIDE] [,DROPID( )] [,  | LEFT   | ]
                               | RIGHT  |
                               | CENTER |
```

PROMPT	Places a prompt for the next active control following it, in the WINDOW or TOOLBAR.
text	A string constant containing the text to display. This may contain an ampersand (&) to indicate the "hot" letter for the prompt.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FONT	Specifies the font used to display the text.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
LEFT	Specifies that the prompt is left justified.
RIGHT	Specifies that the prompt is right justified.
CENTER	Specifies that the prompt is centered.

The **PROMPT** control places a prompt for the next active control following the PROMPT in the WINDOW or TOOLBAR structure. The prompt *text* is placed on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

The *text* may contain an ampersand (&) to indicate the letter immediately following the ampersand is the "hot" letter for the prompt. By default, the

"hot" letter displays with an underscore below it to indicate its special purpose. This "hot" letter, when pressed in conjunction with the ALT key, changes input focus to the next control following the PROMPT in the WINDOW or TOOLBAR structure, which is capable of receiving focus.

Disabling or hiding the control directly following the PROMPT in the window structure does not autmatically disable or hide the PROMPT; it must also be explicitly disabled or hidden, otherwise the PROMPT will then refer to the next currently active control following the disabled control. This allows you to place one PROMPT control on the window that will apply to any of multiple controls (if only one will be active at a time). If the next active control is a BUTTON, it is pressed when the user presses the PROMPT's "hot key."

To include an ampersand as part of the prompt *text*, place two ampersands together (&&) in the *text* string and only one will display.

This control cannot receive input focus.

Events Generated:

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           PROMPT('Enter Data:'),AT(10,100,20,20),USE(?P1),CURSOR(CURSOR:Wait)
           ENTRY(@S8),AT(100,100,20,20),USE(E1)
           PROMPT('Enter More Data:'),AT(10,200,20,20),USE(?P2),CURSOR(CURSOR:Wait)
           ENTRY(@S8),AT(100,200,20,20),USE(E2)
           ENTRY(@D1),AT(100,200,20,20),USE(E3)
         END
  CODE
  OPEN(MDIChild)
  IF SomeCondition
    HIDE(?E2)         !Prompt will refer to E3
  ELSE
    HIDE(?E3)         !Prompt will refer to E2
  END
```

PROGRESS (declare a progress control)

PROGRESS, **AT( )** [,**CURSOR( )**] [,**USE( )**] [,**DISABLE**] [,**FULL**] [,**SCROLL**] [,**HIDE**] [,**DROPID( )**] [,**RANGE( )**]

PROGRESS	Places a control that displays the current progress of a batch process in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable containing the value of the current progress, or a field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
RANGE	Specifies the range of values the progress bar displays. If omitted, the default range is zero (0) to one hundred (100).

The **PROGRESS** control declares a control that displays a progress bar. This usually displays the current percentage of completion of a batch process.

If a variable is named as the USE attribute, the progress bar is automatically updated whenever the value in that variable changes. If the USE attribute is a field equate label, you must directly update the display by assigning a value (within the range defined by the RANGE attribute) to the control's PROP:progress property (an undeclared property equate -- see *Undeclared Properties*).

This control cannot receive input focus.

Events Generated:

EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
BackgroundProcess   PROCEDURE         !Background processing batch process

ProgressVariable    LONG

Win   WINDOW('Batch  Processing...'),AT(,,400,400),TIMER(1),MDI,CENTER
        PROGRESS,AT(100,100,200,20),USE(ProgressVariable),RANGE(0,200)
        PROGRESS,AT(100,140,200,20),USE(?ProgressBar),RANGE(0,200)
        BUTTON('Cancel'),AT(190,300,20,20),STD(STD:Close)
      END

  CODE
  OPEN(Win)
  OPEN(File)
  ?ProgressVariable{PROP:rangehigh} = RECORDS(File)
  ?ProgressBar{PROP:rangehigh} = RECORDS(File)
  SET(File)                           !Set up a batch process
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseWindow
      BREAK
    OF EVENT:Timer                          !Process records when timer allows it
      ProgressVariable += 3                 !Auto-updates 1st progress bar
      LOOP 3 TIMES
        NEXT(File)
        IF ERRORCODE() THEN BREAK.
        ?ProgressBar{PROP:progress} += 1    !Manually update 2nd progress bar
        !Perform some batch processing code
  . . .
  CLOSE(File)
```

RADIO (declare a window radio button control)

```
RADIO(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
        [,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,HIDE] [,ALRT( )] [DROPID( )] [VALUE( )]
        [,TIP( )] [,  | LEFT   | ]
                      | RIGHT |
```

RADIO	Places a radio button on the WINDOW or TOOLBAR.
text	A string constant containing the text to display for the radio button. This may contain an ampersand (&) to indicate the "hot" letter for the radio button.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately selects the radio button.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FONT	Specifies the display font for the control.
ICON	Specifies an .ICO file or standard icon to display on the face of a "latching" button.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
ALRT	Specifies "hot" keys active for the control.

DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
VALUE	Specifies the value the OPTION structure's USE variable receives when the radio button is selected by the user.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
LEFT	Specifies the text appears to the left of the radio button.
RIGHT	Specifies the text appears to the right of the radio button (this is the default position).

The **RADIO** control places a radio button on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. A RADIO control may only be placed within an OPTION control. When selected by the user, the RADIO *text* (with any accelerator key ampersand stripped out) is placed in the OPTION's USE variable, unless the VALUE attribute is used.

A RADIO with an ICON attribute appears as a "latched" pushbutton with the icon on the button face. When the icon appears "up" the RADIO is off; when it appears "down" the RADIO is on and the OPTION's USE variable receives the value in the selected RADIO's *text* parameter (unless the VALUE attribute is used).

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The control has been selected by the user.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           OPTION('Option 1'),USE(OptVar1)
             RADIO('Radio 1'),AT(0,0,20,20),USE(?R1),KEY(F10Key)
             RADIO('Radio 2'),AT(20,0,20,20),USE(?R2),MSG('Radio 2')
           END
           OPTION('Option 2'),USE(OptVar2)
             RADIO('Radio 3'),AT(40,0,20,20),USE(?R3),FONT('Arial',12)
             RADIO('Radio 4'),AT(60,0,20,20),USE(?R4),CURSOR(CURSOR:Wait)
           END
           OPTION('Option 3'),USE(OptVar3)
             RADIO('Radio 5'),AT(80,0,20,20),USE(?R5),HLP('Radio5Help')
             RADIO('Radio 6'),AT(100,0,20,20),USE(?R6)
           END
           OPTION('Option 4'),USE(OptVar4)
             RADIO('Radio 7'),AT(120,0,20,20),USE(?R7),ICON('Radio1.ICO')
             RADIO('Radio 8'),AT(140,0,20,20),USE(?R8),ICON('Radio2.ICO')
           END
           OPTION('Option 5'),USE(OptVar5)
             RADIO('Radio 9'),AT(100,20,20,20),USE(?R9),LEFT
             RADIO('Radio 10'),AT(120,20,20,20),USE(?R10),LEFT
           END
           OPTION('Option 6'),USE(OptVar6),SCROLL
             RADIO('Radio 11'),AT(200,0,20,20),USE(?R11),SCROLL
             RADIO('Radio 12'),AT(220,0,20,20),USE(?R12),SCROLL
           END
         END
```

See Also: OPTION

REGION (declare a window region control)

REGION ,**AT( )** [,**CURSOR( )**] [,**USE( )**] [,**DISABLE**] [,**FILL**] [,**COLOR( )**] [,**IMM**] [,**FULL**] [,**SCROLL**] [,**HIDE**] [,**DRAGID( )**] [,**DROPID( )**]

REGION	Defines an area in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control is disabled when the WINDOW or APPLICATION is first opened.
FILL	Specifies the red, green, and blue component values that create the fill color for the control. If omitted, the region is not filled with color.
COLOR	Specifies the border color of the control. If omitted, there is no border.
IMM	Specifies control generates an event whenever the mouse is moved in the region.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DRAGID	Specifies the control may serve as a drag host for drag-and-drop actions.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.

The **REGION** control defines an area on screen at the position and size specified by its AT attribute. Generally, tracking the position of the mouse is the reason for defining a REGION. The MOUSEX and MOUSEY functions can be used to determine the exact position of the mouse when the event occurs. Use of the IMM attribute causes some excess code and speed overhead at runtime, so it should be used only when necessary. This control cannot receive input focus.

A REGION with the DRAGID attribute can serve as a drag-and-drop host, providing information to be moved or copied to another control. A REGION with the DROPID attribute can serve as a drag-and-drop target, receiving information from another control. These attributes work together to specify drag-and-drop "signatures" that define a valid target for the operation. The DRAGID() and DROPID() functions, along with the SETDROPID procedure, are used to perform the data exchange. Since a REGION can be defined over any other control, you can write drag-and-drop code between any two controls. Simply define REGION controls to handle the required drag-and drop functionality.

Events Generated:

EVENT:Accepted The mouse has been clicked by the user in the region.

A REGION with the IMM attribute also generates the following events:

EVENT:MouseIn The mouse has entered the region.

EVENT:MouseOut The mouse has left the region.

EVENT:MouseMove
The mouse has moved within the region.

A REGION with the DRAGID attribute also generates the following events:

EVENT:Dragging The mouse cursor is over a potential drag target.

EVENT:Drag The mouse cursor has been released over a drag target.

A REGION with the DROPID attribute also generates the following events:

EVENT:Drop The mouse cursor has been released over a drag target.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           REGION,AT(10,100,20,20),USE(?R1)
           REGION,AT(100,100,20,20),USE(?R2),CURSOR(CURSOR:Wait)
           REGION,AT(10,200,20,20),USE(?R3),IMM
           REGION,AT(100,200,20,20),USE(?R4),COLOR(COLOR:ACTIVEBORDER)
           REGION,AT(10,300,20,20),USE(?R4),FILL(COLOR:ACTIVEBORDER)
         END
```

SHEET (declare a group of TAB controls)

```
SHEET ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,FULL] [,SCROLL] [,HIDE]
                [,FONT( )] [,DROPID( )] [,WIZARD] [,SPREAD] [,SKIP]
  tabs
END
```

SHEET	Declares a group of TAB controls.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a variable to receive the choice. If this is a string variable, it receives the value of the TAB string (with any accelerator key ampersand stripped out) currently selected by the user. If a numeric variable, it receives the number of the TAB currently selected by the user (the value returned by the CHOICE() function).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the currently selected TAB in the SHEET control.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
FONT	Specifies the display font for the control and the default for all the controls in the SHEET.
ALRT	Specifies "hot" keys active for controls in the SHEET.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
WIZARD	Specifies the SHEET's TAB controls do not appear. The user is moved from TAB to TAB under the program's control (usually with "Next" and "Previous" buttons).
SPREAD	Specifies the TABs are evenly spaced on one line.
SKIP	Specifies the TAB controls in the SHEET do not receive

input focus through the TAB key sequence and may only be accessed with the mouse or accelerator key.

tabs Multiple TAB control declarations.

The **SHEET** control declares a group of TAB controls that offer the user multiple "pages" of controls for the window. The multiple TAB controls in the SHEET structure define the "pages" displayed to the user.

Input focus changes between the SHEET's TAB controls are signalled only to the individual TAB controls affected. This means the events generated when the user changes input focus within a SHEET structure are field-specific events for the affected TAB controls, not the SHEET structure which contains them.

A string variable as the SHEET structure's USE attribute receives the text of the TAB control selected by the user, and the CHOICE(*?Option*) function returns the number of the selected TAB control. If the SHEET structure's USE attribute is a numeric variable, it receives the number of the TAB control selected by the user (the same value returned by the CHOICE function).

Events Generated:

EVENT:Selected One of the SHEET's TAB controls has received input focus.

EVENT:Accepted One of the SHEET's TAB controls has been selected by the user.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

EVENT:TabChanging
Focus is passing to another tab.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           SHEET,AT(0,0,320,175),USE(SelectedTab)
             TAB('Tab One'),USE(?TabOne)
               OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
                 RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
                 RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
               END
               OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
                 RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
                 RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
               END
               PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
               ENTRY(@S8),AT(100,140,32,20),USE(E1)
               PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
               ENTRY(@S8),AT(100,240,32,20),USE(E2)
             END
             TAB('Tab Two'),USE(?TabTwo)
               OPTION('Option 3'),USE(OptVar3)
                 RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
                 RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
               END
               OPTION('Option 4'),USE(OptVar4)
                 RADIO('Radio 3'),AT(60,0,20,20),USE(?R7)
                 RADIO('Radio 4'),AT(80,0,20,20),USE(?R8)
               END
               PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
               ENTRY(@S8),AT(100,140,32,20),USE(E3)
               PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
               ENTRY(@S8),AT(100,240,32,20),USE(E4)
             END
           END
           BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
           BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
         END
```

See Also: TAB

SPIN (declare a spinning list control)

```
SPIN(picture) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]  [,HLP( )] [,SKIP]
     [,FONT( )] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE][,READONLY] [,REQ] [,IMM] [,TIP( )]
     [DROPID( )] [, | LEFT    | ] [, | INS   | ] , | RANGE()[,STEP] | [,   | UPR   |]
                    | RIGHT   |    | OVR |    | FROM( )          |    | CAP   |
                    | CENTER  |
                    | DECIMAL|
```

SPIN	Places a “spinning” list of data items on the WINDOW or TOOLBAR.
picture	A display picture token that specifies the format for the data displayed in the control.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW’s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code or the label of the variable that receives the value selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies “hot” keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

READONLY	Specifies the control does not allow data entry.
REQ	Specifies the control may not be left blank or zero.
IMM	Specifies immediate event generation whenever the user presses any key.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
LEFT	Specifies that the data is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the data is right justified within the area specified by the AT attribute.
CENTER	Specifies that the data is centered within the area specified by the AT attribute.
DECIMAL	Specifies that the data is aligned on the decimal point within the area specified by the AT attribute.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
RANGE	Specifies the range of values the user may choose.
STEP	Specifies the increment/decrement amount of the choices within the specified RANGE. If omitted, the STEP is 1.0.
FROM	Specifies the origin of the choices displayed for the user.
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) entry.

The **SPIN** control places a "spinning" list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The "spinning" list displays only the current selection with a pair of buttons to the right to allow the user to "spin" through the available selections (similar to a slot machine wheel).

If the SPIN control offers the user regularly spaced numeric choices, the RANGE attribute specifies the valid range of values from which the user may choose. The STEP attribute then works in conjunction with RANGE to increment/decrement those values by the specified amount. If the choices are not regular, or are string values, the FROM attribute is used instead of RANGE and STEP. The FROM attribute provides the SPIN control its list of choices from a memory QUEUE or a string. Using the FROM attribute, you may provide the user any type of choices in the SPIN control.

The user may select an item from the list or type in the desired value, so this control also acts as an ENTRY control.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has selected a value from the control.
EVENT:Rejected	The user has entered an invalid value for the entry picture.
EVENT:NewSelection	The user has changed the displayed value.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           SPIN(@S8),AT(0,0,20,20),USE(SpinVar1),FROM(Que)
           SPIN(@N3),AT(20,0,20,20),USE(SpinVar2),RANGE(1,999),KEY(F10Key)
           SPIN(@N3),AT(40,0,20,20),USE(SpinVar3),RANGE(5,995),STEP(5)
           SPIN(@S8),AT(60,0,20,20),USE(SpinVar4),FROM(Que),HLP('Check4Help')
           SPIN(@S8),AT(80,0,20,20),USE(SpinVar5),FROM(Que),MSG('Button 3')
           SPIN(@S8),AT(100,0,20,20),USE(SpinVar6),FROM(Que),FONT('Arial',12)
           SPIN(@S8),AT(120,0,20,20),USE(SpinVar7),FROM(Que),DROP
           SPIN(@S8),AT(140,0,20,20),USE(SpinVar8),FROM(Que),HVSCROLL,VCR
           SPIN(@S8),AT(160,0,20,20),USE(SpinVar9),FROM(Que),IMM
           SPIN(@S8),AT(180,0,20,20),USE(SpinVar10),FROM(Que),CURSOR(CURSOR:Wait)
           SPIN(@S8),AT(200,0,20,20),USE(SpinVar11),FROM(Que),ALRT(F10Key)
           SPIN(@S8),AT(220,0,20,20),USE(SpinVar12),FROM(Que),LEFT
           SPIN(@S8),AT(240,0,20,20),USE(SpinVar13),FROM(Que),RIGHT
           SPIN(@S8),AT(260,0,20,20),USE(SpinVar14),FROM(Que),CENTER
           SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar15),FROM(Que),DECIMAL
         END
```

STRING (declare a window string control)

```
STRING(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,FONT( )] [,FULL] [,SCROLL] [,HIDE]
    [,TRN] [,DROPID( )] [, | LEFT    | ]
                           | RIGHT   |
                           | CENTER  |
                           | DECIMAL |
```

STRING	Places the *text* on the WINDOW or TOOLBAR.
text	A string constant containing the text to display, or a display picture token to format the variable specified in the USE attribute.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code, or a variable whose contents are displayed in the format of the picture token declared instead of string text.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FONT	Specifies the font used to display the text.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
TRN	Specifies the text or USE variable characters transparently display over the background.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.

DECIMAL Specifies that the text is aligned on the decimal point within the area specified by the AT attribute.

The **STRING** control places the *text* on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

If the *text* parameter is a picture token instead of a string constant, the contents of the variable named in the USE attribute are formatted to that display picture, at the position and size specified by the AT attribute. This makes the STRING with a USE variable a "display-only" control for the variable. The data displayed in the STRING is automatically refreshed every time through the ACCEPT loop, whether the AUTO attribute is present or not.

There is a difference between ampersand (&) use in STRING and PROMPT controls. An ampersand in a STRING displays as part of the *text*, while an ampersand in a PROMPT defines the prompt's "hot" letter.

A STRING with the TRN attribute displays characters transparently, without obliterating the background. This means only the pixels required to create each character are written to screen. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

This control cannot receive input focus.

Events Generated:

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           STRING('String Constant'),AT(10,0,20,20),USE(?S1)
           STRING(@S30),AT(10,20,20,20),USE(StringVar1)
           STRING(@S30),AT(10,20,20,20),USE(StringVar2),CURSOR(CURSOR:Wait)
           STRING(@S30),AT(10,20,20,20),USE(StringVar3),FONT('Arial',12)
         END
```

TAB (declare a page of a SHEET control)

```
TAB( text ) [,CURSOR( )] [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,REQ] [DROPID( )] [,TIP( )]
  controls
END
```

TAB	Declares a group of controls that constitute one of the multiple "pages" of controls contained within a SHEET structure.
text	A string constant containing the text to display on the TAB.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	Specifies a field equate label to reference the control in executable code.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the TAB has focus.
HLP	Specifies a string constant containing the default help system identifier for any control in the TAB.
REQ	Specifies that when another TAB is selected, the runtime library automatically checks all ENTRY controls in the same TAB structure with the REQ attribute to ensure they contain data other than blanks or zeroes.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as "balloon help" when the mouse cursor pauses over the control.
controls	Multiple control declarations. This should not contain any SHEET controls (nested SHEET structures are not supported).

The **TAB** structure declares a group of controls that constitute one of the multiple "pages" of controls contained within a SHEET structure. The multiple TAB controls in the SHEET structure define the "pages" displayed to the user. The SHEET structure's USE attribute receives the *text* of the TAB control selected by the user.

Input focus changes between the SHEET's TAB controls are signalled only to the individual TAB controls affected. This means the events generated when the user changes input focus within a SHEET structure are field-

specific events for the affected TAB controls, not the SHEET structure which contains them.

Events Generated:

EVENT:Selected	The TAB control has received input focus.
EVENT:Accepted	The TAB control has been selected by the user.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           SHEET,AT(0,0,320,175),USE(SelectedTab)
             TAB('Tab One'),USE(?TabOne)
               OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
                 RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
                 RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
               END
               OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
                 RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
                 RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
               END
               PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
               ENTRY(@S8),AT(100,140,32,20),USE(E1)
               PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
               ENTRY(@S8),AT(100,240,32,20),USE(E2)
             END
             TAB('Tab Two'),USE(?TabTwo)
               OPTION('Option 3'),USE(OptVar3)
                 RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
                 RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
               END
               OPTION('Option 4'),USE(OptVar4)
                 RADIO('Radio 3'),AT(60,0,20,20),USE(?R7)
                 RADIO('Radio 4'),AT(80,0,20,20),USE(?R8)
               END
               PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
               ENTRY(@S8),AT(100,140,32,20),USE(E3)
               PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
               ENTRY(@S8),AT(100,240,32,20),USE(E4)
             END
           END
           BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
           BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
         END
```

See Also: SHEET

TEXT (declare a multi-line data entry control)

```
TEXT ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP] [,FONT( )]
     [,REQ] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,READONLY] [,DROPID( )] [,UPR]
     [,TIP( )] [,  | INS  | ] [, | HSCROLL   | ] [, | LEFT   | ]
                   | OVR |    | VSCROLL   |    | RIGHT  |
                              | HVSCROLL  |    | CENTER |
```

TEXT	Places a multi-line data entry field on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable that receives the value entered into the control by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
REQ	Specifies the control may not be left blank or zero.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry.
DROPID	Specifies the control may serve as a drop target for drag-

and-drop actions.

TIP	Specifies the text that displays as “balloon help” when the mouse cursor pauses over the control.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
UPR	Specifies all upper case entry.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the text field when any portion of the data lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the text field when any of the data lies vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the text field when any portion of the data lies outside the visible area.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.

The **TEXT** control places a multi-line data entry field on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The variable specified in the USE attribute receives the data entered when the user has completed data entry and moves on to another control.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has completed data entry in the control.
EVENT:PreAlertKey	The user pressed an ALRT attribute hot key.
EVENT:AlertKey	The user pressed an ALRT attribute hot key.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            TEXT,AT(0,0,40,40),USE(E1),ALRT(F10Key),CENTER
            TEXT,AT(20,0,40,40),USE(E2),KEY(F10Key),HLP('Text4Help')
            TEXT,AT(40,0,40,40),USE(E3),SCROLL,OVR,UPR
            TEXT,AT(60,0,40,40),USE(E4),CURSOR(CURSOR:Wait),RIGHT
            TEXT,AT(80,0,40,40),USE(E5),DISABLE,FONT('Arial',12)
            TEXT,AT(100,0,40,40),USE(E6),HVSCROLL,LEFT
            TEXT,AT(120,0,40,40),USE(E7),REQ,INS,CAP,MSG('Text Field 7')
          END
```

Control Field Attributes

ALRT (set control "hot" keys)

ALRT(*keycode***)**

ALRT Specifies a "hot" key active while the control has focus.

keycode A numeric constant keycode or keycode EQUATE.

The **ALRT** attribute specifies a "hot" key active while the control has focus. When the user presses an ALRT "hot" key for a control, two field-specific events, EVENT:PreAlertKey and EVENT:AlertKey, are generated. If the code executes a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the EVENT:AlertKey, preventing library's default action on the alerted keypress for the control.

You may have multiple ALRT attributes on one control. The ALERT statement and the ALRT attribute of a window or control are completely separate. This means that clearing ALERT keys has no effect on any keys alerted by ALRT attributes.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          ENTRY,AT(6,40),USE(SomeVar1),ALRT(F9Key)    !F9 alerted for control
          ENTRY,AT(60,40),USE(SomeVar2),ALRT(F10Key)  !F10 alerted for control
        END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE FIELD()
    OF ?SomeVar1
      CASE EVENT()
      OF EVENT:PreAlertKey        !Pre-check alert events
        IF NOT SomeVar1
          CYCLE                     !Terminate alert processing on other controls
        END
      OF EVENT:AlertKey             !Alert processing
        DO F9Routine
      END
    OF ?SomeVar2
      CASE EVENT()
      OF EVENT:AlertKey             !Alert processing
        DO F10Routine
      END
    END
  END
```

AT (set control position and size in window)

AT([*x*] [,*y*] [,*width*] [,*height*]**)**

AT	Defines the position and size of a control.
x	An integer constant or constant expression that specifies the horizontal position of the top left corner. If omitted, the runtime library provides a default value (zero).
y	An integer constant or constant expression that specifies the vertical position of the top left corner. If omitted, the runtime library provides a default value (zero).
width	An integer constant or constant expression that specifies the width. If omitted, the runtime library provides a default value.
height	An integer constant or constant expression that specifies the height. If omitted, the runtime library provides a default value.

The **AT** attribute defines the position and size of a control. If any parameter is omitted, the runtime library provides a default value.

The values contained in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
 !Measurement in dialog units
WinOne  WINDOW,AT(0,0,160,400)
          ENTRY,AT(8,40,80,8) !Approx. 2 characters in, 5 down, 20 wide, 1 high
        END

 !Measurement in Tousandths of an Inch
WinOne  WINDOW,AT(0,0,160,400)
          ENTRY,AT(1000,1000,2000,250) !1" in & down, 2" wide, 1/4" high
        END

 !Measurement in Millimeters
WinOne  WINDOW,AT(0,0,160,400)
          ENTRY,AT(100,100,200,50) !1 cm in and down, 2 cm wide, 50 mm high
        END

 !Measurement in Points
WinOne  WINDOW,AT(0,0,160,400)
          ENTRY,AT(72,72,144,18) !1" in & down, 2" wide, 1/4" high
        END
```

BOXED (set window controls group border)

BOXED

The **BOXED** attribute specifies a single-track border around a GROUP or OPTION structure. The *text* parameter of the GROUP or OPTION control appears in a gap at the top of the border box. If BOXED is omitted, the *text* parameter of the GROUP or OPTION control is not displayed on screen.

CAP, UPR (set display case)

CAP
UPR

The **CAP** and **UPR** attributes specify the automatic case of text entered into ENTRY or TEXT controls when the MASK attribute is on the window. UPR specifies all upper case.

The CAP attribute specifies "Proper Name Capitalization," where the first letter of each word is capitalized and all other letters are lower case. The user can override this default behavior by pressing the SHIFT key to allow an upper case letter in the middle of a name (allowing for names such as, "McDowell") or SHIFT while CAPS-LOCK is on, forcing a lower case first letter (allowing for names such as, "von Richtofen").

CHECK (set on/off ITEM)

CHECK

The **CHECK** attribute specifies an ITEM that may be either ON or OFF. When ON, a check appears to the left of the menu selection and the USE variable receives the value one (1). When OFF, the check to the left of the menu selection disappears and the USE variable receives the value zero (0).

CLASS (set .VBX custom control class)

CLASS(*file* [,*name*] **)**

CLASS	The specifies the filename and type of .VBX custom control.
file	A string constant containing the name of the .VBX file (including the .VBX extension) in which the custom control is implemented.
name	A string constant containing the name of the custom control type from the .VBX file. If omitted, the first control type defined in the .VBX file is used.

The **CLASS** attribute specifies the filename and type of .VBX custom control. The *name* parameter identifies the specific control to use in a .VBX that contains multiple controls.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          CUSTOM,AT(0,0,120,320),CLASS('graph.vbx','graph'),'graphstyle'('2')
        END
```

COLOR (set control display color)

COLOR(*rgb*)

COLOR Specifies display color.

rgb A LONG or ULONG integer constant, or constant EQUATE, containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2), or an EQUATE for a standard Windows color value.

The **COLOR** attribute specifies the display color of a BOX, LINE, ELLIPSE, or REGION control. On a BOX, ELLIPSE, or REGION, the color specified is the color used for the border.

EQUATEs for Windows' standard colors are contained in the EQUATES.CLW file. Windows automatically finds the closest match to the specified *rgb* color value for the hardware on which the program is run.

Windows standard colors may be reconfigured by the user in the Windows Control Panel. Any control using a Windows standard color is automatically repainted with the new color when this occurs.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          BOX,AT(20,20,20,20),COLOR(COLOR:ACTIVEBORDER)
                                                  !Windows' active border color
          BOX,AT(100,100,20,20),COLOR(00FF0000h)  !Blue
          BOX,AT(140,140,20,20),COLOR(0000FF00h)  !Green
          BOX,AT(180,180,20,20),COLOR(000000FFh)  !Red
        END
```

COLUMN (set list box highlight bar)

COLUMN

The **COLUMN** attribute specifies a field-by-field highlight bar on a LIST or COMBO control with multiple display columns.

CURSOR (set control mouse cursor type)

CURSOR(*file***)**

CURSOR Specifies a mouse cursor to display for the control.

file A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the control.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital “I” like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow
CURSOR:DragWE	Double-headed horizontal arrow

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          REGION,AT(20,20,20,20),CUSOR(CURSOR:IBeam)
          REGION,AT(100,100,20,20),CURSOR('Custom.CUR')
        END
```

DEFAULT (set enter key button)

DEFAULT

The **DEFAULT** attribute specifies a BUTTON that is automatically pressed when the user presses the ENTER key. Only one active BUTTON on a window should have this attribute.

DISABLE (set control dimmed at open)

DISABLE

The **DISABLE** attribute specifies a control that is disabled when the WINDOW or APPLICATION is opened. The disabled control may be activated with the ENABLE statement.

DROP (set list box behavior)

DROP(*count***)**

DROP	Specifies the list appears only when the user presses an arrow cursor key or clicks on the drop icon.
count	An integer constant that specifies the number of elements displayed.

The **DROP** attribute specifies that the selection list appears only when the user presses an arrow cursor key or clicks on the drop icon to the right of the currently selected value display. Once it drops into view, the list displays *count* number of elements. If the DROP attribute is omitted, the LIST or COMBO control always displays the number of data items specified by the *height* parameter of the control's AT of the selection list.

The DROP attribute does not work on a WINDOW with the MODAL attribute and should not be used.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?L7),FROM(Que1),DROP(6)
          COMBO(@S8),AT(120,120,20,20),USE(?C7),FROM(Que2),DROP(8)
        END
```

DRAGID (set drag-and-drop host signatures)

DRAGID(*signature* [, *signature*] **)**

DRAGID Specifies a LIST or REGION control that can serve as a drag-and-drop host.

signature A string constant containing an identifier used to indicate valid drop targets. Any *signature* that begins with a tilde (~) indicates that the information can also be dragged to an external (Clarion) program. A single DRAGID may contain up to 16 *signatures*.

The **DRAGID** attribute specifies a LIST or REGION control that can serve as a drag-and-drop host. DRAGID works in conjunction with the DROPID attribute. The DRAGID *signature* strings (up to 16) define validation keys to match against the *signature* parameters of the target control's DROPID. This provides control over where successful drag-and-drop operations are allowed.

A drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute. For a successful drag-and-drop operation, both controls must have at least one identical *signature* string in their respective DRAGID and DROPID attributes.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('FromList1')
                                           !Allows drags, but not drops
          LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('FromList1')
                                           !Allows drops from List1, but no drags
        END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag             !When a drag event is attempted
      IF DRAGID()             ! check for success
        SETDROPID(Que1)       ! and setup info to pass
      END
    OF EVENT:Drop             !When drop event is successful
      Que2 = DROPID()         ! get dropped info
      ADD(Que2)               ! and add it to the queue
    END
  END
```

See Also: DROPID

DROPID (set drag-and-drop target signatures)

DROPID(*signature* [, *signature*] **)**

DROPID Specifies a control that can serve as a drag-and-drop target.

signature A string constant containing an identifier used to indicate valid drag hosts. A single DROPID may contain up to 16 *signatures*. Any *signature* that begins with a tilde (~) indicates that the information can also be dropped from an external (Clarion) program. A DROPID *signature* of '~FILE' indicates the target accepts a comma-delimited list of filenames dragged from the Windows File Manager.

The **DROPID** attribute specifies a control that can serve as a drag-and-drop target. DROPID works in conjunction with the DRAGID attribute. The DROPID *signature* strings (up to 16) define validation keys to match against the *signature* parameters of the host control's DRAGID. This provides control over where successful drag-and-drop operations are allowed.

A drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute. For a successful drag-and-drop operation, both controls must have at least one identical *signature* string in their respective DRAGID and DROPID attributes.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('FromList1')
                                        !Allows drags, but not drops
          LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('FromList1','~FILE')
                                !Allows drops from List1 or the Window File Manager,
                                ! but no drags
        END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag               !When a drag event is attempted
      IF DRAGID()               ! check for success
        SETDROPID(Que1)         ! and setup info to pass
      END
    OF EVENT:Drop               !When drop event is successful
      Que2 = DROPID()           ! get dropped info
      ADD(Que2)                 ! and add it to the queue
    END
  END
```

See Also: DRAGID

FILL (set display fill color)

FILL(*rgb*)

FILL	Specifies display fill color.
rgb	A LONG or ULONG integer constant, or constant EQUATE, containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **FILL** attribute specifies the display fill color of a BOX, ELLIPSE, or REGION control. If omitted, the control is not filled with color.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          BOX,AT(20,20,20,20),FILL(COLOR:ACTIVEBORDER)
                                                !Windows' active border color
          BOX,AT(100,100,20,20),FILL(00FF0000h)  !Blue
          BOX,AT(140,140,20,20),FILL(0000FF00h)  !Green
          BOX,AT(180,180,20,20),FILL(000000FFh)  !Red
        END
```

FIRST, LAST (set MENU or ITEM position)

FIRST
LAST

The **FIRST** and **LAST** attributes specify menu selection positioning within the global pulldown menu, when a WINDOW's MENUBAR is merged into the global menu. The order of priorities is:

1. Global selections with FIRST attribute
2. Local selections with FIRST attribute
3. Global selections without FIRST or LAST attributes
4. Local selections without FIRST or LAST attributes
5. Global selections with LAST attribute
6. Local selections with LAST attribute

FONT (set control font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the display font for a control.
typeface	A string constant containing the name of the font. If omitted, the default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, the default font color is used.
style	An integer constant, constant expression, or EQUATE specifying the strike weight and style of the font. If omitted, the default font weight is used.

The **FONT** attribute specifies the display font for the control, overriding any FONT specified on the WINDOW.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000h)
FONT:underline	EQUATE (02000h)
FONT:strikeout	EQUATE (04000h)

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?L7),FROM(Que1),FONT('Arial',14,0FFh)
                               !14 point Arial typeface, Red, normal
          LIST,AT(120,120,20,20),USE(?C7),FROM(Que2),FONT('Arial',14,0,700)
                               !14 point Arial typeface, Black, Bold
          LIST,AT(120,240,20,20),USE(?C7),FROM(Que2),FONT('Arial',14,0,700+01000h)
                               !14 point Arial typeface, Black, Bold Italic
        END
```

FORMAT (set LIST or COMBO layout)

FORMAT(*format string***)**

FORMAT Specifies the display format of the data in the LIST or COMBO control.

format string A string constant specifying the display format.

The **FORMAT** attribute specifies the display format of the data in the LIST or COMBO control. The *format string* contains the information for single or multi-column formatting of the data.

The *format string* contains "field-specifiers" which map to the fields of the QUEUE. Multiple "field-specifiers" may be grouped together as a "field-group" in square brackets (**[ ]**) to display as a single unit.

Only the fields in the QUEUE for which there are "field-specifiers" are included in the display. This means that, if there are two fields specified in the *format string* and three fields in the QUEUE, only the two specified in the *format string* are displayed in the LIST or COMBO control.

The following describes the components allowed in a *format string*:

"Field-specifier" format: *width justification* [**(***indent***)**] [*modifiers*]

width A required integer defining the width of the field. Specified in dialog units.

justification A single capital letter (**L** , **R** , **C** , or **D**) that specifies **L**eft, **R**ight, **C**enter, or **D**ecimal justification. One is required.

indent An optional integer, enclosed in parentheses, that specifies the indent from the justification. This may be negative. With left (**L**) justification, *indent* defines a left margin; with right (**R**) or decimal (**D**), it defines a right margin; and with center (**C**), it defines an offset from the center of the field (negative = left offset).

modifiers: Optional special characters (listed below) to modify the display format of the field or group. Multiple *modifiers* may be used on one field or group.

* An asterisk indicates color information for the field is contained in four LONG fields that immediately follow the data field in the QUEUE (or FROM attribute string). The four colors are normal foreground, normal background, selected foreground, and selected background (in that order).

I An I indicates an icon number for the field is contained in a LONG field that immediately follows the data field in the QUEUE (or FROM attribute string).The LONG field contains a number that refers to an entry in a list of icons associated with the LIST control through the PROP:IconList runtime property. If an asterisk is also specified for color, this LONG must follow all the color information.

T *[(suppress)]* A T indicates the LIST is a tree control. The tree level is contained in a LONG field that immediately follows the data field in the QUEUE (or FROM attribute string). If an asterisk and I are also specified, this LONG must follow all their LONG fields. The expanded/contracted state of the tree level is determined by the sign of the tree level LONG field's value (positive value=expanded and negative value=contracted).

The optional *suppress* parameter can contain an **L** to suppress the conecting lines between levels, a **B** to suppress expansion boxes, and an **I** to suppress level indentation (which also implicitly suppresses both lines and boxes).

~header~ [justification [(indent)]]
A header string enclosed in tildes, followed by optional justification and/or indent, displays the header at the top of the list. The header uses the same justification and indent as the field, if not specifically overidden.

@picture@ The *picture* formats the field for display. The trailing @ is required to define the end of the *picture*, so that display pictures like @N12~Kr~ can be used in the format string without creating ambiguity.

? A question mark defines the locator field for a COMBO list box with a selector field. For a drop-down multi-column list box, this is the value displayed in the current-selection box.

#number# The *number* enclosed in pound signs (#) indicates the QUEUE field to display. Following fields in the format string without an explicit #*number*# are taken in order from the fields following the #*number*# field. For example, #2# on the first field in the format string indicates starting with the second field in the QUEUE, skipping the first. If the number of fields specified in the format string are >= the number of fields in the QUEUE, the format "wraps around" to the start of the QUEUE.

_ An underscore underlines the field.

/ A slash causes the next field to appear on a new line (only used on a field within a group).

|	A vertical bar places a vertical line to the right of the field.
M	An M allows the field or group of fields to be dynamically re-sized at runtime. This allows the user to drag the right vertical bar (if present) or right edge of the data area.
F	An F creates a fixed column in the list that stays on screen when the user horizontally pages through the fields (by the HSCROLL attribute). Fixed fields or groups must be at the start of the list. This is ignored if placed on a field within a group.
S(*integer*)	An S followed by an *integer* in parentheses adds a scroll bar to the group. The *integer* defines the total number of dialog units to scroll. This allows large fields to be displayed in a small column width. This is ignored if placed on a field within a group.

"Field-group" format: **[** *multiple field-specifiers* **]** [**(***size***)**] [*modifiers*]

multiple field-specifiers	A list of field-specifiers contained in square brackets (**[ ]**) that cause them to be treated as a single display unit.
size	An optional integer, enclosed in parentheses, that specifies the default width of the group. If omitted, the size is calculated from the enclosed fields.
modifiers	The "field-group" *modifiers* act on the entire group of fields. These are the same *modifiers* listed above for a field (except the *, I, and T *modifiers* which are not appropriate to groups).

Example:

```
      PROGRAM
      MAP
        RandomAlphaData(*STRING)
      END

TreeDemo        QUEUE,PRE()         !Data list box FROM queue
FName             STRING(20)
ColorNFG          LONG              !Normal Foreground color for FName
ColorNBG          LONG              !Normal Background color for FName
ColorSFG          LONG              !Selected Foreground color for FName
ColorSBG          LONG              !Selected Background color for FName
IconField         LONG              !Icon number for FName
TreeLevel         LONG              !Tree Level
LName             STRING(20)
Init              STRING(4)
                END

Win WINDOW('List Boxes'),AT(0,0,366,181),SYSTEM,DOUBLE
      LIST,AT(0,34,366,146),FROM(TreeDemo),USE(?Show),HVSCROLL,    |
            FORMAT('80L*IT~First Name~*80L~Last Name~16C~Initials~')
    END

  CODE
  LOOP X# = 1 TO 20
    RandomAlphaData(FName)
    ColorNFG = COLOR:White          !Assign FNAME's colors
    ColorNBG = COLOR:Maroon
    ColorSFG = COLOR:Yellow
    ColorSBG = COLOR:Blue
    IconField = ((X#-1) % 4) + 1    !Assign icon number
    TreeLevel = ((X#-1) % 4) + 1    !Assign tree level
    RandomAlphaData(LName)
    RandomAlphaData(Init)
    ADD(TD)
  END
  OPEN(Win)
  ?Show{PROP:iconlist,1} = ICON:VCRback          !Icon 1 = <
  ?Show{PROP:iconlist,2} = ICON:VCRrewind        !Icon 2 = <<
  ?Show{PROP:iconlist,3} = ICON:VCRplay          !Icon 3 = >
  ?Show{PROP:iconlist,4} = ICON:VCRfastforward   !Icon 4 = >>
  ACCEPT
  END

RandomAlphaData PROCEDURE(Field)       !MAP Prototype is: RandomAlphaData(*STRING)
  CODE
  Z# = RANDOM(1,SIZE(Field))           !Random fill size
  LOOP Z# = 1 to Y#                    !Fill each character with
    Field[Z#] = CHR(RANDOM(97,122))    ! a random lower case letter
  END
```

FROM (set window listbox data source)

FROM(*source***)**

FROM	Specifies the source of the data elements displayed in a LIST, COMBO, or SPIN.
source	The label of a QUEUE, a field within a QUEUE, or a string constant containing the data items to display in the list.

The **FROM** attribute specifies the source of the data elements displayed in a LIST, COMBO, or SPIN.

For a SPIN control, the *source* would usually be a QUEUE field or string. If the *source* is a QUEUE with multiple fields, only the first field is displayed in the SPIN.

For LIST and COMBO controls, the data elements are formatted for display according to the information in the FORMAT attribute. If the label of a QUEUE is specified as the *source*, all fields in the QUEUE are displayed. If the label of one field in a QUEUE is specified as the *source*, only that field is displayed.

If a string constant is specified as the *source*, the individual data elements to display in the LIST must be delimited by a vertical bar (|) character. To include a vertical bar as part of one data element, place two adjacent vertical bars in the string (||), and only one will be displayed. To indicate that an element is empty, place at least one blank space between the two vertical bars delimiting the elements (| |).

Example:

```
Que1  QUEUE,PRE(Q1)
F1       LONG
F2     STRING(8)
      END

Win1  WINDOW,AT(0,0,160,400)
        LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),FORMAT('5C~List~15L~Box~'),COLUMN
        COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
        SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q1:F1)
        SPIN(@S4),AT(280,0,20,20),USE(SpinVar2),FROM('Mr.|Mrs.|Ms.|Dr.')
      END
```

FULL (set full-screen)

FULL

The **FULL** attribute specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.

FULL may not be specified for TOOLBAR controls.

HIDE (set control hidden at open)

HIDE

The **HIDE** attribute specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

HLP (set control's on-line help identifier)

HLP(*helpID***)**

HLP	Specifies the *helpID* for the control.
helpID	A string constant specifying the key used to access the Help system. This may be either a Help keyword or a "context string."

The **HLP** attribute specifies the *helpID* for the control. Help, if available, is automatically displayed by Windows whenever the user presses F1. If the user presses F1 to request help when the control has input focus, the library uses the control's *helpID* to search the help file until an object with that *helpID* is found.

The *helpID* may contain a Help keyword or a "context string." A Help keyword is a keyword or phrase that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *helpID*, followed by a unique identifier (no spaces allowed) associated with exactly one help topic. When the user presses F1, the help file is opened at the specific topic associated with that "context string." If the tilde is missing, the *helpID* is assumed to be a help keyword.

Example:

```
Win1  WINDOW
        ENTRY(@s30),USE(SomeVariable),HLP('~Entry1Help')!A help context string
        ENTRY(@s30),USE(SomeVariable),HLP('Control Two Help')!A help keyword
      END
```

HSCROLL, VSCROLL, HVSCROLL (set control scroll bars)

HSCROLL
VSCROLL
HVSCROLL

The **HSCROLL**, **VSCROLL**, and **HVSCROLL** attributes place scroll bars on a COMBO, LIST, IMAGE, or TEXT control. HSCROLL adds a horizontal scroll bar to the bottom; VSCROLL adds a vertical scroll bar on the right side, and HVSCROLL adds both.

The vertical scroll bar allows a mouse to scroll the control's display up or down. The horizontal scroll bar allows a mouse to scroll the control's display left or right. The scroll bars appear whenever any scrollable portion of the control lies outside the visible area on screen.

When you place VSCROLL on a LIST with the IMM attribute, the vertical scroll bar is always present, even when the list is not full. When the user clicks on the scroll bar, events are generated, but the list contents do not move (executable code should perform this task). You can interrogate the PROP:VscrollPos property to determine the scroll thumb's position in the range 0 (top) to 100 (bottom).

ICON (set control icon)

ICON([*file*] **)**

ICON	Specifies an icon to display as the control.
file	A string constant or EQUATE containing the name of an .ICO file or Windows standard icon to display. The .ICO file is automatically linked into the .EXE as a resource.

The **ICON** attribute specifies an icon to display as the control. The icon is displayed on the button face of the control. The ICON attribute may be specified on a BUTTON, RADIO, or CHECK control. For RADIO and CHECK controls, the ICON attribute creates "latched" pushbuttons, where the control button appears "down" when on and "up" when off.

EQUATE statements for the Windows-standard icons are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

ICON:None	No icon
ICON:Application	
ICON:Question	?
ICON:Exclamation	!
ICON:Asterisk	*
ICON:VCRtop	>>|
ICON:VCRrewind	<<
ICON:VCRback	<
ICON:VCRplay	>
ICON:VCRfastforward	>>
ICON:VCRbottom	|<<
ICON:VCRlocate	?

Example:

```
MDIChild  WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            OPTION('Option'),USE(OptVar)
              RADIO('Radio  1'),AT(120,0,20,20),USE(?R1),ICON('Radio1.ICO')
              RADIO('Radio  2'),AT(140,0,20,20),USE(?R2),ICON('Radio2.ICO')
            END
           CHECK('&A'),AT(0,120,20,20),USE(?C7),ICON(ICON:Asterisk)
           BUTTON('&1'),AT(120,0,20,20),USE(?B7),ICON(ICON:Question)
         END
```

IMM (set immediate event notification)

IMM

The **IMM** attribute specifies immediate generation of an event.

On a REGION control, the IMM attribute generates an event whenever the mouse enters, moves within, or leaves the area specified by the REGION's AT attribute. The exact position of the mouse can be deteremined by the MOUSEX and MOUSEY functions.

On a BUTTON control, the IMM attribute indicates the BUTTON generates an event when the left mouse button is pressed down on the control, instead of on its release. The event is continuously generated as along as the user keeps the mouse button pressed.

The IMM attribute specifies immediate event generation each time the user presses any keystroke on a LIST or COMBO control, usually requiring the QUEUE to be re-filled. When the user presses a printable character, EVENT:NewSelection is generated. It does the same thing on an ENTRY or SPIN control.

INS, OVR (set typing mode)

INS
OVR

The **INS** and **OVR** attributes specify the typing mode for an ENTRY or TEXT control when the MASK attribute is present on the window. INS specifies insert mode while OVR specifies overwrite mode. These modes are only active on windows with the MASK attribute.

KEY (set control execution keycode)

KEY(*keycode***)**

KEY	Specifies a “hot” key for the control
keycode	A Clarion Keycode or keycode equate label.

The **KEY** attribute specifies a “hot” key to immediately give focus to the control or execute the control’s associated action.

The following controls receive focus:

COMBO
CUSTOM
ENTRY
GROUP
LIST
OPTION
PROMPT
SPIN
TEXT

The following controls both receive focus and immediately execute:

BUTTON
CHECK
CUSTOM
RADIO
MENU
ITEM

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),KEY(F1Key)
          LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),KEY(F2Key)
          SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),KEY(F3Key)
          TEXT,AT(20,0,40,40),USE(E2),KEY(F4Key)
          PROMPT('Enter &Data in E2:'),AT(10,200,20,20),USE(?P2),KEY(F5Key)
          ENTRY(@S8),AT(100,200,20,20),USE(E2),KEY(F6Key)
          BUTTON('&1'),AT(120,0,20,20),USE(?B7),KEY(F7Key)
          CHECK('&A'),AT(0,120,20,20),USE(?C7),KEY(F8Key)
          OPTION('Option'),USE(OptVar),KEY(F9Key)
            RADIO('Radio 1'),AT(120,0,20,20),USE(?R1),KEY(F10Key)
            RADIO('Radio 2'),AT(140,0,20,20),USE(?R2),KEY(F11Key)
          END
        END
```

LEFT, RIGHT, CENTER, DECIMAL (set display justification)

LEFT([*indent*] **)**
RIGHT([*indent*] **)**
CENTER([*indent*] **)**
DECIMAL([*indent*] **)**

indent An integer constant specifying the amount of offset from the justification point. This is in dialog units.

The **LEFT**, **RIGHT**, **CENTER**, and **DECIMAL** attributes specify the justification of data displayed. LEFT specifies left justification, RIGHT specifies right justification, CENTER specifies centered text, and DECIMAL specifies numeric data aligned on the decimal point.

The *indent* parameter on the CENTER attribute specifies an offset from the center (negative = left offset). On the DECIMAL attribute, *indent* specifies the offset of the decimal point from the right.

The CHECK and RADIO controls allow LEFT or RIGHT only (without an *indent* parameter). The TEXT control allows only LEFT(*indent*), RIGHT(*indent*), or CENTER(*indent*).

The following controls allow LEFT(*indent*), RIGHT(*indent*), CENTER(*indent*), or DECIMAL(*indent*):

COMBO
ENTRY
LIST
SPIN
STRING

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),RIGHT(4)
          LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),CENTER
          SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),DECIMAL(8)
          TEXT,AT(20,0,40,40),USE(E2),LEFT(8)
          ENTRY(@S8),AT(100,200,20,20),USE(E2),LEFT(4)
          CHECK('&A'),AT(0,120,20,20),USE(?C7),LEFT
          OPTION('Option'),USE(OptVar)
            RADIO('Radio  1'),AT(120,0,20,20),USE(?R1),LEFT
            RADIO('Radio  2'),AT(140,0,20,20),USE(?R2),RIGHT
          END
        END
```

MARK (set multiple selection mode)

MARK(*flag*)

MARK Enables multiple items selection.

flag The label of a QUEUE field.

The **MARK** attribute enables multiple items selection from a LIST or COMBO control. When an item in the LIST is selected, the appropriate *flag* field is set to true (1). Each marked entry is automatically highlighted in the LIST or COMBO. Changing the value of the *flag* field also changes the screen display for the related LIST or COMBO entry.

If the MARK attribute is specified on the LIST or COMBO, the IMM attribute may not be.

Example:

```
Que1      QUEUE,PRE(Q1)
MarkFlag    BYTE
F1          LONG
F2          STRING(8)
          END

WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?L1),FROM(Q1:F1),MARK(Q1:MarkFlag)
          COMBO(@S8),AT(120,120,,),USE(?C1),FROM(Q1:F2),MARK(Q1:MarkFlag)
        END
```

MSG (set control status bar message)

MSG(*text*)

MSG	Specifies *text* to display in the status bar.
text	A string constant containing the message to display in the status bar.

The **MSG** attribute specifies the *text* to display in the first zone of the status bar when the control has focus.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),MSG('Enter or Select')
          LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),MSG('Select One')
          SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),MSG('Choose One')
          TEXT,AT(20,0,40,40),USE(E2),MSG('Enter Text')
          ENTRY(@S8),AT(100,200,20,20),USE(E2),MSG('Enter Data')
          CHECK('&A'),AT(0,120,20,20),USE(?C7),MSG('On or Off')
          OPTION('Option 1'),USE(OptVar),MSG('Pick One or Two')
            RADIO('Radio 1'),AT(120,0,20,20),USE(?R1)
            RADIO('Radio 2'),AT(140,0,20,20),USE(?R2)
          END
          OPTION('Option'),USE(OptVar)
            RADIO('Radio 1'),AT(120,40,20,20),USE(?R1),MSG('Pick One')
            RADIO('Radio 2'),AT(140,40,20,20),USE(?R2),MSG('Pick Two')
          END
        END
```

NOBAR (set no highlight bar)

NOBAR

The **NOBAR** attribute specifies the currently selected element in the LIST is only highlighted when the LIST control has focus.

PASSWORD (set data non-display)

PASSWORD

The **PASSWORD** attribute specifies non-display of the data entered in the ENTRY control. When the user types in data, asterisks are displayed on screen for each character entered.

RANGE (set range limits)

RANGE(*lower,upper***)**

RANGE	Specifies the valid range of data values the user may select in a SPIN control, or the range of values displayed in a PROGRESS control.
lower	A numeric constant that specifies the lower inclusive limit of valid data.
upper	A numeric constant that specifies the upper inclusive limit of valid data.

The **RANGE** attribute specifies the valid range of data values the user may select in a SPIN control. RANGE also defines the range of values that are displayed in a PROGRESS control.This attribute works in conjunction with the STEP attribute on SPIN controls. On a SPIN control, the STEP attribute provides the user with the valid choices within the range.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          SPIN(@N4.2),AT(280,0,20,20),USE(SpinVar1),RANGE(.05,9.95),STEP(.05)
          SPIN(@n3),AT(280,0,20,20),USE(SpinVar2),RANGE(5,995),STEP(5)
        END
```

READONLY (set display-only)

READONLY

The **READONLY** attribute specifies a display-only COMBO, ENTRY, SPIN or TEXT control. The control may receive input focus with the mouse, but may not enter data. If the user attempts to change the displayed value, a beep warns the user that data entry is not allowed.

REQ (set required entry)

REQ

The **REQ** attribute specifies an ENTRY or TEXT control that may not be left blank or zero. The REQ attribute on an ENTRY or TEXT control is not checked until a BUTTON with the REQ attribute is pressed, or the INCOMPLETE() function is called.

When a BUTTON with the REQ attribute is pressed, or the INCOMPLETE() function is called, all ENTRY and TEXT controls with the REQ attribute are checked to ensure they contain data. The first control encountered in this check that does not contain data inmediately receives input focus.

RIGHT (set MENU position)

RIGHT

The **RIGHT** attribute specifies the MENU is placed at the right end of the action bar.

ROUND (set round-cornered window BOX)

ROUND

The **ROUND** attribute specifies a BOX control with rounded corners.

SCROLL (set scrolling control)

SCROLL

The **SCROLL** attribute specifies a control that moves with the window when the WINDOW scrolls. This allows “virtual” windows larger than the physical video display.

The presence of the SCROLL attribute means that the control stays fixed at a position in the window relative to the top left corner of the virtual window, whether that position is currently in view or not. This means that the control appears to move as the window scrolls.

If the SCROLL attribute is omitted, the control stays fixed at a position in the window relative to the top left corner of the currently visible portion of the window. This means that the control appears to stay in the same position on screen while the rest of the window scrolls. This is useful for controls which should stay visible to the user at all times (such as Ok or Cancel buttons).

Mixing controls with and without the SCROLL attribute on the same WINDOW can result in multiple controls appearing to occupy the same screen position. This occurs because the controls with SCROLL move and the controls without SCROLL do not. This condition is temporary and scrolling the window will correct the situation. The situation can be avoided entirely by careful placement of controls in the window. For example, you can place all controls without SCROLL at the bottom of the window then place all controls with SCROLL above them extending to the right and left. This would create a window that only scrolls horizontally.

SEPARATOR (set separator line ITEM)

SEPARATOR

The **SEPARATOR** attribute specifies an ITEM in a MENU that displays a horizontal line to group ITEMs within the MENU. No other attributes may be specified for the ITEM.

SKIP (set Tab key skip)

SKIP

The **SKIP** attribute specifies the control may only be accessed with the mouse or an accelerator key. Controls that allow data entry receive input focus only during data entry and the control does not retain focus. Controls that do not allow data entry do not receive or retain input focus. The effect of this is to create the same behavior as a control in a toolbar. When the mouse cursor is over a control with the SKIP attribute, the control's MSG attribute is displayed in the status bar.

SPREAD (set evenly spaced TAB controls)

SPREAD

The **SPREAD** attribute specifies a SHEET's TAB controls are evenly spaced.

STD (set standard behavior)

STD(*behavior***)**

STD	Specifies standard Windows *behavior*.
behavior	An integer constant or EQUATE specifying the identifier of a standard windows behavior.

The **STD** attribute specifies the control activates some standard Windows action. This action is automatically executed by the runtime library and does not generate an event.

EQUATE statements for the standard Windows actions are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

STD:WindowList	List of open MDI windows
STD:TileWindow	Tile Windows
STD:CascadeWindow	Cascade Windows
STD:ArrangeIcons	Arrange Icons
STD:HelpIndex	Help Contents
STD:HelpSearch	Help Search dialog

Example:

```
MDIChild  WINDOW('Child One'),MDI,SYSTEM,MAX
            MENUBAR
              MENU('Edit'),USE(?EditMenu)
                ITEM('Undo'),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
                ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
                ITEM('Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
                ITEM('Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
              END
            END
            TOOLBAR
              BUTTON('Cut'),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut)
              BUTTON('Copy'),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy)
              BUTTON('Paste'),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste)
            END
          END
```

STEP (set SPIN increment)

STEP(*count***)**

STEP	Specifies a SPIN control RANGE attribute's increment/decrement value.
count	A numeric constant specifying the amount to increment or decrement.

The **STEP** attribute specifies the amount by which a SPIN control's value is incremented or decremented within its valid RANGE. The default STEP value is 1.0.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          SPIN(@N4.2),AT(280,0,20,20),USE(SpinVar1),RANGE(.05,9.95),STEP(.05)
          SPIN(@n3),AT(280,0,20,20),USE(SpinVar2),RANGE(5,995),STEP(5)
        END
```

TIP (set "balloon help" text)

TIP(*string* **)**

TIP	Specifies the text to display when the mouse cursor pauses over the control.
string	A string constant that specifies the text to display.

The **TIP** attribute on a control specifies the text.to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Although it is valid on any control that can gain focus for user input, this attribute is most commonly used on BUTTON controls with the ICON attribute that are placed on the TOOLBAR. This allows the user to quickly determine the control's purpose without accessing the on-line Help system.

Automatic TIP attribute display can be disabled for any single control or window by setting the PROP:NoTips undeclared property to one (1). It can be disabled for an entire application by setting the PROP:NoTips for the built-in variable SYSTEM to one (1).

The time delay before TIP display can be set for an entire application by setting the PROP:TipDelay for the built-in variable SYSTEM to the desired delay amount (in hundredths of a second). This is valid only for 16-bit applications; in 32-bit operating systems, the amount of tip delay is an operating system setting under the user's control.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          TOOLBAR
            BUTTON('E&xit'),USE(?MainExitButton),ICON(ICON:hand),TIP('Exit  Window')
            BUTTON('&Open'),USE(?OpenButton),ICON(ICON:Open),TIP('Open  a  File')
          END
          COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
          ENTRY(@S8),AT(100,200,20,20),USE(E2)
        END
```

TRN (set transparent window string)

TRN

The **TRN** attribute on a STRING control specifies the characters display transparently, without obliterating the background over which the STRING is placed. Only the pixels required to create each character are written to the screen. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          IMAGE('PIC.BMP'),USE(?I1),FULL              !Full window image
          STRING('String  Constant'),AT(10,0,20,20),USE(?S1),TRN
                                                      !Transparent string on image
        END
```

USE (set control variable or equate label)

USE(| *label* | [,*number*] [,*equate*] **)**
| *variable* |

USE	Specifies a variable or field equate label for the control.
label	A field equate label to reference the control in executable code.
variable	The label of the field to receive the value entered in the control. This label (with a ? prepended) becomes the field equate label for the control, unless the *equate* parameter is used.
number	An integer constant that specifies the number the compiler equates to the field equate label for the control.
equate	A field equate label to reference the control in executable code when the named *variable* has already been used in the same structure. This provides a mechanism to provide a unique field equate when the *variable* would not.

The **USE** attribute specifies a variable or field equate label for the control. USE with a *label* parameter simply provides a mechanism for executable source code statements to reference the control. Some controls only allow a field equate *label* as the USE parameter, not a *variable*. These controls are: PROMPT, IMAGE, LINE, BOX, ELLIPSE, GROUP, RADIO, REGION, MENU, and BUTTON. USE with a *variable* parameter supplies the control with a variable to update by operator entry. This is applicable to an ITEM with the CHECK attribute, or an ENTRY, OPTION, SPIN, TEXT, LIST, COMBO, CHECK, or CUSTOM.

All controls in an APPLICATION or WINDOW are automatically assigned numbers by the compiler. For an APPLICATION's MENUBAR controls, these numbers start at negative one (-1) and decrement by one (1) for each MENU and ITEM in the MENUBAR. On a WINDOW, these numbers start at one (1) and increment by one (1) for each control in the WINDOW.

The USE attribute's *number* parameter allows you to specify the actual field number the compiler assigns to the control. This *number* also is used as the new starting point for subsequent field numbering for fields without a *number* parameter in their USE attribute. Subsequent controls without a *number* parameter in their USE attribute are incremented (or decremented) relative to the last *number* assigned.

Two or more controls with exactly the same USE variable in one WINDOW or APPLICATION structure would create the same Field Equate Label for all, therefore, when the compiler encounters this condition, all Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion

about which control you really want to reference. It also allows you to deliberately create this condition to display the contents of the variable in multiple controls with different display pictures.

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
          COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
          ENTRY(@S8),AT(100,200,20,20),USE(E2)
        END
```

VALUE (set RADIO control OPTION USE variable assignment)

VALUE(*string* **)**

VALUE Specifies the value assigned to the OPTION structure's USE variable when the RADIO control is selected by the user.

string A string constant that specifies the value to assign.

The **VALUE** attribute specifies the value that is automatically assigned to the OPTION structure's USE variable when the RADIO control is selected by the user. This attribute overrides the RADIO control's *text* parameter.

All automatic type conversion rules apply to the *string* assigned to the OPTION structure's USE variable. Therefore, if the *string* contains only numeric data and the USE variable is a numeric data type, it receives the numeric value of the *string*.

Example:

```
Win WINDOW,AT(0,0,160,400)
      OPTION('Option 1'),USE(OptVar1),MSG('Pick One or Two')
        RADIO('Radio  1'),AT(120,0,20,20),USE(?R1),VALUE('10')!OptVar1 gets 10
        RADIO('Radio  2'),AT(140,0,20,20),USE(?R2),VALUE('20')!OptVar1 gets 20
      END
      OPTION('Option 2'),USE(OptVar2),MSG('Pick One or Two')
        RADIO('Radio  1'),AT(120,0,20,20),USE(?R1),VALUE('10')!OptVar2 gets '10'
        RADIO('Radio  2'),AT(140,0,20,20),USE(?R2),VALUE('20')!OptVar2 gets '20'
      END
    END
```

VCR (set VCR control)

VCR([*field*])

VCR	Places Video Cassette Recorder (VCR) style buttons on a LIST or COMBO control.
field	A field equate label that specifies the ENTRY control to use as a locator for a LIST (not valid on a COMBO).

The **VCR** attribute places **V**ideo **C**assette **R**ecorder (VCR) style buttons on a LIST or COMBO control. The VCR style buttons affect the scrolling characteristics of the data displayed in the LIST or COMBO.

There are six buttons displayed as the VCR:

|<	Top of list	(EVENT:ScrollTop)
<<	Page Up	(EVENT:PageUp)
<	Entry Up	(EVENT:ScrollUp)
>	Entry Down	(EVENT:ScrollDown)
>>	Page Down	(EVENT:PageDown)
>|	Bottom of list	(EVENT:ScrollBottom)

On a LIST control's VCR(*field*), there also appears a button with a question mark (?) in the middle of the other buttons. This is the locator button that gives focus to the control specified by the *field* parameter. When the user enters data and then presses TAB on the locator *field*, the LIST scrolls to its closest matching entry.

Example:

```
MDIChild  WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            LIST,AT(140,0,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR
            ENTRY(@S8),AT(100,200,20,20),USE(E2)
            LIST,AT(140,100,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR(?E2)
          END
```

WIZARD (set "tabless" SHEET control)

WIZARD

The **WIZARD** attribute specifies a SHEET control that does not display its TAB controls. This allows the program to direct the user through each TAB in a specified sequence (usually with "Next" and "Previous" buttons.

Event Processing

Contents

Event-driven Programming

Windows programs are generally event-driven. This means the user causes an event by clicking the mouse on a screen control or pressing a key. Every user action in the program results in Windows sending a message to the program which owns the window telling it what the user has done. Once Windows has sent the message signaling an event to the program, the program has the opportunity to handle the event in the appropriate manner. This basically means the Windows programming paradigm is exactly opposite from the DOS programming paradigm—the operating system (Windows) tells the program what to do, instead of the program telling the operating system what to do.

Writing a Windows program in a programming language other than Clarion becomes very complex, because the program must be coded to explicitly handle every message from Windows. Common tasks, such as re-drawing graphics that have been overwritten by a window that was open and is now closed, must be explicitly coded in the program.

These common tasks could be handled automatically by writing generic procedures to accomplish the task and call them every time the need arises. Of course, in other programming languages, you would have to write these procedures yourself. In Clarion for Windows, they are already written and included as part of our runtime library. The Clarion language, therefore, has persistent graphics commands that do not require an explicit re-draw each time they are overwritten (unlike other languages).

In Clarion Windows programs, most of the messages from Windows are automatically handled internally by the ACCEPT event processor. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by ACCEPT to your Clarion code. The net effect of this is to make your programming job easier by removing the low-level “drudgery” code from your program, allowing you to concentrate on the high-level aspects of programming, instead.

There are two types of events passed on to the program by ACCEPT: **Field-specific** and **Field-independent** events.

A **Field-specific** event occurs when the user presses a key that may require the program to perform a specific action related to that control.

A **Field-independent** event does not relate to any one control but requires some program action (for example, to close a window, quit the program, or change execution threads). Most of these events cause the system to become modal, since they require a response before the program may continue.

ACCEPT (the event processor)

```
ACCEPT
  statements
END
```

ACCEPT The event handler.

statements Executable code statements.

The **ACCEPT** loop is the event handler that processes events generated by Windows for the APPLICATION or WINDOW structures. An ACCEPT loop and a window are bound together, in that, when the window is opened, the next ACCEPT loop encountered will process all events for that window.

ACCEPT operates in the same manner as a LOOP—the BREAK and CYCLE statements can be used within it. The ACCEPT loop cycles for every event that requires program action. ACCEPT waits until the Clarion runtime library sends it an event that the program should process, then cycles through to execute its *statements*. During the time ACCEPT is waiting, the Clarion runtime library has control, automatically handling common events from Windows that do not need specific program action (such as screen re-draws).

The current contents of all STRING control USE variables (in the top window of each thread) automatically display on screen each time the ACCEPT loop cycles to the top. This eliminates the need to explicitly issue a DISPLAY statement to update the video display for display-only data. USE variable contents for any other control automatically display on screen for any event generated for that control, unless PROP:Auto is turned on to automatically display all USE variables each time through the ACCEPT loop.

Within the ACCEPT loop, the program determines what happened by using the following functions:

EVENT()	Returns a value indicating what happened. Symbolic constants for events are in the EQUATES.CLW file.
FIELD()	Returns the field number for the control to which the event refers, if the event is a field-specific event.
ACCEPTED()	Returns the field number for the control to which the event refers for the EVENT:Accepted event.
SELECTED()	Returns the field number for the control to which the event refers for the EVENT:Selected event.
FOCUS()	Returns the field number of the control that has input focus, no matter what event occurred.
MOUSEX()	Returns the x-coordinate of the mouse cursor.
MOUSEY()	Returns the y-coordinate of the mouse cursor.

Two events cause an implicit BREAK from the ACCEPT loop. These are the events that signal the close of a window (EVENT:CloseWindow) or close of a program (EVENT:CloseDown). The program's code need not check for these events as they are handled automatically. However, the code may check for them and execute some specific action, such as displaying a "You sure?" window or handling some housekeeping details. A CYCLE statement at that point returns to the top of the ACCEPT loop without exiting the window or program.

Similarly, there are several other events whose action can also be terminated by a CYCLE statement: EVENT:PreAlertKey, EVENT:Move, EVENT:Size, EVENT:Restore, EVENT:Maximize, and EVENT:Iconize. A CYCLE statement in response to any of these events stops the normal action and prohibits generation of the related EVENT:AlertKey, EVENT:Moved, EVENT:Sized, EVENT:Restored, EVENT:Maximized, or EVENT:Iconized.

Example:

```
CODE
OPEN(Window)
ACCEPT                                  !Event handler
  CASE FIELD()
  OF 0                                  !Handle Field-independent events
    CASE EVENT()
    OF EVENT:Move
       CYCLE                            !Do not allow user to move the window
    OF EVENT:Suspend
      CASE FOCUS()
      OF ?Field1
          !Save some stuff
     END
    OF EVENT:Resume
        !Restore the stuff
    END
  OF ?Field1                            !Handle events for Field1
    CASE EVENT()
    OF EVENT:Selected
        ! pre-edit code for field1
    OF EVENT:Accepted
        ! completion code for field1
    END
  OF ?Field2
    CASE EVENT()
    OF EVENT:Selected
        ! pre-edit code for field2
    OF EVENT:Accepted
        ! completion code for field2
    END
  END
```

See Also: EVENT, FIELD, FOCUS, ACCEPTED, SELECTED, CYCLE

ALERT (set event generation key)

ALERT([*first-keycode*] [,*last-keycode*]**)**

ALERT	Specifies keys that generate an event.
first-keycode	A numeric keycode or keycode equate label. This may be the lower limit in a range of keycodes.
last-keycode	The upper limit keycode, or keycode equate label, in a range of keycodes.

ALERT specifies a key, or an inclusive range of keys, as event generation keys. Two field-independent events, EVENT:PreAlertKey and EVENT:AlertKey, are generated when the user presses the ALERTed key. If the code executes a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the EVENT:AlertKey, preventing the library's default action on the alerted keypress for the window.

The ALERT statement with no parameters clears all ALERT keys. Any key with a keycode may be used as the parameter of an ALERT statement. ALERT generates field-independent events, since it is not associated with any particular control. When EVENT:AlertKey is generated by an ALERT key, the USE variable of the control that currently has input focus is not automatically updated (use UPDATE if this is required).

The ALERT statement alerts its keys separately from the ALRT attribute of a window or control. This means that clearing all ALERT keys has no effect on any keys alerted by ALRT attributes.

Example:

```
Screen  WINDOW,ALRT(F10Key),ALRT(F9Key) !F10 and F9 alerted
          LIST,AT(109,48,50,50),USE(?List),FROM(Que),IMM
          BUTTON('&Ok'),AT(111,108,,),USE(?Ok)
          BUTTON('&Cancel'),AT(111,130,,),USE(?Cancel)
        END
  CODE
  ALERT                             !Turn off all alerted keys
  ALERT(F1Key,F12Key)               !Alert all function keys
  ALERT(279)                        !Alert the Ctrl-Esc key
  OPEN(Screen)
  ACCEPT
    CASE EVENT()
    OF EVENT:PreAlertKey            !Pre-check alert events
      IF KEYCODE() = F4Key          !Dis-Allow F4 key
        CYCLE                       !Terminate alert processing
      END
    OF EVENT:AlertKey               !Alert processing
      CASE KEYCODE()
      OF 279                        !Check for Ctrl+Esc
        BREAK
      OF F9Key                      !Check for F9
        F9HotKeyProc                !Call hot key procedure
      OF F10Key                     !Check for F10
        F10HotKeyProc               !Call hot key procedure
      END
    END
  END
```

See Also: UPDATE

EVENT (return event number)

EVENT( )

The **EVENT** function returns a number indicating what caused ACCEPT to alert the program that something has happened that it may need to handle. There are EQUATEs listed in EQUATES.CLW for all the events the program may need to handle.

There are two types of events generated by ACCEPT: field-specific and field-independent events. Field-specific events affect a single control, while field-independent events affect the window or program. The type of event can be determined by the values returned by the ACCEPTED, SELECTED, and FIELD functions. If you need to know which field has input focus on a field-independent event, use the FOCUS function.

For field-specific events:

The FIELD function returns the field number of the control on which the event occurred. The ACCEPTED function returns the field number if the event is EVENT:Accepted. The SELECTED function returns the field number if the event is EVENT:Selected.

For field-independent events:

The FIELD, ACCEPTED, and SELECTED functions all return zero (0).

Return Data Type: SHORT

Example:

```
ACCEPT
  CASE EVENT()
  OF EVENT:Selected
    CASE SELECTED()
    OF ?Control1
      !Pre-edit code here
    OF ?Control2
      !Pre-edit code here
    END
  OF EVENT:Accepted
    CASE ACCEPTED()
    OF ?Control1
      !Post-edit code here
    OF ?Control2
      !Post-edit code here
    END
  OF EVENT:Suspend
      !Save some stuff
  OF EVENT:Resume
      !Restore the stuff
  END
END
```

POST (post user-defined event)

POST(*event* [,*control*] [,*thread*] **)**

POST	Posts an event.
event	An integer constant, variable, expression, or EQUATE containing an event number. A value in the range 400h to 0FFFh is a User-defined event.
control	An integer constant, EQUATE, variable, or expression containing the field number of the control affected by the event. If omitted, the event is field-independent.
thread	An integer constant, EQUATE, variable, or expression containing the execution thread number whose ACCEPT loop is to process the event. If omitted, the event is posted to the current thread.

POST posts an event to the currently active ACCEPT loop of the specified *thread*. This may be User-defined events, or any other event. User-defined event numbers can be defined as any integer between 400h and 0FFFh. Any *event* posted with a *control* specified is a field-specific event, while those without are field-independent events.

Example:

```
Win1     WINDOW('Tools'),AT(156,46,32,28),TOOLBOX
           BUTTON('Date'),AT(0,0,,),USE(?Button1)
           BUTTON('Time'),AT(0,14,,),USE(?Button2)
         END
  CODE
  OPEN(Win1)
  ACCEPT
    IF EVENT() = EVENT:User THEN BREAK.       !Detect user-defined event
    CASE ACCEPTED()
    OF ?Button1
      POST(EVENT:User,,UseToolsThread)
                                    !Post field-independent event to other thread
    OF ?Button2
      POST(EVENT:User)              !Post field-independent event to this thread
    END
  END
  CLOSE(Win1)
```

YIELD (allow event processing)

YIELD

YIELD temporarily gives control to Windows to allow other concurrently executing Windows applications to process events they need to handle (except those events that would post messages back to the program containing the YIELD statement ,or events that would change focus to the other application).

YIELD is used to ensure that long batch processing in a Clarion application does not completely “lock out” other applications from completing their tasks. This is known as “cooperative multi-tasking” and ensures that your Windows programs peacefully co-exist with any other Windows applications.

Within your Clarion application, YIELD only allows control to pass to EVENT:Timer events in other execution threads. This allows you to code a “background” procedure in its own execution thread using the TIMER attribute to perform some long batch processing without requiring the user to wait until the task is complete before continuing with other work in the application. This is an industry-standard Windows method of doing background processing within an application.

The example code on the next page demonstrates both approaches to performing batch processing: making the user wait for the process to complete, and processing in the background. Only the WaitForProcess procedure requires the YIELD statement, because it takes full control of the program. Background processing using EVENT:Timer does not need a YIELD statement, since the ACCEPT loop automatically performs cooperative multi-tasking with other Windows applications.

Example:

```
StartProcess  PROCEDURE
Win   WINDOW('Choose a Batch Process'),MDI
         BUTTON('Full Control'),USE(?FullControl)
         BUTTON('Background'),USE(?Background)
         BUTTON('Close'),USE(?Close)
       END
  CODE
  OPEN(Win)
  ACCEPT
    CASE FIELD()
    OF ?FullControl
      DISABLE(FIRSTFIELD(),LASTFIELD())     !Disable all buttons
      WaitForProcess                        ! and call the batch process procedure
      ENABLE(FIRSTFIELD(),LASTFIELD())      !Enable buttons when batch is complete
    OF ?Background
       X# = START(BackgroundProcess)   !Start new execution thread for the process
    OF ?Close
      BREAK
    END
  END

WaitForProcess  PROCEDURE             !Full control Batch process
  CODE
  SETCURSOR(CURSOR:Wait)              !Alert user to batch in progress
  SET(File)                           !Set up a batch process
  LOOP
    NEXT(File)
    IF ERRORCODE() THEN BREAK.
    !Perform some batch processing code
    YIELD                             !Yield to other applications and EVENT:Timer
  END
  SETCURSOR                           !Restore mmouse cursor

BackgroundProcess   PROCEDURE         !Background processing batch process
Win   WINDOW('Batch Processing...'),TIMER(1),MDI
         BUTTON('Cancel'),STD(STD:Close)
       END
  CODE
  OPEN(Win)
  SET(File)                           !Set up a batch process
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseWindow
      BREAK
    OF EVENT:Timer                    !Process records whenever the timer allows it
      LOOP 3 TIMES
        NEXT(File)
        IF ERRORCODE() THEN BREAK.
        !Perform some batch processing code
  . . .
```

Multi-Threaded Applications

Multi-Threading vs. Multi-Tasking

Multi-threading, as the term is used here, should not be confused with the ability to have the computer perform multiple tasks concurrently. Multiple execution threads within a single executing program do not necessarily imply multi-tasking, because only one thread normally executes at a time.

Windows 3.1 allows cooperative, non-preemptive, multi-tasking between separately executing applications in any mode, and preemptive multi-tasking in 386 enhanced mode. Its preemptive multi-tasking is based on "time slicing" between the applications and the amount of time each simultaneously executing application receives is governed by the end user's Windows configuration. See your Windows 3.1 documentation for an explanation of Windows' multi-tasking settings.

Windows 95 allows preemptive multi-tasking between separately executing applications. Its preemptive multi-tasking is not based on "time slicing" between the applications as Windows 3.1 was; instead, it has true preemptive multi-tasking in which the amount of time each simultaneously executing application receives is governed by each program yielding control to allow other programs to execute.

A form of cooperative, non-preemptive, multi-threading (similar to inter-application multi-tasking) can be accomplished within a single Clarion application by using the TIMER attribute. This is not based on "time slicing" between execution threads. Instead, each execution thread gains control and does not relinquish it until it executes an ASK or ACCEPT statement.

When the top window of an execution thread has the TIMER attribute, a timer event (EVENT:Timer) is periodically generated to cycle its ACCEPT loop to process the event. This occurs even if the thread does not currently have input focus. Therefore, if you want to perform this type of multi-threading, you must ensure that any lengthy execution code includes YIELD statements that occasionally execute to allow the timer events in other threads to generate and execute.

Multi-Threading and MDI

A multi-threaded application allows the user the ability to switch between multiple execution threads at runtime, as they choose. This makes the Windows Multiple Document Interface (MDI) approach to programming possible. A single Windows application may have a maximum of 64 execution threads concurrently available.

The first execution thread in any program is the main program code. This opens an APPLICATION structure as the MDI “parent” window, containing the main menu selections for the application.

The menu selections in the APPLICATION’s MENUBAR call the START function to begin each subsequent execution thread. The procedures called by START usually open an MDI “child” WINDOW, as a document window or dialog box. These windows allow the user to perform the tasks the application is designed to perform.

The last MDI “child” WINDOW opened (and not closed) in any execution thread is the “top” window in the thread and has input focus when that thread is executing. The user can switch between execution threads by using the mouse to CLICK on the top window of another execution thread. Thread switching can also be accomplished by selecting an open window from an MDI window list in the main menu, if the APPLICATION’s menu contains this standard Windows menu item.

START (return new execution thread)

START(*procedure* [,*stack*] **)**

START	Begins a new execution thread.
procedure	The label of the first PROCEDURE to call on the new execution thread. The *procedure* must have been prototyped not to receive any parameters.
stack	An integer constant or variable containing the size of the stack to allocate to the new execution thread. If omitted, the default stack is 10,000 bytes.

The **START** function begins a new execution thread, calling the *procedure* and returning the number assigned to the new thread. The returned thread number number is used by procedures and functions whose action may be performed on any execution thread (such as SETTARGET). The maximum number of simultaneously available execution threads in a single application is 64.

The first execution thread in any program is the main program code, which is always numbered one (1). Therefore, the lowest value START can return is two (2), when the first START function is executed in a program. START may return zero (0), which indicates failure to open the thread. This can occur by attempting to START a 65th thread, or by running out of memory, or by starting a thread when the system is modal.

Return Data Type: LONG

Example:

```
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
         MENUBAR
           MENU('&File'),USE(?FileMenu)
             ITEM('Selection &1...'),USE(?MenuSelection1)
             ITEM('Selection &2...'),USE(?MenuSelection2)
           END
         END
       END

SaveThread1   LONG   !Declare thread number save variable
SaveThread2   LONG   !Declare thread number save variable
  CODE
  OPEN(MainWin)                        !Open the APPLICATION
  ACCEPT                               !Handle Global events
    CASE ACCEPTED()
    OF ?MenuSelection1
      SaveThread1 = START(NewProc1)   !Start a new thread
    OF ?MenuSelection2
      SaveThread2 = START(NewProc2)   !Start a new thread
    OF ?Exit
      RETURN
    END
```

THREAD (return current execution thread)

THREAD( )

The **THREAD** function returns the currently executing thread number. The returned thread number number can be used by procedures and functions whose action may be performed on any execution thread (such as SETTARGET).

The maximum number of simultaneously available execution threads in a single application is 64. The first execution thread in any program is the main program code, which is always thread number one (1). Therefore, THREAD always returns a value in the range of one (1) to sixty-four (64).

Return Data Type: LONG

Example:

```
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
          MENUBAR
            MENU('&File'),USE(?FileMenu)
              ITEM('Selection &1...'),USE(?MenuSelection1)
              ITEM('Selection &2...'),USE(?MenuSelection2)
            END
          END
        END

SaveThread    LONG   !Declare thread number save variable
SaveThread1   LONG   !Declare thread number save variable
SaveThread2   LONG   !Declare thread number save variable
  CODE
  SaveThread = THREAD()              !Save thread number
  OPEN(MainWin)                      !Open the APPLICATION
  ACCEPT                             !Handle Global events
    CASE ACCEPTED()
    OF ?MenuSelection1
      SaveThread1 = START(NewProc1)   !Start a new thread
    OF ?MenuSelection2
      SaveThread2 = START(NewProc2)   !Start a new thread
    OF ?Exit
      RETURN
    END
  END
```

Window Procedures

CHANGE (change control field value)

CHANGE(*control,value***)**

CHANGE	Changes the *value* displayed in a *control* in an APPLICATION or WINDOW structure.
control	Field number or field equate label of a window control field.
value	A constant or variable containing the *control's* new value.

The **CHANGE** statement changes the *value* displayed in a *control* in an APPLICATION or WINDOW structure. CHANGE updates the *control's* USE variable with the *value*, and then displays that new *value* in the control field.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  ACCEPT
    CASE EVENT()
    OF EVENT:Selected
      CASE SELECTED()
      OF ?Ctl:Code
        CHANGE(?Ctl:Code,4)             !Change Ctl:Code to 4 and display it
      OF ?Ctl:Name
        CHANGE(?Ctl:Name,'ABC Company')
                                        !Change Ctl:Name to ABC Company and display
      END
    OF EVENT:Accepted
      CASE ACCEPTED()
      OF ?OkButton
        BREAK
      OF ?CanxButton
        CLEAR(Ctl:Record)
        BREAK
      END
    END
```

CLOSE (close window)

CLOSE(*label*)

CLOSE	Closes the active APPLICATION or WINDOW structure.
label	The label of an APPLICATION or WINDOW structure.

CLOSE terminates processing on the active APPLICATION or WINDOW structure. Memory used by the active window is released when it is closed and the underlying screen is automatically re-drawn.

When a window is closed, if it is not the top-most window on its execution thread, all windows opened subsequent to the window being closed are automatically closed first. This occurs in the reverse order from which they were opened.

An APPLICATION or WINDOW that is declared local to (within) a PROCEDURE or FUNCTION is automatically closed when the program RETURNs from the procedure.

Example:

```
CLOSE(MenuScr)          !Close the  menu screen
CLOSE(CustEntry)        !Close customer data entry screen
```

CREATE (create new control)

CREATE(*control* ,*type* [,*parent*] **)**

CREATE	Creates a new control.
control	A field number or field equate label for the control to create.
type	An integer constant, expression, EQUATE, or variable that specifies the type of control to create.
parent	A field number or field equate label. This specifies an OPTION, GROUP, or MENU to contain the new *control*.

CREATE dynamically creates a new control in the currently active APPLICATION or WINDOW. When first created, the new *control* is initially hidden, so its properties can be set using the runtime property assignment syntax, SETPOSITION, and SETFONT. It appears on screen only by issuing an UNHIDE statement for the *control*. To place the new control on the toolbar, add CREATE:TOOLBAR to the equate for the new control's *type*.

EQUATE statements for the *type* parameter are contained in the EQUATES.CLW file. The following list is a comprehensive sample of these (see EQUATES.CLW for the complete list):

```
CREATE:sstring      STRING(picture),USE(variable)
CREATE:string       STRING(constant)
CREATE:image        IMAGE()
CREATE:region       REGION()
CREATE:line         LINE()
CREATE:box          BOX()
CREATE:ellipse      ELLIPSE()
CREATE:entry        ENTRY()
CREATE:button       BUTTON()
CREATE:prompt       PROMPT()
CREATE:option       OPTION()
CREATE:radio        RADIO()
CREATE:check        CHECK()
CREATE:group        GROUP()
CREATE:list         LIST()
CREATE:combo        COMBO()
CREATE:spin         SPIN()
CREATE:text         TEXT()
CREATE:custom       CUSTOM()
CREATE:droplist     LIST(),DROP()
CREATE:dropcombo    COMBO(),DROP()
CREATE:menu         MENU()
CREATE:item         ITEM()
```

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END

X       SHORT
Y       SHORT
Width   SHORT
Height  SHORT

Code4Entry  STRING(10)
?Code4Entry EQUATE(100)

  CODE
  OPEN(Screen)
  ACCEPT
    CASE ACCEPTED()
    OF ?Ctl:Code
      IF Ctl:Code = 4
        CREATE(?Code4Entry,CREATE:entry)           !Create the control
        ?Code4Entry{PROP:use} = 'Code4Entry'       !Set USE variable
        ?Code4Entry{PROP:text} = '@s10'            !Set entry picture
        GETPOSITION(?Ctl:Code,X,Y,Width,Height)
        ?Code4Entry{PROP:at,1} = X + Width + 40   !Set x position
        ?Code4Entry{PROP:at,2} = Y                 !Set y position
        UNHIDE(?Code4Entry)                        !Display the new control
      END
    OF ?OkButton
      BREAK
    OF ?CanxButton
      CLEAR(Ctl:Record)
      BREAK
    END
  END
  CLOSE(Screen)
  RETURN
```

See Also: DESTROY

DESTROY (remove a control)

DESTROY(*first control* [,*last control*] **)**

DESTROY	Removes window controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **DESTROY** statement removes a control, or range of controls, from an APPLICATION or WINDOW structure. When removed, the control's resources are returned to the operating system.

DESTROYing a GROUP, OPTION, MENU, TAB, or SHEET control also destroys all controls contained within it.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  DESTROY(?Ctl:Code)                !Remove a control
  DESTROY(?Ctl:Code,?Ctl:Name)      !Remove range of controls
  DESTROY(2)                        !Remove the second control
```

See Also: CREATE

DISABLE (dim a control)

DISABLE(*first control* [,*last control*] **)**

DISABLE	Dims controls on the window.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **DISABLE** statement disables a control or a range of controls on an APPLICATION or WINDOW structure. When disabled, the control appears dimmed on screen.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  DISABLE(?Ctl:Code)                !Disable a control
  DISABLE(?Ctl:Code,?Ctl:Name)      !Disable range of controls
  DISABLE(2)                        !Disable the second control
```

See Also: ENABLE, HIDE, UNHIDE

DISPLAY (write USE variables to screen)

DISPLAY([*first control*] [,*last control*] **)**

DISPLAY	Writes the contents of USE variables to their associated controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

DISPLAY writes the contents of the USE variables to their associated controls on the active window. DISPLAY with no parameters writes the USE variables for all controls on the screen. Using *first control* alone, as the parameter of DISPLAY, writes a specific USE variable to the screen. Both *first control* and *last control* parameters are used to display the USE variables for an inclusive range of controls on the screen.

The current contents of the USE variables of all controls are automatically displayed on screen each time the ACCEPT loop cycles. This eliminates the need to explicitly issue a DISPLAY statement to update the video display. Of course, if your application performs some operation that takes a long time and you want to indicate to the user that something is happening without cycling back to the top of the ACCEPT loop, you should DISPLAY some variable that you have updated.

Example:

```
DISPLAY                  !Display all controls on the screen
DISPLAY(2)               !Display control number 2
DISPLAY(3,7)             !Display controls 3 through 7
DISPLAY(?MenuControl)    !Display the menu control
DISPLAY(?TextBlock,?Ok)  !Display range of controls
```

See Also: Field Equate Labels, UPDATE, ERASE

ENABLE (re-activate dimmed control)

ENABLE(*first control* [,*last control*] **)**

ENABLE	Reactivates disabled controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **ENABLE** statement reactivates a control, or range of controls, that were dimmed by the DISABLE statement, or were declared with the DISABLE attribute. Once reactivated, the control is again available to the operator for selection.

Example:

```
CODE
OPEN(Screen)
DISABLE(?Control2)                !Control2 is deactivated
IF Ctl:Password = 'Supervisor'
  ENABLE(?Control2)               !Re-activate Control2
END
```

See Also: DISABLE, HIDE, UNHIDE

ERASE (clear screen control and USE variables)

ERASE([*first control*] [,*last control*] **)**

ERASE	Blanks controls and clears their USE variables.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **ERASE** statement erases the data from controls in the window and clears their corresponding USE variables. ERASE with no parameters erases all controls in the window. Using *first control* alone, as the parameter of ERASE, clears a specific USE variable and its associated control. Both *first control* and *last control* parameters are used to clear the USE variables and associated controls for an inclusive range of controls in the window.

Example:

```
ERASE(?)                !Erase the currently selected control
ERASE                   !Erase all controls on the screen
ERASE(3,7)              !Erase controls 3 through 7
ERASE(?Name,?Zip)       !Erase controls from name through zip
ERASE(?City,?City+2)    !Erase City and 2 controls following City
```

See Also: Field equate labels

GETFONT (get font information)

GETFONT(*control* ,*typeface* , *size* ,*color* ,*style***)**

GETFONT	Gets display font information.
control	A field number or field equate label for the control from which to get the information. If *control* is zero (0), it specifies the WINDOW.
typeface	A string variable to receive the name of the font.
size	An integer variable to receive the size (in points) of the font.
color	A LONG integer variable to receive the red, green, and blue values for the color of the font in the low-order three bytes. If the value is negative, the *color* represents a system color.
style	An integer variable to receive the strike weight and style of the font.

GETFONT gets the display font information for the *control*. If the *control* parameter is zero (0), GETFONT gets the default display font for the window.

Example:

```
TypeFace  STRING(20)
Size      BYTE
Color     LONG
Style     LONG

  CODE
  OPEN(Screen)
  GETFONT(0,TypeFace,Size,Color,Style)         !Get font info for the window
```

See Also: SETFONT

GETPOSITION (get control position)

GETPOSITION(*control* ,*x* , *y* ,*width* ,*height***)**

GETPOSITION	Gets the position and size of an APPLICATION, WINDOW, or control.
control	A field number or field equate label for the control from which to get the information. If *control* is zero (0), it specifies the window.
x	An integer variable to receive the horizontal position of the top left corner.
y	An integer variable to receive the vertical position of the top left corner.
width	An integer variable to receive the width.
height	An integer variable to receive the height.

GETPOSITION gets the position and size of an APPLICATION, WINDOW, or control. The position and size values are dependent upon the presence or absence of the SCROLL attribute on the *control*. If SCROLL is present, the values are relative to the virtual window. If SCROLL is not present, the values are relative to the top left corner of the currently visible portion of the window. This means the values returned always match those specified in the AT attribute or most recent SETPOSITION.

The values in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END

X       SHORT
Y       SHORT
Width   SHORT
Height  SHORT
  CODE
  OPEN(Screen)
  GETPOSITION(?Ctl:Code,X,Y,Width,Height)
```

See Also: SETPOSITION

HELP (help window access)

HELP([*helpfile*] [,*window-id*]**)**

HELP	Opens a help file and activates a help window.
helpfile	A string constant or the label of a STRING variable that has the DOS directory file specification for the help file. If the file specification does not contain a complete path and filename, the help file is assumed to be in the current directory. If the file extension is omitted, ".HLP" is assumed. If the *helpfile* parameter is omitted, a comma is required to hold its position.
window-id	A string constant or the label of a STRING variable that contains the key used to access the help system. This may be either a help keyword or a "context string."

The **HELP** statement opens a designated *helpfile*, and activates the window named by the *window-id*. While an ASK or ACCEPT is controlling program execution, the active help window is displayed when the operator presses F1 (the "Help" key).

If the *window-id* parameter is omitted, the *helpfile* is nominated but not opened. If the *helpfile* parameter is omitted, the current help file is opened, and the window identified by *window-id* is activated. If both parameters are omitted, the current *helpfile* is opened at the current topic.

The *window-ID* may contain a Help keyword. This is a keyword that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *window-ID*, followed by a unique identifier associated with exactly one help topic. If the tilde is missing, the *window-ID* is assumed to be a help keyword. When the user presses F1, the help file is opened at the specific topic associated with that "context string."

Help windows are also activated by the HLP attribute of an APPLICATION, WINDOW, or control.

Example:

```
HELP('C:\HLPDIR\LEDGER.HLP')    !Open the gen ledger help file
HELP(,'~CustUpd')               !Activate customer update help window
HELP                            !Display the help window
```

See Also: ASK, ACCEPT, HLP

HIDE (blank a control)

HIDE(*first control* [,*last control*] **)**

HIDE	Hides window controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **HIDE** statement hides a control, or range of controls, on an APPLICATION or WINDOW structure. When hidden, the control does not appear on screen.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  HIDE(?Ctl:Code)               !Hide a control
  HIDE(?Ctl:Code,?Ctl:Name)     !Hide range of controls
  HIDE(2)                       !Hide the second control
```

See Also: UNHIDE, ENABLE, DISABLE

OPEN (open window for processing)

OPEN(*label***)**

OPEN	Opens a window.
label	The label of an APPLICATION or WINDOW structure.

OPEN activates an APPLICATION or WINDOW for processing. However, nothing is displayed until a DISPLAY statement or the ACCEPT loop is encountered. This allows an opportunity to execute pre-display code to customize the display.

Example:

```
OPEN(MenuScr)         !Open the menu screen
OPEN(CustEntry)       !Open customer data entry screen
```

SELECT (select next control to process)

SELECT([*control*] [,*position*] [,*endposition*] **)**

SELECT	Sets the next control to receive input focus.
control	A field number or field equate label of the next control to process. If omitted, the SELECT statement initiates AcceptAll mode.
position	Specifies a position within the *control* to place the cursor. For an ENTRY or TEXT, SPIN, or COMBO control this is a character position, or a beginning character position for a marked block. For an OPTION structure, this is the selection number within the OPTION. For a LIST control, this is the QUEUE entry number.
endposition	Specifies an ending character position within an ENTRY, TEXT, SPIN, or COMBO *control*. The character position specified by *position* and *endposition* are marked as a block, available for cut and paste operations.

SELECT overrides the normal TAB key sequence control selection order of an APPLICATION or WINDOW. Its action affects the next ACCEPT statement that executes. The *control* parameter determines which control the ACCEPT loop will process next. If *control* specifies a control which cannot rceive focus because a DISABLE or HIDE statment has been issued, focus goes to the next control following it in the window's source code that can receive focus. If *control* specifies a control on a TAB which does not have focus, the TAB is brought to the front before the control receives focus.

SELECT with *position* and *endposition* parameters specifies a marked block in the *control* which is available for cut and paste operations.

SELECT with no parameters initiates AcceptAll mode (also called non-stop mode). This is a field edit mode in which each control in the window is processed in TAB key sequence by generating EVENT:Accepted for each. This allows data entry validation code to execute for all controls, including those that the user has not touched.

AcceptAll mode terminates when any of the following conditions is met:

- A SELECT(?) statement selects the same control for the user to edit. This code usually indicates the value it contains is invalid and the user must re-enter data.
- The Window{PROP:AcceptAll} property is set to zero (0). This property contains one (1) when AcceptAll mode is active. Assigning values to this property can also be used to initiate and terminate AcceptAll mode.

- A control with the REQ attribute is blank or zero. AcceptAll mode terminates with the control highlighted for user entry, without processing any more fields in the TAB key sequence.

When all controls have been processed, EVENT:Completed is posted to the window.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  SELECT(?Ctl:Code)                         !Start with Ctl:Code
  ACCEPT
    CASE ACCEPTED()
    OF ?Ctl:Code
      IF Ctl:Code > 150                     !If data entered is invalid
        BEEP                                ! alert the user and
        SELECT(?)                           ! make them re-enter the data
      END
    OF ?Ctl:Name
      SELECT(?Ctl:Name,1,5)                 !Mark first five characters as a block
    OF ?OkButton
      SELECT                                !Initiate AcceptAll mode
    END
    IF EVENT() = EVENT:Completed THEN BREAK.
                                            !AcceptAll mode terminated
  END
```

See Also: ACCEPT

SET3DLOOK (set 3D window look)

SET3DLOOK([*switch*] **)**

SET3DLOOK Toggles three-dimensional look and feel.

switch An integer constant switching the 3D look off (0) and on (1). If omitted, the default is one (1).

The **SET3DLOOK** procedure sets up the program to display a three-dimensional look and feel. The default program setting is 3D enabled. On a WINDOW, the GRAY attribute causes the controls to display with a three-dimensional appearance. Controls in the TOOLBAR are always displayed with the three-dimensional look, unless disabled by SET3DLOOK. When three-dimensional look is disabled by SET3DLOOK, the GRAY attribute has no effect.

SET3DLOOK(0) turns off the three-dimensional look and feel. SET3DLOOK(1) turns on the three-dimensional look and feel. Values other than zero or one are reserved for future use.

Example:

```
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
         MENUBAR
           MENU('&File'),USE(?FileMenu)
             ITEM('&Open...'),USE(?OpenFile)
             ITEM('&Close'),USE(?CloseFile),DISABLE
             ITEM('Turn off 3D Look'),USE(?Toggle3D),CHECK
             ITEM('E&xit'),USE(?MainExit)
           END
         END
       END
  CODE
  OPEN(MainWin)
  ACCEPT
    CASE ACCEPTED()
    OF ?Toggle3D
      IF MainWin$?Toggle3D{PROP:text} = 'Turn off 3D Look'      !If on
        SET3DLOOK(0)                                            !Turn off
        MainWin$?Toggle3D{PROP:text} = 'Turn on 3D Look'        ! and change text
      ELSE                                                      !Else
        SET3DLOOK(1)                                            !Turn on
        MainWin$?Toggle3D{PROP:text} = 'Turn off 3D Look'       ! and change text
      END
    OF ?OpenFile
      START(OpenFileProc)
    OF ?MainExit
      BREAK
    END
  END
  CLOSE(MainWin)
```

SETCURSOR (set temporary mouse cursor)

SETCURSOR([*cursor*] **)**

SETCURSOR	Specifies a temporary mouse cursor to display.
cursor	An EQUATE naming a Windows-standard mouse cursor. If omitted, turns off the temporary cursor.

The **SETCURSOR** statement specifies a temporary mouse *cursor* to display until a SETCURSOR statement without a *cursor* parameter turns it off. This cursor overrides all CURSOR attributes. When SETCURSOR without a *cursor* parameter is encountered, all CURSOR attributes once again take effect. SETCURSOR is generally used to display the hourglass while your program is doing some "behind the scenes" work that the user should not break into.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital "I" like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow

Example:

```
MainWin APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS |
   ,HVSCROLL,RESIZE
         MENUBAR
           ITEM('Batch Update'),USE(?Batch)
         END
       END
  CODE
  OPEN(MainWin)
  ACCEPT
    CASE ACCEPTED()
    OF ?Batch
      SETCURSOR(CURSOR:Wait)    !Turn on hourglass mouse cursor
      BatchUpdate               ! and call the batch update procedure
    END
  END
```

SETFONT (specify font)

SETFONT(*control* ,*typeface* , *size* ,*color* ,*style***)**

SETFONT	Dynamically sets the display font for a control.
control	A field number or field equate label for the control to affect. If *control* is zero (0), it specifies the WINDOW.
typeface	A string constant or variable containing the name of the font. If omitted, the system font is used.
size	An integer constant or variable containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant or variable containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant, constant expression, EQUATE, or variable specifying the strike weight and style of the font. If omitted, the weight is normal.

SETFONT dynamically specifies the display font for the *control*, overriding any FONT attribute previously specified. If the *control* parameter is zero (0), SETFONT specifies the default display font for the window.

SETFONT allows you to specify all parameters of a font change at once, instead of one at a time as runtime property assignment allows. This has the advantage of implementing all changes at once, whereas runtime property assignment would change each individually, displaying each separate change as it occurs.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may also add values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
SETFONT(0,'Arial',14,,FONT:thin+FONT:Italic)      !14 pt. Arial black thin italic
```

See Also: GETFONT

SETPOSITION (specify new control position)

SETPOSITION(*control* ,*x* , *y* ,*width* ,*height***)**

SETPOSITION	Dynamically specifies the position and size of an APPLICATION, WINDOW, or control.
control	A field number or field equate label for the control to affect. If *control* is zero (0), it specifies the window.
x	An integer constant, expression, or variable that specifies the horizontal position of the top left corner. If omitted, the *x* position is not changed.
y	An integer constant, expression, or variable that specifies the vertical position of the top left corner. If omitted, the *y* position is not changed.
width	An integer constant, expression, or variable that specifies the width. If omitted, the *width* is not changed.
height	An integer constant, expression, or variable that specifies the height. If omitted, the *height* is not changed.

SETPOSITION dynamically specifies the position and size of an APPLICATION, WINDOW, or control. If any parameter is omitted, the value is not changed.

The values contained in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Using SETPOSITION produces a "smoother" control appearance change than using property expressions to change the AT attribute's parameter values. This is because SETPOSITION changes all four parameters at once. Property expressions must change one parameter at a time. Since each individual parameter change would be immediately visible on screen, this would cause the control to appear to "jump."

Example:

```
CREATE(?Code4Entry,CREATE:entry,?Ctl:Code)  !Create a control
?Code4Entry{PROP:use} = 'Code4Entry'        !Set USE variable
?Code4Entry{PROP:text} = '@s10'             !Set entry picture
GETPOSITION(?Ctl:Code,X,Y,Width,Height)     !Get Ctl:Code position
SETPOSITION(?Code4Entry,X+Width+40,Y)       !Set x 40 past Ctl:Code
UNHIDE(?Code4Entry)                         !Display the new control
```

See Also: GETPOSITION

SETTARGET (set current window or report)

SETTARGET([*target*] [,*thread*] **)**

SETTARGET	Sets the current window (or report) for drawing graphics and other window-interaction statements.
target	The label of an APPLICATION, WINDOW or REPORT structure. If omitted, the last window opened and not yet closed in the specified *thread* is used.
thread	The number of the execution thread in which the *target* structure is contained in the topmost procedure or function. If omitted, the current execution thread is used.

The **SETTARGET** procedure makes the *target* the structure which is current for drawing with the graphics primitives functions. SETTARGET also sets the *target* for runtime property assignment, and the CREATE, SETPOSITION, GETPOSITION, SETFONT, GETFONT, DISABLE, HIDE, CONTENTS, DISPLAY, ERASE, and UPDATE statements. Using these statements with SETTARGET allows you to manipulate the window display in the topmost window of any execution thread.

This *target* will receive any graphics drawn with the graphics procedures and functions described in the Graphics Commands chapter. This allows you to draw graphics to the topmost window, or report, in any execution thread.

SETTARGET sets the ”built-in” variable, TARGET (also set when a window is opened), which may be used in any statement which requires the label of the current window or report. A REPORT data structure is never the default *target*. Therefore, SETTARGET must be used before using the graphics primitives functions to draw graphics on a REPORT.

SETTARGET does not change procedures, and it does not change which ACCEPT loop receives the events generated by Windows. SETTARGET without any parameters resets to the procedure and execution thread with the currently active ACCEPT loop.

Example:

```
Report REPORT
         !Report structure controls
       END
  CODE
  OPEN(Report)
  SETTARGET(Report)               !Make the report the current target
  TARGET{PROP:Landscape} = 1      ! and turn on landscape mode
```

UNHIDE (show hidden control)

UNHIDE(*first control* [,*last control*] **)**

UNHIDE	Displays previously hidden controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **UNHIDE** statement reactivates a control or range of controls, that were hidden by the HIDE statement. Once un-hidden, the control is again visible on screen.

Example:

```
CODE
OPEN(Screen)
HIDE(?Control2)                    !Control2 is hidden
IF Ctl:Password = 'Supervisor'
  UNHIDE(?Control2)                !Unhide Control2
END
```

See Also: HIDE, ENABLE, DISABLE

UPDATE (write from screen to USE variables)

UPDATE([*first control*] [,*last control*] **)**

UPDATE	Writes the contents of a control to its USE variable.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

UPDATE writes the contents of a screen control to its USE variable. This takes the value displayed on screen and places it in the variable specified by the control's USE attribute.

USE variables are updated automatically by ACCEPT as each control is accepted. However, certain events (such as an ALERTed key press) do not automatically update USE variables. This is the purpose of the UPDATE statement.

UPDATE	Updates all controls on the screen.
UPDATE(*first control*)	Updates a specific USE variable from its associated screen control.
UPDATE(*first control*,*last control*)	Updates the USE variables of an inclusive range of screen controls.

Example:

```
UPDATE(?)                !Update the currently selected control
UPDATE                   !Update all controls on the screen
UPDATE(?Address)         !Update the address control
UPDATE(3,7)              !Update controls 3 through 7
UPDATE(?Name,?Zip)       !Update controls from name through zip
UPDATE(?City,?City+2)    !Update city and 2 controls following
```

See Also: Field equate Labels

Window Functions

ACCEPTED (return control just completed)

ACCEPTED( )

The **ACCEPTED** function returns the field number of the control on which an EVENT:Accepted event occurred. ACCEPTED returns zero (0) for all other events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels—the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SHORT

Example:

```
CASE ACCEPTED()         !Process post-edit code
OF ?Cus:Company
  !Edit field value
OF ?Cus:CustType
  !Edit field value
END
```

See Also: ACCEPT, EVENT

CHOICE (return relative item position)

CHOICE([*control*] **)**

CHOICE	Returns a user selection number.
control	A field equate label of a LIST, COMBO, or OPTION control.

The **CHOICE** function returns the sequence number of a selected item in an OPTION structure, LIST box, or COMBO control. With no parameter, CHOICE returns the sequence number of the selected item in the last control (LIST, OPTION, or COMBO) that generated a Field-specific event to cycle the ACCEPT loop. CHOICE(*control*) returns the current selection number of any LIST, OPTION, or COMBO in the currently active window.

CHOICE returns the sequence number of the selected RADIO control within an OPTION structure. The sequence number is determined by relative position within the OPTION. The first control listed in the OPTION structure's code is relative position 1, the second is 2, etc.

CHOICE returns the memory QUEUE entry number of the selected item when a LIST or COMBO box is completed.

Return Data Type: LONG

Example:

```
CODE
ACCEPT
  EXECUTE CHOICE()    !Perform menu option
    AddRec            ! procedure to add record
    PutRec            ! procedure to change record
    DelRec            ! procedure to delete record
    RETURN            ! return to caller
  END
END
```

CONTENTS (return contents of USE variable)

CONTENTS(*control*)

CONTENTS Returns the value in the USE variable of a control.

control A field number or field equate label.

The **CONTENTS** function returns a string containing the value in the USE variable of an ENTRY, OPTION RADIO, or TEXT control.

A USE variable may be longer than its associated control field display picture OR may contain fewer characters than its total capacity. The CONTENTS function always returns the full length of the USE variable.

Return Data Type: STRING

Example:

```
IF CONTENTS(?LastName) = '' AND CONTENTS(?FirstName) = ''
                                                !If first and last name are blank
  MessageField = 'Must Enter a First or Last Name'     ! display error message
END
```

FIELD (return control with focus)

FIELD( )

The **FIELD** function returns the field number of the control which has focus at the time of any field-specific event. This includes both the EVENT:Selected and EVENT:Accepted events. FIELD returns zero (0) for field-independent events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels—the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: LONG

Example:

```
Screen  WINDOW
          ENTRY(@N4),USE(Control1)
          ENTRY(@N4),USE(Control2)
          ENTRY(@N4),USE(Control3)
          ENTRY(@N4),USE(Control4)
        END
  CODE
  ACCEPT
    IF NOT ACCEPTED() THEN CYCLE.
    CASE FIELD()              !Control edit control
    OF ?Control1              ! Field number 1
      IF Control1 = 0         ! if no entry
        BEEP                  !   sound alarm
        SELECT(?)             !   stay on control
      END
    OF ?Control2              ! Field number 2
      IF Control2 > 4         ! if status is more than 4
        Scr:Message = 'Control must be less than 4'
        ERASE(?)              !   clear control
        SELECT(?)             !   edit the control again
      ELSE                    ! value is valid
        CLEAR(Scr:Message)    !   clear message
      END
    OF ?Control4              ! Field number 4
      BREAK                   ! exit processing loop
  . .                         ! end case, end loop
```

See Also: ACCEPTED, SELECTED, FOCUS, EVENT

FIRSTFIELD (return first window control)

FIRSTFIELD( )

The **FIRSTFIELD** function returns the lowest field number in the currently active window.

Return Data Type: LONG

Example:

```
DISABLE(FIRSTFIELD(),LASTFIELD())   !Dim all control fields
```

FOCUS (return control with focus)

FOCUS( )

The **FOCUS** function returns the field number of the control which has input focus at any time any event occurs.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels—the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: LONG

Example:

```
Screen  WINDOW
          ENTRY(@N4),USE(Control1)
          ENTRY(@N4),USE(Control2)
          ENTRY(@N4),USE(Control3)
          ENTRY(@N4),USE(Control4)
        END
  CODE
  ACCEPT
    CASE EVENT()
    OF EVENT:LoseFocus
    OROF EVENT:CloseWindow
      CASE FOCUS()            !Control edit control
      OF ?Control1            ! Field number 1
        UPDATE(?Control1)
      OF ?Control2            ! Field number 2
        UPDATE(?Control2)
      OF ?Control4            ! Field number 4
        UPDATE(?Control4)
      END
    END
  END
```

See Also: ACCEPTED, SELECTED, FIELD, EVENT

INCOMPLETE (return empty REQ control)

INCOMPLETE( )

The **INCOMPLETE** function returns the field number of the first control with the REQ attribute in the currently active window that has been left zero or blank, and gives input focus to that control. If all REQ controls in the window contain data, INCOMPLETE returns zero (0) and leaves input focus on the control that already had it.

The INCOMPLETE function duplicates the action performed by the REQ attribute on a BUTTON control.

Return Data Type: LONG

Example:

```
CODE
OPEN(Screen)
ACCEPT
  CASE ACCEPTED()
  OF ?OkBUtton
    IF INCOMPLETE()             !Any REQ fields empty?
      SELECT(INCOMPLETE())      ! if so, go to it
      CYCLE
    ELSE
      BREAK                     !If not, go on
    END
  END
END
```

LASTFIELD (return last window control)

LASTFIELD( )

The **LASTFIELD** function returns the highest field number in the currently active window.

Return Data Type: LONG

Example:

```
DISABLE(FIRSTFIELD(),LASTFIELD())     !Dim all control fields
```

MESSAGE (return message box response)

MESSAGE(*text* [,*caption*] [,*icon*] [,*buttons*] [,*default*] [,*style*] **)**

MESSAGE	Displays a message dialog box and returns the button the user pressed.
text	A string constant or variable containing the text to display in the message box.
caption	The dialog box title. If omitted, the dialog has no title.
icon	A string constant, variable, or EQUATE for a Windows standard icon. If omitted, no icon is displayed on the dialog box.
buttons	An integer constant, variable, EQUATE, or expression which indicates which Windows standard buttons to place on the dialog box. This may indicate multiple buttons. If omitted, the dialog displays an Ok button.
default	An integer constant, variable, EQUATE, or expression which indicates the default button on the dialog box. If omitted, the first button is the default.
style	An integer constant or variable which specifies the window is Application Modal (0) or System Modal (1). If omitted, the window is Application Modal.

The **MESSAGE** function displays a Windows-standard message box, typically requiring only a Yes or No response, or no specific response at all. The function returns the number of the button the user presses to exit the dialog box.

The EQUATES.CLW file contains symbolic constants for the *icon*, *buttons*, and *default* parameters. The *style* parameter determines whether the message window is Application Modal or System Modal. An Application Modal window must be closed before the user is allowed to do anything else in the application, but does not prevent the user from switching to another Windows application. A System Modal window must be closed before the user is allowed to do anything else in Windows.

Return Data Type: USHORT

Example:

```
CASE  MESSAGE('Quit?','Editor',ICON:Question,BUTTON:Yes+BUTTON:No,BUTTON:No,1)
                        !A ? icon with Yes and No buttons, the default button is No
                        ! the window is System Modal
OF BUTTON:No
OF BUTTON:Yes
  MESSAGE('Goodbye')    !A message with only an Ok button.
  RETURN
END
```

MOUSEX (return mouse horizontal position)

MOUSEX( )

The **MOUSEX** function returns a numeric value corresponding to the current horizontal position of the mouse cursor at the time of the event. The position is relative to the origin of that window.

The return value is in dialog units.

Return Data Type: SHORT

Example:

```
SaveMouseX = MOUSEX()     !Save mouse position
```

MOUSEY (return mouse vertical position)

MOUSEY( )

The **MOUSEY** function returns a numeric value corresponding to the current vertical position of the mouse cursor at the time of the event. The position is relative to the origin of that window.

The return value is in dialog units.

Return Data Type: SHORT

Example:

```
SaveMouseY = MOUSEY()       !Save mouse position
```

POPUP (return popup menu selection)

POPUP(*selections* [, *x*] [, *y*] **)**

POPUP	Returns an integer indicating the user's choice from the menu.
selections	A string constant, variable, or expression containing the text for the menu choices.
x	An integer constant, variable, or expression that specifies the horizontal position of the top left corner. If omitted, the menu appear at the current cursor position.
y	An integer constant, variable, or expression that specifies the vertical position of the top left corner. If omitted, the menu appear at the current cursor position.

The **POPUP** function returns an integer indicating the user's choice from the popup menu that appears when the function is invoked. If the user CLICKS outside the menu or presses ESC (indicating no choice), POPUP returns zero.

Within the *selections* string, each choice in the popup menu must be delimited by a vertical bar (|) character. A set of vertical bars containing only a hyphen (|-|) defines a separator between groups of menu choices. A menu choice immediately preceded by a tilde (~) is disabled (it appears dimmed out in the popup menu). A menu choice immediately preceded by a plus sign (+) appears with a check mark to its left in the popup menu. A menu choice immediately followed by a set of choices contained within curly braces (|SubMenu{{SubChoice 1|SubChoice 2}|) defines a sub-menu within the popup menu (the two beginning curly braces are required by the compiler to differentiate your sub-menu from a string repeat count).

Each menu selection is numbered in ascending sequence according to its position within the *selections* string, beginning with one (1). Separators and selections that call a sub-menu are not included in the numbering sequence (which makes an EXECUTE structure the most efficient code structure to use with this function). When the user CLICKS or presses ENTER on a choice, the function terminates, returning the position number of the selected menu item.

Return Data Type: SHORT

Example:

```
PopupString = 'First|+Second|Sub menu{{One|Two}|-|Third|~Disabled'
ToggleChecked = 1
ACCEPT
  CASE EVENT()
  OF EVENT:AlertKey
    IF KEYCODE() = MouseRight
      EXECUTE POPUP(PopupString)
        FirstProc                !Call proc for selection 1
        BEGIN                    !Code to execute for toggle selection 2
          IF ToggleChecked = 1   !Check toggle state
            SecondProc(Off)      !Call proc to turn off something
            PopupString = 'First|Second|Sub menu{{One|Two}|-|Third|~Disabled'
                                 !Reset string so the check mark does not appear
            ToggleChecked = 0    !Set toggle flag
          ELSE
            SecondProc(On)       !Call proc to turn off something
            PopupString = 'First|+Second|Sub menu{{One|Two}|-|Third|~Disabled'
                                 !Reset string so the check mark does appear
            ToggleChecked = 1    !Set toggle flag
          END
        END                      !End Code to execute for toggle selection 2
        OneProc                  !Call proc for selection 3
        TwoProc                  !Call proc for selection 4
        ThirdProc                !Call proc for selection 5
        DisabledProc    !Selection 6 is dimmed so it cannot execute this procedure
      END
    END
  END
END
```

SELECTED (return control that has received focus)

SELECTED( )

The **SELECTED** function returns the field number of the control receiving input focus when an EVENT:Selected event occurs. SELECTED returns zero (0) for all other events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels—the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SHORT

Example:

```
CASE SELECTED()           !Process pre-edit code
OF ?Cus:Company
  !Pre-load field value
OF ?Cus:CustType
  !Pre-load field value
END
```

See Also: ACCEPT, SELECT

Keyboard Procedures

ALIAS (set alternate keycode)

ALIAS([*keycode*,[*new keycode*]] **)**

ALIAS	Changes the keycode generated when the original key is pressed.
keycode	A numeric keycode or keycode EQUATE. If both parameters are omitted, all ALIASed keys are reset to their original values.
new keycode	A numeric keycode or keycode EQUATE. If omitted, the *keycode* is reset to its original value.

ALIAS changes the *keycode* to generate the *new keycode* when the user presses the original key. ALIAS does not affect keypresses generated by PRESSKEY. The effect of ALIAS is global, throughout all execution threads, no matter where the ALIAS statement executes. Therefore, to only change the *keycode* locally, you must reset ALIASed keys when the window loses focus.

Keycode values 0800h through 0FFFFh are unassigned and may be used as a *new keycode*. The practical effect of this is to disable the original key if your program does not test for the *new keycode.*

Example:

```
ALIAS(EnterKey,TabKey)    !Allow user to press enter instead of tab
ALIAS(F3Key,F1Key)        !Move help to F3
ALIAS                     !Clear all aliased keys
```

ASK (get one keystroke)

ASK

ASK reads a single keystroke from the keyboard buffer. Program execution stops to wait for a keystroke. If there is already a keystroke in the keyboard buffer, ASK gets one keystroke without waiting.

The ASK statement also allows any TIMER attribute events to generate and cycle their own ACCEPT loop. This means any batch processing code can allow other threads to execute their TIMER attribute tasks during the batch process.

Example:

```
ASK                             !Wait for a keystroke
LOOP WHILE KEYBOARD()           !Empty the keyboard buffer
  ASK                           ! without processing keystrokes
END
```

PRESS (put characters in the buffer)

PRESS(*string***)**

PRESS Places characters in the keyboard input buffer.

string A string constant, variable, or expression.

PRESS places characters in the Windows keyboard input buffer. The entire *string* is placed in the buffer. Once placed in the keyboard buffer, the *string* is processed just as if the user had typed in the data.

Example:

```
IF LocalRequest = AddRecord             !On the way into a memo on adding a record
  TempString = FORMAT(TODAY(),@D1) & ' ' & FORMAT(CLOCK(),@T4)
  PRESS(TempString)                    !Pre-load first line of memo with date and time
END
```

PRESSKEY (put a keystroke in the buffer)

PRESSKEY(*keycode***)**

PRESSKEY Places one keystroke in the keyboard input buffer.

keycode An integer constant or keycode EQUATE label.

PRESSKEY places one keystroke in the Windows keyboard input buffer. Once placed in the keyboard buffer, the *keycode* is processed just as if the user had pressed the key. ALIAS does not transform a PRESSKEY *keycode*.

Example:

```
IF Action = 'Add'         !On the way into a memo control on an add record
  Cus:MemoControl = FORMAT(TODAY(),@D1) & ' ' & FORMAT(CLOCK(),@T4)
                          !Pre-load first line of memo with date and time
  PRESSKEY(EnterKey)      ! and position user on second line
END
```

SETKEYCODE (specify keycode)

SETKEYCODE(*keycode***)**

SETKEYCODE Sets the keycode returned by the KEYCODE function.

keycode An integer constant or keycode EQUATE label.

SETKEYCODE sets the internal keycode returned by the KEYCODE function. The keycode is not put into the keyboard buffer.

Example:

```
SETKEYCODE(0800h)       !Set up the keycode function to return 0800h
```

See Also: KEYCODE, Keycode Equate Labels

Keyboard Functions

KEYBOARD (return keystroke waiting)

KEYBOARD( )

The **KEYBOARD** function returns the keycode of the first keystroke in the keyboard buffer. It is used to determine if there are keystrokes waiting to be processed by an ASK or ACCEPT statement.

Return Data Type: LONG

Example:

```
LOOP UNTIL KEYBOARD()                   !Wait for any key
  ASK
  IF KEYCODE() = EscKey THEN BREAK.     !On esc key, break the loop
END
```

See Also: ASK, ACCEPT, Keycode Equate Labels

KEYCHAR (return ASCII code)

KEYCHAR( )

The **KEYCHAR** function returns the ASCII value of the last key pressed at the time the event occurred.

Return Data Type: LONG

Example:

```
ACCEPT                  !Wait for an event
  CASE KEYCHAR()        !Process the last keystroke
  OF 'A' TO 'Z'         ! upper case?
    DO ProcessUpper
  OF 'a' TO 'z'         ! lower case?
    DO ProcesLower
  END
END
```

See Also: ASK, ACCEPT, SELECT, Keycode Equate labels

KEYCODE (return last keycode)

KEYCODE( )

The **KEYCODE** function returns the keycode of the last key pressed at the time the event occurred, or the last keycode value set by the SETKEYCODE procedure.

Return Data Type: LONG

Example:

```
ACCEPT                  !Loop on the display
  CASE KEYCODE()        !Process the keystroke
  OF UpKey              ! up arrow
    DO GetRecordUp      !   get a record
  OF DownKey            ! down arrow
    DO GetRecordDn      !   get a record
  END
END
```

See Also: ASK, ACCEPT, SELECT, Keycode Equate labels

KEYSTATE (return keyboard status)

KEYSTATE( )

The **KEYSTATE** function returns a bitmap containing the status of the SHIFT, CTRL, ALT, any extended key, CAPS LOCK, NUM LOCK, SCROLL LOCK, and INSERT keys for the last KEYCODE function return value. The bitmap is contained in the high-order byte of the returned SHORT.

```
x . . . . . . . insert key  (8000h)
. x . . . . . . scroll lock (4000h)
. . x . . . . . num lock    (2000h)
. . . x . . . . caps lock   (1000h)
. . . . x . . . extended    (0800h)
. . . . . x . . alt         (0400h)
. . . . . . x . ctrl        (0200h)
. . . . . . . x shift       (0100h)
```

Return Data Type: SHORT

Example:

```
ACCEPT                              !Loop on the display
  CASE KEYCODE()                    !Process the keystroke
  OF EnterKey                       ! up arrow
    IF BAND(KEYSTATE(),0800h)       !Detect enter on numeric keypad
      PRESSKEY(TabKey)              ! press tab for the user
    END
  END
END
```

See Also: KEYCODE, BAND

Windows Standard Dialog Functions

COLORDIALOG (return chosen color)

COLORDIALOG([*title*] [,*rgb*] **)**

COLORDIALOG Displays the Windows standard color choice dialog box to allow the user to choose a color.

title A string constant or variable containing the title to place on the color choice dialog. If omitted, a default *title* is supplied by Windows.

rgb A LONG integer variable to receive the selected color.

The **COLORDIALOG** function displays the Windows standard color choice dialog box and returns the color chosen by the user in the *rgb* parameter. Any existing value in the *rgb* parameter sets the default color choice presented to the user in the color choice dialog.

COLORDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the color choice dialog.

Return Data Type: SHORT

Example:

```
MDIChild1  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
             !window controls
           END

ColorNow  LONG

  CODE
  IF NOT COLORDIALOG('Choose Box Color',ColorNow)
    ColorNow = 000000FFh            !Default to Blue if user pressed Cancel
  END
  OPEN(MDIChild1)
  BOX(100,50,100,50,ColorNow)       !User-defined color for box
```

FILEDIALOG (return chosen file)

FILEDIALOG([*title*] [,*file*] [,*extensions*] [,*flag*] **)**

FILEDIALOG	Displays the Windows standard file choice dialog box to allow the user to choose a file.
title	A string constant or variable containing the title to place on the file choice dialog. If omitted, a default *title* is supplied by Windows.
file	A string variable to receive the selected filename.
extensions	A string constant or variable containing the available file extension selections for the List Files of Type drop list. If omitted, the default is all files (*.*).
flag	An integer constant or variable to indicate type of file action to perform. If omitted, or zero (0), the Open... dialog is displayed and the user is warned if the file they choose does not exist (the file is not automatically opened). If one (1), the Save... dialog is displayed and the user is warned if the file does exist (the file is not automatically saved).

The **FILEDIALOG** function displays the Windows standard file choice dialog box and returns the file chosen by the user in the *file* parameter. Any existing value in the *file* parameter sets the default file choice presented to the user in the file choice dialog.

The *extensions* parameter string must contain a description followed by the file mask. All elements in the string must be delimited by the vertical bar (|) character. For example, the *extensions* string 'All Files | *.* | Clarion Source | *.CLW' defines two selections for the List Files of Type drop list. The first extension listed in the *extensions* string is the default.

FILEDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the file choice dialog.

Return Data Type: SHORT

Example:

```
ViewTextFile  PROCEDURE

ViewQue       QUEUE                    !LIST control display queue
                 STRING(255)
              END

FileName      STRING(64),STATIC        !Filename variable

ViewFile      FILE,DRIVER('ASCII'),NAME(FileName),PRE(Vew)
Record          RECORD
                  STRING(255)
                END
              END

MDIChild1 WINDOW('View Text File'),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
             LIST,AT(0,0,320,200),USE(?L1),FROM(ViewQue),HVSCROLL
          END

  CODE
  IF NOT FILEDIALOG('Choose File to View',FileName,'Text|*.TXT|Source|*.CLW',0)
    RETURN                             !Return if no file chosen
  END
  OPEN(ViewFile)                       !Open the file
    IF ERRORCODE() THEN RETURN.        ! aborting on any error
  SET(ViewFile)                        !Start at top of file
  LOOP
    NEXT(ViewFile)                     !Reading each line of text
      IF ERRORCODE() THEN BREAK.       !Break loop at end of file
    ViewQue = Vew:Record               !Assign text to queue
    ADD(ViewQue)                       ! and add a queue entry
  END
  CLOSE(ViewFile)                      !Close the file
  OPEN(MDIChild1)                      ! and open the window
  ACCEPT                               !Allow the user to read the text and
  END                                  ! break out of ACCEPT loop only from
                                       ! system menu close option
  FREE(ViewQue)                        !Free the queue memory
  RETURN                               ! and return to caller
```

FONTDIALOG (return chosen font)

FONTDIALOG([*title*] [,*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONTDIALOG	Displays the standard Windows font choice dialog box to allow the user to choose a font.
title	A string constant or variable containing the title to place on the font choice dialog. If omitted, a default *title* is supplied by Windows.
typeface	A string variable to receive the name of the chosen font.
size	An integer variable to receive the size (in points) of the chosen font.
color	A LONG integer variable to receive the red, green, and blue values for the color of the chosen font in the low-order three bytes.
style	An integer variable to receive the strike weight and style of the chosen font.

The **FONTDIALOG** function displays the Windows standard font choice dialog box to allow the user to choose a font. When called, any values in the parameters set the default font values presented to the user in the font choice dialog. They also receive the user's choice when the user presses the Ok button on the dialog.

FONTDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button.

Return Data Type: SHORT

Example:

```
MDIChild1   WINDOW('View Text File'),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
              !window controls
            END
Typeface    STRING(20)
FontSize    LONG
FontColor   LONG
FontStyle   LONG
  CODE
  OPEN(MDIChild1)                       !open the window
  IF FONTDIALOG('Choose Display Font',Typeface,FontSize,FontColor,FontStyle)
    SETFONT(0,Typeface,FontSize,FontColor,FontStyle) !Set window font
  ELSE
    SETFONT(0,'Arial',12)               !Set default font
  END
  ACCEPT
    !Window handling code
  END
```

PRINTERDIALOG (return chosen printer)

PRINTERDIALOG([*title*] [,*flag*] **)**

PRINTERDIALOG

Displays the Windows standard printer choice dialog box to allow the user to choose a file.

title A string constant or variable containing the title to place on the file choice dialog. If omitted, a default *title* is supplied by Windows.

flag A numeric constant or variable which, if non-zero, displays the Print Setup dialog instead of the printer choice dialog. This is the same dialog as called by placing STD:PrintSetup in the STD attribute of a menu item.

The **PRINTERDIALOG** function displays the Windows standard printer choice dialog box (or the Print Setup dialog) and returns the printer chosen by the user in the PRINTER "built-in" variable in the internal library. This sets the default printer used for the next REPORT opened.

PRINTERDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the dialog.

Return Data Type: SHORT

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS,FONT('Arial',12),PRE(Rpt)
            !Report structures and controls
        END

  CODE
  IF NOT PRINTERDIALOG('Choose Printer')
    RETURN                              !Abort if user pressed Cancel
  END
  OPEN(CustRpt)
```

Drag and Drop Processing

Drag-and-drop is a very powerful Windows tool that allows a user to copy or move data from one control to another (or even within the same control). These controls may be in the same window, separate windows in the same application, or even separately executing Clarion applications.

Implementing drag-and-drop in a Clarion application involves two processes:

- Specifying drag host and drop target controls.
- Performing the data exchange when the user initiates drag-and-drop by handling the drag-and-drop events.

To specify a drag host, you place the DRAGID attribute on a LIST or REGION control with a set of "signatures" that verify valid drop targets for the data. To specify a drop target, you place the DROPID attribute on a control to list the set of valid drag "signatures" from which the control will accept data. Drag-and-drop operations only occur between controls with matching "signatures" in their respective DRAGID and DROPID attributes.

A successful drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute and both controls have at least one identical *signature* string in their respective DRAGID and DROPID attributes. When the user initiates drag-and-drop, EVENT:Dragging is posted to the host control whenever the mouse is over a potential target control (valid or not). EVENT:Drag is posted to the host control when the user releases the mouse button over a potential target control (valid or not). EVENT:Drop is posted to the target control only if it is a valid match.

The DRAGID() function detects the successful drop. The DROPID() function can also detect a successful drop, or can pass the exchanged data as a string, if its value is set by the SETDROPID procedure. The actual data exchange between the controls can be accomplished several ways:

- If the two controls are in the same window, you can exchange data using local or global variables, the DROPID function can exchange the data, or you can use the Windows clipboard.
- If the two controls are in the same application, you can exchange data using global variables, the DROPID function can exchange the data, or you can use the Windows clipboard.
- If the controls are in separate Clarion applications, you must use SETDROPID to have the DROPID function exchange the data, or use the Windows clipboard.

You can copy or move the data to the target control, depending upon how you write the data exchange code. Also, you should write the data exchange code for the most difficult coding circumstance. Therefore, if the drag host might be an external program's control, you could pass the data through the DROPID() function (using SETDROPID), or through the Windows clipboard. If the drag host could be a control in any window within the program, you should either pass the data through the DROPID() function, or use global variables. Only for those instances where the drag host and drop target are always going to be in the same procedure should you use local variables.

CLIPBOARD (return windows clipboard contents)

CLIPBOARD( )

The **CLIPBOARD** function returns the current contents of the windows clipboard.

Return Data Type: STRING

Example:

```
Que1  QUEUE
        STRING(30)
      END

Que2  QUEUE
        STRING(30)
      END

WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
                                           !Allows drags, but not drops
          LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1','~FILE')
                                           !Allows drops from List1, but no drags
        END

  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag                    !When a drag event is attempted
      IF DRAGID()                    ! check for success
        SETCLIPBOARD(Que1)           ! and setup info to pass
      END
    OF EVENT:Drop                    !When drop event is successful
      Que2 = CLIPBOARD()             ! get dropped info
      ADD(Que2)                      ! and add it to the queue
    END
  END
```

See Also: SETCLIPBOARD

DRAGID (return matching drag-and-drop signature)

DRAGID([*thread*] [, *control*] **)**

DRAGID	Returns matching host and target signatures on a successful drag-and-drop operation.
thread	The label of a numeric variable to receive the thread number of the host control. If the host control is in an external program, *thread* receives zero (0).
control	The label of a numeric variable to receive the field equate label of the host control.

The **DRAGID** function returns the matching host and target control signatures on a successful drag-and-drop operation. If the user aborted the operation, DRAGID returns an empty string (‘’), otherwise it returns the first signature that matched between the two controls.

Return Data Type: STRING

Example:

```
Que1  QUEUE
        STRING(30)
      END
Que2  QUEUE
        STRING(30)
      END
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
                                      !Allows drags, but not drops
          LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1')
                                      !Allows drops from List1, but no drags
        END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag                     !When a drag event is attempted
      IF DRAGID()                     ! check for success
        SETDROPID(Que1)               ! and setup info to pass
      END
    OF EVENT:Drop                     !When drop event is successful
      Que2 = DROPID()                 ! get dropped info
      ADD(Que2)                       ! and add it to the queue
    END
  END
```

See Also: DROPID, SETDROPID

DROPID (return drag-and-drop string)

DROPID([*thread*] [, *control*] **)**

DROPID	Returns matching host and target signatures on a successful drag-and-drop operation.
thread	The label of a numeric variable to receive the thread number of the target control. If the target control is in an external program, *thread* receives zero (0).
control	The label of a numeric variable to receive the field equate label of the target control.

The **DROPID** function returns the matching host and target control signatures on a successful drag-and-drop operation (just as DRAGID does), or the specific string set by the SETDROPID procedure. The DROPID function returns a comma-delimited list of filenames dragged from the Windows File Manager when '~FILE' is the DROPID attribute.

Return Data Type: STRING

Example:

```
DragDrop    PROCEDURE
Que1  QUEUE
        STRING(90)
      END
Que2  QUEUE
        STRING(90)
      END
WinOne  WINDOW('Test Drag Drop'),AT(10,10,240,320),SYSTEM,MDI
          LIST,AT(12,0,200,80),USE(?List1),FROM(Que1),DRAGID('List1')
                              !Allows drags, but not drops
          LIST,AT(12,120,200,80),USE(?List2),FROM(Que2),DROPID('List1','~FILE')
                              !Allows drops from List1 or the Windows File Manager,
        END                   ! but no drags
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag             !When a drag event is attempted
      IF DRAGID()             ! check for success
        GET(Que1,CHOICE())
        SETDROPID(Que1)       ! and setup info to pass
      END
    OF EVENT:Drop             !When drop event is successful
      IF INSTRING(',',DROPID(),1,1)     !Check for multiple files from File Manager
        Que2 = SUB(DROPID(),1,INSTRING(',',DROPID(),1,1)-1)  ! and only get first
        ADD(Que2)             ! and add it to the queue
      ELSE
        Que2 = DROPID()       ! get dropped info, from List1 or File Manager
        ADD(Que2)             ! and add it to the queue
      END
    END
  END
```

See Also: DRAGID, SETDROPID

SETCLIPBOARD (set windows clipboard contents)

SETCLIPBOARD(*string* **)**

SETCLIPBOARD Puts information in the Windows clipboard.

string A string constant or variable containing the information to place in the Windows clipboard.

The **SETCLIPBOARD** procedure places the contents of the *string* into the Windows clipboard, overwriting any previous contents.

Example:

```
Que1  QUEUE
        STRING(30)
      END
Que2  QUEUE
        STRING(30)
      END
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
                                        !Allows drags, but not drops
          LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1')
                                        !Allows drops from List1, but no drags
        END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag                       !When a drag event is attempted
      IF DRAGID()                       ! check for success
        SETCLIPBOARD(Que1)              ! and setup info to pass
      END
    OF EVENT:Drop                       !When drop event is successful
      Que2 = CLIPBOARD()                ! get dropped info
      ADD(Que2)                         ! and add it to the queue
    END
  END
```

See Also: CLIPBOARD

SETDROPID (set DROPID return string)

SETDROPID(*string* **)**

SETDROPID Sets the DROPID function's return value.

string A string constant or variable containing the value the DROPID function will return.

The **SETDROPID** procedure sets the DROPID function's return value. This allows the DROPID function to pass the data in a drag-and-drop operation. When drag-and-drop operations are performed between separate Clarion applications, this is the mechanism to use to pass the data.

Example:

```
Que1  QUEUE
        STRING(30)
      END
Que2  QUEUE
        STRING(30)
      END
WinOne  WINDOW,AT(0,0,160,400)
          LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID('List1')
                            !Allows drags, but not drops
          LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID('List1')
                            !Allows drops from List1 or the Windows File Manager,
                            ! but no drags
        END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag           !When a drag event is attempted
      IF DRAGID()           ! check for success
        SETDROPID(Que1)     ! and setup info to pass
      END
    OF EVENT:Drop           !When drop event is successful
      Que2 = DROPID()       ! get dropped info, from List1 or File Manager
      ADD(Que2)             ! and add it to the queue
    END
  END
```

See Also: DRAGID, DROPID

Maintaining INI Files

GETINI (return INI file entry)

GETINI(*section* ,*entry* [,*default*] [,*file*] **)**

GETINI	Returns the value for an INI file entry.
section	A string constant or variable containing the name of the portion of the INI file which contains the *entry*.
entry	A string constant or variable containing the name of the specific setting for which to return the value.
default	A string constant or variable containing the default value to return if the *entry* does not exist. If omitted and the entry does not exist, GETINI returns an empty string.
file	A string constant or variable containing the name of the INI file to search (looks for the *file* in the Windows directory unless a full path is specified). If omitted, GETINI searches the WIN.INI file.

The **GETINI** function returns the value of an *entry* in a Windows-standard INI file. A Windows-standard INI file is an ASCII text file with the following format:

```
[some section name]
entry=value
next entry=another value
```

For example, WIN.INI contains entries such as:

```
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
```

The GETINI function searches the specified *file* for the *entry* within the *section* you specify. It returns everything on the *entry's* line of text that appears to the right of the equal sign (=).

Return Data Type: STRING

Example:

```
Value   STRING(30)
  CODE
  Value = GETINI('intl','sLanguage')      !Get the language entry
```

See Also: PUTINI

PUTINI (set INI file entry)

PUTINI(*section* ,*entry* [,*value*] [,*file*] **)**

PUTINI	Sets the value for an INI file entry.
section	A string constant or variable containing the name of the portion of the INI file which contains the *entry*.
entry	A string constant or variable containing the name of the specific entry to set.
value	A string constant or variable containing the setting to place in the *entry*. An empty string (‘’) leaves the *entry* empty. If omitted, the *entry* is deleted.
file	A string constant or variable containing the name of the INI file to search (looks for the *file* in the Windows directory unless a full path is specified). If omitted, PUTINI places the *entry* in the WIN.INI file.

The **PUTINI** procedure places the *value* into an *entry* in a Windows-standard .INI file. A Windows-standard .INI file is an ASCII text file with the following format:

```
[some section name]
entry=value
next entry=another value
```

For example, WIN.INI contains entries such as:

```
[windows]
spooler=yes
load=nwpopup.exe
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
```

The PUTINI function searches the specified *file* for the *entry* within the *section* you specify. It replaces the current entry value with the *value* you specify. If necessary, the *section* and *entry* are created.

Example:

```
CODE
PUTINI('MyApp','SomeSetting','Initialized')  !Place setting in WIN.INI
PUTINI('MyApp','ASetting','2','MYAPP.INI')   !Place setting in MYAPP.INI
```

See Also: GETINI

Reports in Windows

Contents

Clarion for Windows reports use a page-based printing paradigm instead of a line-based paradigm. Instead of printing each line as it's values are generated, nothing is sent to the printer until an entire page is ready to print. This means that the "print engine" in the Clarion runtime library can do a lot of work for you, based on the attributes you specify in the REPORT structure.

Some of the things that the "print engine" in the Clarion runtime library does for you are:

- Prints "pre-printed" forms on each page, that are then filled in by the data
- Calculates totals (count, sum, average, minimum, maximum)
- Provides automatic page break handling, including page headers and footers
- Provides automatic group break handling, including group headers and footers
- Provides complete widow/orphan control.

This automatic functionality makes the executable code required to print a complex report very small, making your programming job easier.

Since the "print engine" is page-based, the concepts of headers and footers lose their context indicating both page positioning and print sequence, and only retain their meaning of print sequence. Headers are printed at the beginning of a print sequence, and footers are printed at the end—their actual positioning on the page is irrelevant. For example, you could position the page footer, containing page totals, to print at the top of the page.

Page Overflow

Page Overflow occurs when the PRINT statement cannot fit a DETAIL structure on a page. This may be due to a lack of space, or the presence of the PAGEBEFORE or PAGEAFTER attribute on a DETAIL structure. The following steps occur during page overflow, in this sequence:

1 If the REPORT has a page FOOTER, it is printed at the position specified by its AT attribute.

2 The page counter is incremented.

3 If the REPORT has a FORM structure, it is printed at the position specified by its AT attribute.

4 If the REPORT has a page HEADER, it is printed at the position specified by its AT attribute.

Report Structure

REPORT (declare a report structure)

```
label   REPORT([jobname]), AT( ) [, FONT( )] [, PRE( )] [, LANDSCAPE] [, PREVIEW] [, PAPER]
                                                                    [ | THOUS   | ]
                                                                      | MM      |
                                                                      | POINTS  |
              [FORM
                 controls
               END ]
              [HEADER
                 controls
               END ]
label         DETAIL
                 controls
               END
label         [BREAK( )
                 group break structures
               END ]
              [FOOTER
                 controls
               END ]
        END
```

REPORT	Declares the beginning of a report data structure.
label	The name by which the structure is addressed in executable code.
jobname	Names the print job for the Windows Print Manager. If omitted, the REPORT's *label* is used.
AT	Specifies the size and location, relative to the top left corner of the page, of the area devoted to printing report detail.
FONT	Specifies the default font for all controls in this report. If omitted, the printer's default font is used.
PRE	Specifies the label prefix for the report or structure.
LANDSCAPE	Specifies printing in landscape mode. If omitted, printing defaults to portrait mode.
PREVIEW	Specifies report output to Windows metafiles (.WMF); one file per report page.
PAPER	Specifies the paper size for the report output. If omitted, the default printer's paper size is used.
THOUS	Specifies thousandths of an inch as the measurement unit used for all attributes which use coordinates.
MM	Specifies millimeters as the measurement unit used for all attributes which use coordinates.

POINTS	Specifies points as the measurement unit used for all attributes which use coordinates. There are 72 points per inch, vertically and horizontally.
FORM	Page layout structure defining pre-printed items on every page.
controls	Report output controls.
HEADER	Page header structure, printed at the beginning of each page.
DETAIL	Report detail structure.
BREAK	A group break structure, defining the variable which causes a group break to occur when its value changes.
group break structures	Group break HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures.
FOOTER	Page footer structure, printed at the end of each page.

The **REPORT** statement declares the beginning of a report data structure. A REPORT structure must be terminated with a period or END statement. Within the REPORT, the FORM, HEADER, DETAIL, FOOTER, and BREAK structures are the components that format the output of the report. A REPORT must be explicitly opened with the OPEN statement.

A REPORT with the PREVIEW attribute sends the report output to Windows metafiles (.WMF) containing one report page per file. The PREVIEW attribute names a QUEUE to receive the names of the metafiles. You can then create a window to display the report in an IMAGE control, using the QUEUE field contents (the file names) to set the IMAGE control's {PROP:text} property. This allows the end user to view the report before printing.

Only DETAIL structures can (and must) be printed with the PRINT statement. All other report structures (HEADER, FOOTER, and FORM) are automatically printed for you at the appropriate place in the report.

The REPORT's AT attribute defines the area of each page devoted to printing DETAIL structures. This includes any HEADERs and FOOTERs that are contained within a BREAK structure (group headers and footers).

The FORM structure is printed on every page except pages containing DETAIL structures with the ALONE attribute. Its format is determined once at the beginning of the report. This makes it the logical place to design a pre-printed form template, which is filled in by the subsequent HEADER, DETAIL, and FOOTER structures.

The page HEADER and FOOTER structures are not within a BREAK structure. They are automatically printed whenever a page break occurs.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes.

A REPORT data structure never defaults as the current target for runtime property assignment the way the most recently opened WINDOW or APPLICATION structure does. Therefore, the REPORT *label* must be explicitly named as the target, or the SETTARGET statement must be used to make the REPORT the current target, before using runtime property assignment to a REPORT control. Since the graphics commands draw graphics only to the current target, the SETTARGET statement must be used to make the REPORT the current target before using the graphics functions on a REPORT.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS,FONT('Arial',12),PRE(Rpt)
          FORM,AT(1000,1000,6500,9000)
            IMAGE('LOGO.BMP'),AT(0,0,1200,1200),USE(?I1)
            STRING(@n3),AT(6000,500,500,500),PAGENO
          END
          HEADER,AT(1000,1000,6500,1000)
            STRING('ABC  Company'),AT(3000,500,1500,500),FONT('Arial',18)
          END
Break1    BREAK(Pre:Key1)
            HEADER,AT(0,0,6500,1000)
              STRING('Group  Head'),AT(3000,500,1500,500),FONT('Arial',18)
            END
Detail      DETAIL,AT(0,0,6500,1000)
              STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1)
            END
            FOOTER,AT(0,0,6500,1000)
              STRING('Group  Total:'),AT(5500,500,1500,500)
              STRING(@N$11.2'),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Break1)
            END
          END
          FOOTER,AT(1000,1000,6500,1000)
            STRING('Page  Total:'),AT(5500,1500,1500,500)
            STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1),SUM,PAGE
          END
        END                                          !End report declaration

  CODE
  OPEN(CustReport)
  SET(DataFile)
  LOOP
    NEXT(DataFile)
      IF ERRORCODE() THEN BREAK.
    PRINT(Rpt:Detail)
  END
  CLOSE(CustReport)
```

AT (set detail print area)

AT([*x*] [,*y*] [,*width*] [,*height*]**)**

AT	Defines the position and size of the area of the page devoted to printing report detail.
x	An integer constant or constant expression that specifies the horizontal position of the top left corner of the detail area.
y	An integer constant or constant expression that specifies the vertical position of the top left corner of the detail area.
width	An integer constant or constant expression that specifies the width of the detail area.
height	An integer constant or constant expression that specifies the height of the detail area.

The **AT** attribute on a REPORT structure defines the position and size of the area of the page devoted to printing report detail. This includes the area to print all DETAIL structures and any group HEADER and FOOTER structures contained within BREAK structures.

The values contained in the *x*, *y*, *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer's system default font.

Example:

```
CustRpt1  REPORT,AT(1000,1000,6500,9000),THOUS  !1" margins all around for detail
                                                ! area on 8.5" x 11" paper
            !report declarations
          END

CustRpt2  REPORT,AT(72,72,468,648),POINTS       !1" margins all around for detail
                                                ! area on 8.5" x 11" paper
            !report declarations
          END
```

FONT (set report default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default print font.
typeface	A string constant containing the name of the font. If omitted, the printer's default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the printer's default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on a REPORT structure specifies the default print font for all controls in the REPORT. This font is used when the control does not have a FONT attribute or its own, and the print structure it is in also has no FONT attribute.

The *typeface* may name any font registered in the Windows system which the printer driver supports. This includes the TrueType fonts for most printers. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

```
FONT:thin        EQUATE (100)
FONT:regular     EQUATE (400)
FONT:bold        EQUATE (700)
FONT:italic      EQUATE (01000H)
FONT:underline   EQUATE (02000H)
FONT:strikeout   EQUATE (04000H)
```

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS, |
                    FONT('Arial',12,,FONT:Bold+FONT:Italic)
          !report declarations
          END
```

PRE (set report label prefix)

PRE([*prefix*] **)**

PRE Provides a label prefix for structures in the report.

prefix A string constant containing the prefix for labels within the REPORT. Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alphabet character and must not be a reserved word. By convention, a *prefix* is 1-3 characters, although it can be longer.

The **PRE** attribute on a REPORT provides a label prefix for DETAIL and BREAK structures. It is used to distinguish between identical names that occur in different structures. When referenced in executable statements, the *prefix* is attached to a label by a colon (Pre:Label). You can also use the Field Qualification syntax to reference DETAIL and BREAK strutcures.

Example:

```
Report    REPORT,PRE('Rpt')
DetailOne   DETAIL              !Always referenced as Rpt:DetailOne
              !Report controls
            END                 ! in executable code
          END
```

See Also: Reserved Words

PREVIEW (set report output to metafiles)

PREVIEW(*queue*)

PREVIEW	Specifies report output goes to Windows metafiles (.WMF) containing one report page per file.
queue	The label of a QUEUE or a field in a QUEUE to receive the names of the metafiles.

The **PREVIEW** attribute on a REPORT sends the report output to Windows metafiles (.WMF) containing one report page per file. The PREVIEW attribute names a *queue* to receive the names of the metafiles. The filenames are temporary filenames internally created by the Clarion library and are complete file specifications (up to 64 characters, including drive and path). These temporary files are deleted from disk when you CLOSE the REPORT.

You can create a window to display the report in an IMAGE control, using the *queue* containing the file names to set the IMAGE control's {PROP:text} property. This allows the end user to view the report before printing. A runtime-only property, {PROP:flushpreview}, when set to ON, flushes the metafiles to the printer.

Example:

```
SomeReport  PROCEDURE

WMFQue      QUEUE                   !Queue to contain .WMF filenames
              STRING(64)
            END

NextEntry  BYTE(1)                  !Queue entry counter variable

Report    REPORT,PREVIEW(WMFQue)    !Report with PREVIEW attribute
DetailOne   DETAIL
              !Report controls
            END
          END

ViewReport  WINDOW('View  Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(''),AT(0,0,320,180),USE(?ImageField)
              BUTTON('View  Next  Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
              BUTTON('Print  Report'),AT(80,180,60,20),USE(?PrintReport)
              BUTTON('Exit  Without  Printing'),AT(160,180,60,20),USE(?ExitReport)
            END

  CODE
  OPEN(Report)
  SET(SomeFile)                             !Code to generate the report
  LOOP
    NEXT(SomeFile)
      IF ERRORCODE() THEN BREAK.
    PRINT(DetailOne)
  END
  ENDPAGE(Report)
  OPEN(ViewReport)                          !Open report preview window
  GET(WMFQue,NextEntry)                     !Get first queue entry
  ?ImageField{PROP:text} = WMFQue           !Load first report page
  ACCEPT
    CASE ACCEPTED()
    OF ?NextPage
      NextEntry += 1                        !Increment entry counter
      IF NextEntry > RECORDS(WMFQue) THEN CYCLE. !Check for end of report
      GET(WMFQue,NextEntry)                 !Get next queue entry
      ?ImageField{PROP:text} = WMFQue       !Load next report page
      DISPLAY                               ! and display it
    OF ?PrintReport
      Report{PROP:flushpreview} = ON        !Flush files to printer
      BREAK                                 ! and exit procedure
    OF ?ExitReport
      BREAK                                 !Exit procedure
    END
  END
  CLOSE(ViewReport)                         !Close window
  FREE(WMFQue)                              !Free the queue memory
  CLOSE(Report)                             !Close report (deleting all .WMF files)
  RETURN                                    ! and return to caller
```

PAPER (set report paper size)

PAPER([*type*] [,*width*] [,*height*]**)**

PAPER	Defines the paper size for the report.
type	An integer constant or EQUATE that specifies a standard Windows paper size. EQUATES for these are contained in the EQUATES.CLW file.
width	An integer constant or constant expression that specifies the width of the paper.
height	An integer constant or constant expression that specifies the height of the paper.

The **PAPER** attribute on a REPORT structure defines the paper size for the report. The *width* and *height* parameters are only required when PAPER:Custom is selected as the *type*.

The values contained in the *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer's default font.

Example:

```
CustRpt1  REPORT,AT(1000,1000,6500,9000),THOUS,PAPER(PAPER:Custom,8500,7000)
                                                ! print on 8.5" x 7" paper
            !report declarations
          END

CustRpt2  REPORT,AT(72,72,468,648),POINTS,PAPER(PAPER:A4)
                                                ! print on A4 size paper
            !report declarations
          END
```

LANDSCAPE (set page orientation)

LANDSCAPE

The **LANDSCAPE** attribute on a REPORT indicates the report is to print in landscape mode by default. If the LANDSCAPE attribute is omitted, printing defaults to portrait mode.

Example:

```
Report  REPORT,PRE('Rpt'),LANDSCAPE      !Defaults to landscape mode
          !Report structure declarations
        END
```

THOUS, MM, POINTS (set report coordinate measure)

THOUS
MM
POINTS

The **THOUS**, **MM**, and **POINTS** attributes specify the coordinate measures used to position controls on the REPORT.

THOUS specifies thousandths of an inch, **MM** specifies millimeters, and **POINTS** specifies points (there are seventy-two points per inch, both vertically and horizontally).

If all these attributes are omitted, the measurements default to dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the REPORT, or the system default font specified by Windows.

Print Structures

BREAK (declare group break structure)

```
label   BREAK(variable)
            group break structures
        END
```

BREAK Declares a group break structure.

label The name by which the structure is addressed in executable code.

variable The variable whose change in value signals the group break.

group break structures
Group break HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures.

The **BREAK** structure declares the *variable* which signals a group break when the value in the *variable* changes. A BREAK structure must be terminated with a period or END statement. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. Only one HEADER and FOOTER are allowed in a BREAK structure; it may contain multiple DETAIL and/or BREAK structures.

The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in the group break *variable* changes.

Example:

```
CustRpt   REPORT                        !Declare customer report
Break1      BREAK(SomeVariable)
              HEADER                    ! begin group header declaration
                !report controls
              END                       ! end header declaration
GroupDet      DETAIL
                !report controls
              END                       ! end detail declaration
              FOOTER                    ! begin group footer declaration
                !report controls
              END                       ! end footer declaration
            END                         ! end group break declaration
          END                           !End report declaration
```

DETAIL (report detail line structure)

```
label   DETAIL ,AT( ) [,FONT( )] [,ALONE] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
                [,WITHPRIOR( )] [,WITHNEXT( )] [,USE( )]
          controls
        END
```

DETAIL	Declares items to be printed as the body of the report.
label	The name by which the structure is addressed in executable code.
AT	Specifies the offset and minimum width and height of the DETAIL, relative to the size of the area specified by the REPORT's AT attribute.
FONT	Specifies the default font for all controls in this structure. If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
ALONE	Declares the DETAIL structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures.
ABSOLUTE	Declares the DETAIL prints at a fixed position relative to the page.
PAGEBEFORE	Declares the DETAIL prints at the start of a new page, after normal page overflow actions.
PAGEAFTER	Declares the DETAIL prints, and then starts a new page by activating normal page overflow actions.
WITHPRIOR	Declares the DETAIL prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it.
WITHNEXT	Declares the DETAIL prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it.
USE	A field equate label to reference the DETAIL structure in executable code.
controls	Report output control fields.

The **DETAIL** structure declares items to be printed as the body of the report. A DETAIL structure must be terminated with a period or END statement. A REPORT may have multiple DETAIL structures.

A DETAIL structure is never automatically printed, therefore DETAIL structures are always explicitly printed by the PRINT statement. This means that a *label* is required for each DETAIL you wish to PRINT.

The DETAIL structure may be printed whenever necessary. Since you may have multiple DETAIL structures, they provide the ability to optionally print

alternate print formats. This is determined by the logic in the executable code which prints the report.

Example:

```
CustRpt     REPORT                    !Declare customer report
              HEADER                  ! begin page header declaration
                !structure elements
              END                     ! end header declaration
CustDetail1   DETAIL                  ! begin detail declaration
                !structure elements
              END                     ! end detail declaration
CustDetail2   DETAIL                  ! begin detail declaration
                !structure elements
              END                     ! end detail declaration
            END                       !End report declaration
  CODE
  OPEN(CustRpt)
  SET(SomeFile)
  LOOP
    NEXT(SomeFile)
      IF ERRORCODE() THEN BREAK.
    IF SomeCondition
      PRINT(CustDetail1)
    ELSE
      PRINT(CustDetail2)
    END
  END
  CLOSE(CustRpt)
```

See Also: PRINT

FOOTER (page or group footer structure)

```
FOOTER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
       [,WITHPRIOR( )] [,WITHNEXT( )] [,ALONE] [,USE( )]
  controls
END
```

FOOTER	Declares a page or group footer structure.
AT	Specifies the size and location of the FOOTER.
FONT	Specifies the default font for all controls in this structure. If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
ABSOLUTE	Declares the FOOTER prints at a fixed position relative to the page. Valid only on a FOOTER within a BREAK structure (page FOOTER position is always fixed).
PAGEBEFORE	Declares the FOOTER prints at the start of a new page, after normal page overflow actions. Valid only on a FOOTER within a BREAK structure.
PAGEAFTER	Declares the FOOTER prints, and then starts a new page by activating normal page overflow actions. Valid only on a FOOTER within a BREAK structure.
WITHPRIOR	Declares the FOOTER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it. Valid only on a FOOTER within a BREAK structure.
WITHNEXT	Declares the FOOTER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it. Valid only on a FOOTER within a BREAK structure.
ALONE	Declares the (group) FOOTER structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures.
USE	A field equate label to reference the FOOTER structure in executable code.
controls	Report output control fields.

The **FOOTER** structure declares the output which prints at the end of each page or group. A FOOTER structure must be terminated with a period or END statement.

A FOOTER structure that is not within a BREAK structure is a page footer. Only one page FOOTER is allowed in a REPORT. The page FOOTER is automatically printed whenever a page break occurs, at the page-relative position specified by its AT attribute.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes, at the next position available in the detail print area (specified by the REPORT's AT attribute). Only one FOOTER is allowed in a BREAK structure.

Example:

```
CustRpt   REPORT                !Declare customer report
            FOOTER              ! begin page FOOTER declaration
              !report controls
            END                 ! end FOOTER declaration
Break1      BREAK(SomeVariable)
GroupDet      DETAIL
                !report controls
              END               ! end detail declaration
              FOOTER            ! begin group footer declaration
                !report controls
              END               ! end footer declaration
            END                 ! end group break declaration
          END                   !End report declaration
```

FORM (page layout structure)

```
FORM ,AT( ) [,FONT( )] [,USE( )]
  controls
END
```

FORM	Declares a report structure which prints on each page.
AT	Specifies the size and location, relative to the top left corner of the page, of the FORM.
FONT	Specifies the default font for all controls in this report structure. If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
USE	A field equate label to reference the FORM structure in executable code.
controls	Report output control fields.

FORM declares a report structure which prints on every page of the report (except pages containing DETAIL structures with the ALONE attribute). A FORM structure must be terminated with a period or END statement. Only one FORM is allowed in a REPORT structure. The FORM structure automatically prints during page overflow.

The printed output of the FORM is determined only once at the beginning of the report. The page positioning of the FORM does not affect the page positioning of any other report structure. Once printed, all other structures may "overwrite" the FORM. Therefore, FORM is most often used to design pre-printed forms which are filled in by the subsequent HEADER, DETAIL, and FOOTER structures. It may also be used to generate "watermarks" or page border graphics.

Example:

```
CustRpt REPORT            !Declare customer report
          FORM
             IMAGE('LOGO.BMP'),AT(0,0,1200,1200),USE(?I1)
             STRING(@N3),AT(6000,500,500,500),PAGENO
          END
GroupDet  DETAIL
             !report controls
          END
        END               !End report declaration
```

HEADER (page or group header structure)

```
HEADER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
       [,WITHPRIOR( )] [,WITHNEXT( )] [,ALONE] [,USE( )]
  controls
END
```

HEADER	Declares a page or group header structure.
AT	Specifies the size and location of the HEADER.
FONT	Specifies the default font for all controls in this structure. If omitted, the REPORT's FONT attribute (if present) is used, or else the printer's default font is used.
ABSOLUTE	Declares the HEADER prints at a fixed position relative to the page. Valid only on a HEADER within a BREAK structure (page HEADER position is always fixed).
PAGEBEFORE	Declares the HEADER prints at the start of a new page after normal page overflow actions. Valid only on a HEADER within a BREAK structure.
PAGEAFTER	Declares the HEADER prints, and then starts a new page by activating normal page overflow actions. Valid only on a HEADER within a BREAK structure.
WITHPRIOR	Declares the HEADER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it. Valid only on a HEADER within a BREAK structure.
WITHNEXT	Declares the HEADER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it. Valid only on a HEADER within a BREAK structure.
ALONE	Declares the (group) HEADER structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures.
USE	A field equate label to reference the HEADER structure in executable code.
controls	Report output control fields.

The **HEADER** structure declares the output which prints at the beginning of each page or group. A HEADER structure must be terminated with a period or END statement.

A HEADER structure that is not within a BREAK structure is a page header. Only one page HEADER is allowed in a REPORT. The page HEADER is automatically printed whenever a page break occurs, at the page-relative position specified by its AT attribute.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes, at the next position available in the detail print area (specified by the REPORT's AT attribute). Only one HEADER is allowed in a BREAK structure.

Example:

```
CustRpt REPORT              !Declare customer report
          HEADER            ! begin page header declaration
             !report controls
          END               ! end header declaration
Break1    BREAK(SomeVariable)
            HEADER          ! begin group header declaration
               !report controls
            END             ! end header declaration
GroupDet    DETAIL
               !report controls
            END             ! end detail declaration
          END               ! end group break declaration
        END                 !End report declaration
```

Print Structure Attributes

ABSOLUTE (set fixed-position printing)

ABSOLUTE

The **ABSOLUTE** attribute ensures that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), always prints at a fixed position on the page. When ABSOLUTE is present, the position specified by the *x* and *y* parameters of the structure's AT attribute is relative to the top left corner of the page.

Example:

```
CustRpt     REPORT
              HEADER
                !structure elements
              END
CustDetail1   DETAIL
                !structure elements
              END
CustDetail2   DETAIL,ABSOLUTE            ! fixed position detail
                !structure elements
              END
              FOOTER
                !structure elements
              END
            END
```

ALONE (set to print without page header, footer, or form)

ALONE

The **ALONE** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to be printed on the page without any FORM, page HEADER or FOOTER (not within a BREAK structure). The normal use is for report title and grand total pages.

Example:

```
CustRpt   REPORT
TitlePage   DETAIL,ALONE            !Title page detail structure
              !structure elements
            END
CustDetail  DETAIL
              !structure elements
            END
            FOOTER
              !structure elements
            END
          END
```

AT (set print structure position and size)

AT([*x*] [,*y*] [,*width*] [,*height*]**)**

AT	Defines the position and size at which the structure prints.
x	An integer constant or constant expression that specifies the horizontal position of the top left corner of the print structure.
y	An integer constant or constant expression that specifies the vertical position of the top left corner of the print structure.
width	An integer constant or constant expression that specifies the minumum width of the print structure.
height	An integer constant or constant expression that specifies the minimum height of the print structure.

The **AT** attribute on print structures performs two different functions, depending upon the structure on which it is placed.

When placed on a FORM, or page HEADER or FOOTER (not within a BREAK structure), the AT attribute defines the position and size on the page at which the structure is printed. The position specified by the *x* and *y* parameters is relative to the top left corner of the page.

When placed on a DETAIL, or group HEADER or FOOTER (contained within a BREAK structure) the print structure is printed according to the following rules (unless the ABSOLUTE attribute is also present):

- The *width* and *height* parameters of the AT attribute specify the minimum print size of the structure.
- The structure is actually printed at the next available position within the detail print area (specified by the REPORT's AT attribute).
- The position specified by the *x* and *y* parameters of the structure's AT attribute is an offset from the next available print position within the detail print area.
- The first print structure on the page is printed at the top left corner of the detail print area (at the offset specified by its AT attribute).
- Next and subsequent print structures are printed relative to the ending position of the previous print structure:
- If there is room to print the next structure beside the previous structure, it is printed there.
- If not, it is printed below the previous.

The values contained in the *x*, *y*, *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer's default font.

Example:

```
CustRpt     REPORT,AT(1000,2000,6500,7000),THOUS      !1" margins all around
              HEADER,AT(1000,1000,6500,1000)          !Page relative position
                !structure elements                   !1" band across top of page
              END
CustDetail1   DETAIL,AT(0,0,6500,1000)                !Detail relative position
                !structure elements                   !1" band across page
              END
CustDetail2   DETAIL,ABSOLUTE,AT(1000,8000,6500,1000)   !Page relative position
                !structure elements                     !1" band near page bottom
              END
              FOOTER,AT(1000,9000,6500,1000)          !Page relative position
                !structure elements                   !1" band across page bottom
              END
            END
```

FONT (set print structure default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default print font.
typeface	A string constant containing the name of the font. If omitted, the printer's default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the printer's default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on FORM, DETAIL, HEADER, and FOOTER structures specifies the default print font for all controls in the structures that do not have a FONT attribute.

The *typeface* may name any font registered in the Windows system which the printer driver supports. This includes the TrueType fonts for most printers. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
CustRpt   REPORT,FONT('Arial',12)             !Default font: 12 point Arial
            HEADER,FONT('Arial',18,,FONT:bold) !18 point bold Arial for the header
               !structure elements
            END
CustDetail1 DETAIL                             !Detail uses the default font
               !structure elements
            END
            FOOTER,FONT('Arial',12,00FF0000h)  !12 point Red Arial for the footer
                !structure elements
            END
          END
```

PAGEAFTER (set page break after)

PAGEAFTER([*newpage*] **)**

PAGEAFTER Specifies the structure is printed, then initiates page overflow.

newpage An integer constant or constant expression that specifies the page number to print on the next page. If omitted, the current page number is incremented during page overflow.

The **PAGEAFTER** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), initiates page overflow after it is printed. This means that the print structure on which the PAGEAFTER attribute is present is printed, followed by the page FOOTER, and then the FORM and page HEADER.

The *newpage* parameter, if present, resets automatic page numbering at the number specified.

Example:

```
CustRpt   REPORT
            HEADER
              !structure elements
            END
Break1      BREAK(SomeVariable)
              HEADER
                !structure elements
              END
CustDetail    DETAIL
                !structure elements
              END
              FOOTER,PAGEAFTER           !Group Footer, initiates page overflow
                !structure elements
              END
            END
            FOOTER
              !structure elements
            END
          END
```

PAGEBEFORE (set page break first)

PAGEBEFORE([*newpage*] **)**

PAGEBEFORE	Specifies the structure is printed on a new page, after page overflow.
newpage	An integer constant or constant expression that specifies the page number to print on the new page. If omitted, the current page number is incremented during page overflow.

The **PAGEBEFORE** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is printed on a new page, after page overflow. This means that first, the page FOOTER is printed, then the FORM and page HEADER. The print structure on which the PAGEBEFORE attribute is present is printed only after these page overflow actions are complete.

The *newpage* parameter, if present, resets automatic page numbering at the number specified.

Example:

```
CustRpt   REPORT
            HEADER
              !structure elements
            END
Break1      BREAK(SomeVariable)
              HEADER,PAGERBEFORE       !Group Header, initiates page overflow
                !structure elements
              END
CustDetail    DETAIL
                !structure elements
              END
              FOOTER
                !structure elements
              END
            END
            FOOTER
              !structure elements
            END
          END
```

USE (set structure equate label)

USE(*label* [,*number*] **)**

USE	Specifies a field equate label for the structure.
label	A field equate label to reference the structure in executable code.
number	An integer constant that specifies the number the compiler equates to the field equate label for the structure.

The **USE** attribute on a FORM, DETAIL, HEADER, or FOOTER structure specifies a field equate label for the structure. This provides a mechanism for executable source code statements to reference the structure.

The print structures in a REPORT are treated just as controls in a WINDOW; they are automatically assigned positive numbers by the compiler.

The USE attribute's *number* parameter allows you to specify the actual field number the compiler assigns to the structure. This *number* also is used as the new starting point for subsequent field equate numbering for all structures and controls without a *number* parameter in their USE attribute. Subsequent structures or controls without a *number* parameter in their USE attribute are incremented (or decremented) relative to the last *number* assigned.

Example:

```
BuildRpt  PROCEDURE
CustRpt   REPORT
            HEADER,USE(?PageHeader)     !Page header
              !structure elements
            END
CustDetail  DETAIL,USE(?Detail)         !Line item detail
              !structure elements
            END         !
            FOOTER,USE(?PageFooter)     !Page footer
              !structure elements
            END
          END
  CODE
  PrintRpt(CustRpt,?Detail)          !Pass report and detail equate to print proc

PrintRpt  PROCEDURE(RptToPrint,DetailNumber)
  CODE
  OPEN(RptToPrint)                   !Open passed report
  PRINT(RptToPrint,DetailNumber)     !Print its detail
  CLOSE(RptToPrint)                  !Close passed report
```

WITHNEXT (set widow elimination)

WITHNEXT([*siblings*] **)**

WITHNEXT	Specifies the structure is always printed on the same page as print structures PRINTed immediately following it.
siblings	An integer constant or constant expression that specifies the number of following print structures to print on the same page. If omitted, the default value is one.

The **WITHNEXT** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is always printed on the same page as the specified number of print structures PRINTed immediately following it. This ensures that the structure is never printed on a page by itself, eliminating “widow” print structures. A “widow” print structure is defined as a group header, or first detail item in a related group of items, printed on the preceding page, separated from the rest of its related items.

The *siblings* parameter, if present, sets the number of following print structures that must be printed on the same page with the structure. To be counted, the following print structures must come from the same, or nested, BREAK structures. They must be related items. Any print structures not within the same, or nested, BREAK structures are printed but not counted as part of the required number of *siblings*.

Example:

```
CustRpt   REPORT
Break1      BREAK(SomeVariable)
              HEADER,WITHNEXT(2)        !Always print with 2 siblings
                !structure elements
              END
CustDetail    DETAIL,WITHNEXT()         !Always print with 1 sibling
                !structure elements
              END
              FOOTER
                !structure elements
              END
            END
          END
```

WITHPRIOR (set orphan elimination)

WITHPRIOR([*siblings*] **)**

WITHPRIOR Specifies the structure is always printed on the same page as print structures PRINTed immediately preceding it.

siblings An integer constant or constant expression that specifies the number of preceding print structures to print on the same page. If omitted, the default value is one.

The **WITHPRIOR** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is always printed on the same page as the specified number of print structures PRINTed immediately preceding it. This ensures that the structure is never printed on a page by itself, eliminating "orphan" print structures. An "orphan" print structure is defined as a group footer, or last detail item in a related group of items, that is printed on the following page separated from the rest of its related items.

The *siblings* parameter, if present, sets the number of preceding print structures that must be printed on the same page with the structure. To be counted, the preceding print structures must come from the same, or nested, BREAK structures. They must be related items. Any print structures not within the same, or nested, BREAK structures are printed, but not counted as part of the required number of *siblings*.

Example:

```
CustRpt   REPORT
Break1      BREAK(SomeVariable)
              HEADER
               !structure elements
              END
CustDetail    DETAIL,WITHPRIOR()           !Always print with 1 sibling
               !structure elements
              END
              FOOTER,WITHPRIOR(2)          !Always print with 2 siblings
               !structure elements
              END
            END
          END
```

Report Controls

BOX (declare a report box control)

BOX ,**AT( )** [,**USE( )**] [,**COLOR( )**] [,**FILL( )**] [,**ROUND**] [,**HIDE**]

BOX	Places a rectangular box in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
COLOR	Specifies the color for the border of the control. If omitted, the border is black.
FILL	Specifies the fill color for the control. If omitted, the box is not filled with color.
ROUND	Specifies the box corners are rounded. If omitted, the corners are square.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **BOX** control places a rectangular box in the REPORT at the position and size specified by its AT attribute, relative to the top left corner of the print structure containing the BOX.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              BOX,AT(0,0,20,20),USE(?B1)          !Unfilled, black border
              BOX,AT(20,20,20,20),ROUND           !Unfilled, rounded, black border
              BOX,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                                  !Filled, black border
              BOX,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
                                                !Unfilled, active border color border
            END
          END
```

CHECK (declare a report checkbox control)

CHECK(*text***) ,AT( ) [,USE( )] [,FONT( )] [,HIDE] [,DISABLE] [, | LEFT |]**
| RIGHT |

CHECK	Places a check box in the REPORT.
text	A string constant containing the text to display for the check box.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	The label of a numeric variable containing the value of the check box, zero (0 = OFF) or one (1 = ON).
FONT	Specifies the display font for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
DISABLE	Specifies the control appears dimmed in the REPORT.
LEFT	Specifies that the text appears to the left of the check box.
RIGHT	Specifies that the text appears to the right of the check box (the default position).

The **CHECK** control places a check box in the REPORT at the position and size specified by its AT attribute, relative to the top left corner of the print structure containing the CHECK.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              CHECK('1'),AT(0,0,20,20),USE(C1)
              CHECK('2'),AT(20,80,20,20),USE(C2),LEFT
              CHECK('3'),AT(0,100,20,20),USE(C3),FONT('Arial',12)
            END
          END
```

CUSTOM (declare a report .VBX custom control)

CUSTOM(*text***) ,AT( ) [,CLASS( )] [,USE( )] [,DISABLE] [,FONT( )] [,META]**
[,*property(value)***]**

CUSTOM	Places a Visual Basic .VBX control on the REPORT.
text	A string constant containing the title for the control.
AT	Specifies the size and location of the control. If omitted, default values are selected by the library.
CLASS	Specifies the .VBX filename and type of control.
USE	The label of a variable to supply the value of the control.
DISABLE	Specifies the control appears dimmed in the REPORT.
FONT	Specifies the display font for the control.
META	Specifies printing as a Windows metafile (.WMF).
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
property	A string constant containing the name of a custom property setting for the control.
value	A string constant containing the property value number or EQUATE for the *property*.

The **CUSTOM** control places a Visual Basic .VBX control in the report at the position and size specified by its AT attribute.

The *property* attribute allows you to specify any additional property settings the .VBX control may require. These are properties that need to be set for the .VBX control to properly function, and are not standard Clarion properties (such as AT or USE). The custom control should only receive values for these properties that are defined for that control. Valid properties and *values* for those properties would be defined in the custom control's documentation. You may have multiple *property* attributes on a single CUSTOM control.

Example:

```
Report    REPORT
DetailOne   DETAIL
              CUSTOM,AT(0,0,120,320),CLASS('graph.vbx','graph'),'graphstyle'('2')
            END
          END
```

ELLIPSE (declare a report ellipse control)

ELLIPSE ,**AT( )** [,**USE( )**] [,**COLOR( )**] [,**FILL( )**] [,**HIDE**]

ELLIPSE	Places a "circular" figure in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
COLOR	Specifies the color for the border of the ellipse. If omitted, the ellipse has a black border.
FILL	Specifies the fill color for the control. If omitted, the ellipse is not filled with color.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **ELLIPSE** control places a "circular" figure in the REPORT at the position and size specified by its AT attribute. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters of its AT attribute. The *x* and *y* parameters specify the starting point, relative to the top left corner of the print structure containing it, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              ELLIPSE,AT(0,0,20,20)              !Unfilled, black border
              ELLIPSE,AT(0,20,20,20),USE(?Ellipse1),DISABLE
                                                 !Unfilled, black border, dimmed
              ELLIPSE,AT(20,20,20,20),ROUND      !Unfilled, rounded, black border
              ELLIPSE,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                                 !Filled, black border
              ELLIPSE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
                                              !Unfilled, active border color border
             END
            END
```

GROUP (declare a group of report controls)

```
GROUP(text) ,AT( ) [,USE( )] [,FONT( )] [,BOXED] [,HIDE]
        controls
END
```

GROUP	Declares a group of controls that may be referenced as one entity.
text	A string constant containing the prompt for the group of controls. The *text* is printed only if the BOXED attribute is also present.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
FONT	Specifies the display font for the control and the default for all the controls in the GROUP.
BOXED	Specifies a single-track border around the group of controls with the text at the top of the border.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
controls	Control declarations that may be referenced as the GROUP.

The **GROUP** control declares a group of controls that may be referenced as one entity. This control allows you to design reports that look the same on paper as on the screen.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              GROUP('Group 1'),USE(!G1),AT(80,0,20,20),BOXED
                STRING(@S8),AT(80,0,20,20),USE(E5)
                STRING(@S8),AT(100,0,20,20),USE(E6)
              END
              GROUP('Group 2'),USE(?G2),FONT('Arial',12)
                STRING(@S8),AT(120,0,20,20),USE(E7)
                STRING(@S8),AT(140,0,20,20),USE(E8)
              END
            END
```

IMAGE (declare a report graphic image control)

IMAGE(*file***) ,AT( ) [,USE( )] [,HIDE]**

IMAGE	Places a graphic image on the REPORT.
file	A string constant containing the name of the file to print. The file is linked into the .EXE as a resource.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **IMAGE** control places a graphic image on the REPORT at the position and size specified by its AT attribute. This may be a bitmap (.BMP), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or windows metafile (.WMF). This may not be an icon (.ICO) because Windows does not support printing icons.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              IMAGE('PIC.BMP'),AT(0,0,20,20),USE(?I1)
              IMAGE('PIC.WMF'),AT(40,0,20,20),USE(?I2)
              IMAGE('PIC.JPG'),AT(60,0,20,20),USE(?I3)
            END
          END
```

LINE (declare a report line control)

LINE ,**AT( )** [,**USE( )**] [,**COLOR( )**] [,**HIDE**]

LINE	Places a straight line in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
COLOR	Specifies the color for the line. If omitted, the color is black.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **LINE** control places a straight line in the REPORT at the position and size specified by its AT attribute.

The *x* and *y* parameters of the AT attribute specify the starting point of the line. The *width* and *height* parameters of the AT attribute specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* parameter is negative, the line slopes left; if the *height* parameter is negative, the line slopes left. If either the *width* or *height* parameter is zero, the line is horizontal or vertical.

Width	Height	Result
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              LINE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER) !Border color
              LINE,AT(480,180,20,20),USE(?L2)
            END
          END
```

LIST (declare a report list control)

```
LIST ,FROM( ) ,AT( ) [,FONT( )] [,USE( )] [,HIDE] [,  | FORMAT( )   | ]
                                                      | LEFT        |
                                                      | RIGHT       |
                                                      | CENTER      |
                                                      | DECIMAL     |
```

LIST	Places the current item of a list of data items in the REPORT.
FROM	Specifies the origin of the data displayed in the list.
AT	Specifies the size and location of the control. If omitted, the runtime library chooses a value.
FONT	Specifies the display font for the control.
USE	Specifies a field equate label for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
FORMAT	Specifies the print format of the data.
LEFT	Specifies that the data is left justified within the LIST.
RIGHT	Specifies that the data is right justified within the LIST.
CENTER	Specifies that the data is centered within the LIST.
DECIMAL	Specifies that the data is aligned on the decimal point within the LIST.

The **LIST** control places the current item of a list of data items in the REPORT at the position and size specified by its AT attribute. LIST is valid only in a DETAIL structure. Its purpose is to allow the report format to duplicate the screen appearance of the LIST's FORMAT setting. When the first instance of the DETAIL structure containing the LIST is printed, any headers in the FORMAT attribute are printed along with the current FROM attribute entry. When the last DETAIL structure containing the LIST is printed, the LIST footers are printed along with the current FROM attribute entry.

Example:

```
Q          QUEUE
F1           STRING(1)
F2           STRING(4)
           END

CustRpt    REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              LIST,AT(80,0,20,20),USE(?L1),FROM(Q),FORMAT('5C~List~15L~Box~')
            END
           END
```

OPTION (declare a group of report RADIO controls)

```
OPTION(text) ,AT( ) [,USE( )] [,BOXED]  [,HIDE]
  radios
END
```

OPTION	Prints a group of RADIO controls.
text	A string constant containing the prompt for the group of controls. The *text* is printed only if the BOXED attribute is also present.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	The label of a string variable containing the value of the RADIO selected by the user.
BOXED	Specifies a single-track border around the RADIO controls with the text at the top of the border.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
radios	Multiple RADIO control declarations.

The **OPTION** control prints a group of RADIO controls that display a list of choices. The multiple RADIO controls in the OPTION structure define the choices. The selected choice is identified by a filled RADIO button.

No RADIO button selected is a valid option. This occurs only when the OPTION structure's USE variable does not contain a value duplicated in a RADIO *text* parameter.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              OPTION('Option'),USE(OptVar),AT(80,0,20,20),BOXED
                RADIO('Radio 1'),AT(80,0,20,20),USE(?R1)
                RADIO('Radio 2'),AT(100,0,20,20),USE(?R2)
               END
             END
           END
```

RADIO (declare a report radio button control)

RADIO(*text***) ,AT( ) [,FONT( )] [, | LEFT |] [,USE( )] [,HIDE]**
| RIGHT |

RADIO	Places a radio button in the REPORT.
text	A string constant containing the text to display for the radio button.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
FONT	Specifies the display font for the control.
LEFT	Specifies that the text appears to the left of the radio button.
RIGHT	Specifies that the text appears to the right of the radio button (the default position).
USE	Specifies a field equate label for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **RADIO** control places a radio button in the REPORT at the position and size specified by its AT attribute. A RADIO control may only be placed within an OPTION control. The RADIO selected by the user (the value in the OPTION's USE variable) is displayed as a filled RADIO button.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              OPTION('Option'),USE(OptVar),AT(80,0,20,20),BOXED
                RADIO('Radio 1'),AT(80,0,20,20),USE(?R1)
                RADIO('Radio 2'),AT(100,0,20,20),USE(?R2)
                RADIO('Radio 3'),AT(100,0,20,20),USE(?R2),LEFT
              END
            END
          END
```

STRING (declare a report string control)

```
STRING(text),AT( ) [,FONT( )] [,HIDE] [,TRN] [,USE( )]
        [, | LEFT    | ] [,  | PAGENO                    | ]
           | RIGHT   |       | CNT   [, RESET( ) / PAGE ] |
           | CENTER  |       | SUM   [, RESET( ) / PAGE ] |
           | DECIMAL |       | AVE   [, RESET( ) / PAGE ] |
                             | MIN   [, RESET( ) / PAGE ] |
                             | MAX   [, RESET( ) / PAGE ] |
```

STRING	Places the *text* in the REPORT.
text	A string constant containing the text to display, or a display picture token to format the variable specified in the USE attribute.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
FONT	Specifies the font used to display the text.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
TRN	Specifies the text or USE variable characters transparently print over the background.
USE	Specifies a variable whose contents are printed in the format of the picture token declared instead of string text.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.
DECIMAL	Specifies that the text is aligned on the decimal point within the area specified by the AT attribute.
PAGENO	Specifies the current page number is printed in the format of the picture token declared instead of string text.
CNT	Specifies the number of details printed is printed in the format of the picture token declared instead of string text.
SUM	Specifies the sum of the USE variable is printed in the format of the picture token declared instead of string text.
AVE	Specifies the average value of the USE variable is printed in the format of the picture token declared

instead of string text.

MIN Specifies the mimimum value of the USE variable is printed in the format of the picture token declared instead of string text.

MAX Specifies the maximum value of the USE variable is printed in the format of the picture token declared instead of string text.

RESET Specifies the CNT, SUM, AVE, MIN, or MAX is reset when the specified group break occurs.

PAGE Specifies the CNT, SUM, AVE, MIN, or MAX is reset to zero when the page break occurs.

The **STRING** control places the *text* in the REPORT at the position and size specified by its AT attribute. If the *text* parameter is a picture token instead of a string constant or variable, the contents of the variable named in the USE attribute are formatted to that display picture, at the position and size specified by the AT attribute.

A STRING with the TRN attribute prints characters transparently, without obliterating the background. This means only the dots required to create each character are printed. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1    BREAK(Pre:Key1)
            HEADER,AT(0,0,6500,1000)
              STRING('Group  Head'),AT(3000,500,1500,500),FONT('Arial',18)
            END
Detail      DETAIL,AT(0,0,6500,1000)
              STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1)
            END
            FOOTER,AT(0,0,6500,1000)
              STRING('Group  Total:'),AT(5500,500,1500,500)
              STRING(@N$11.2'),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Pre:Key1)
            END
          END
        END
```

TEXT (declare a multi-line text control)

```
TEXT ,AT( ) [,USE( )] ,FONT( )] [,  | CAP | ] [,   | LEFT    | ]  [,HIDE]
                                    | UPR |        | RIGHT   |
                                                   | CENTER  |
```

TEXT	Places a multi-line print field in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	The label of the variable that contains the value to print.
FONT	Specifies the display font for the control.
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized).
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **TEXT** control places a multi-line print field in the REPORT at the position and size specified by its AT attribute. The variable specified in the USE attribute contains the data to print.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail      DETAIL,AT(0,0,6500,1000)
              TEXT,AT(0,0,40,40),USE(E1)
              TEXT,AT(100,0,40,40),USE(E6),FONT('Arial',12)
              TEXT,AT(120,0,40,40),USE(E7),CAP
              TEXT,AT(140,0,40,40),USE(E8),UPR
              TEXT,AT(160,0,40,40),USE(E9),LEFT
              TEXT,AT(180,0,40,40),USE(E10),RIGHT
              TEXT,AT(200,0,40,40),USE(E11),CENTER
            END
          END
```

Control Attributes

AT (set control position and size in report)

AT([*x*] [,*y*] [,*width*] [,*height*]**)**

AT	Defines the position and size of a control.
x	An integer constant or constant expression that specifies the initial horizontal position of the top left corner of the control, relative to the top left corner of the print structure containing it. If omitted, the runtime library provides a default value.
y	An integer constant or constant expression that specifies the initial vertical position of the top left corner of the control, relative to the top left corner of the print structure containing it. If omitted, the runtime library provides a default value.
width	An integer constant or constant expression that specifies the width of the control. If omitted, the runtime library provides a default value.
height	An integer constant or constant expression that specifies the height of the control. If omitted, the runtime library provides a default value.

The **AT** attribute defines the position and size of a control, relative to the top left corner of the print structure containing it. If any parameter is omitted, the runtime library provides a default value.

The values contained in the *x*, *y*, *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the REPORT, or the printer's default font.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS    !AT specifies detail print area
Detail      DETAIL,AT(0,0,6500,1000)              !AT specifies band size and
                                                  ! relative position offset from
                                                  ! last printed detail
              STRING('String  Constant'),AT(500,500,1500,500)
                                                  !AT specifies control size and
                                                  ! offset within the detail band
            END
          END
```

AVE (set total average)

AVE

The **AVE** attribute specifies the average (arithmetic mean) of the STRING controls' USE variable is printed.

- An AVE field in a DETAIL structure is calculated each time the DETAIL structure containing the control is PRINTed.
- An AVE field in a group FOOTER structure is calculated each time any DETAIL structure in the BREAK structure containing the control is PRINTed.
- An AVE field in a page FOOTER structure is calculated each time any DETAIL structure in any BREAK structure is PRINTed.
- An AVE field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The average is reset only if the RESET or PAGE attribute is also specified. The STRING control using this attribute would usually be placed in a group or page FOOTER.

BOXED (set report controls group border)

BOXED

The **BOXED** attribute specifies a single-track border around a GROUP or OPTION structure. The *text* parameter of the GROUP or OPTION control appears in a gap at the top of the border box. If BOXED is omitted, the *text* parameter of the GROUP or OPTION control is not printed.

CAP, UPR (set print case)

CAP
UPR

The **CAP** and **UPR** attributes specify the automatic case of text printed in a TEXT control. UPR specifies all upper case; CAP specifies "Proper Name Capitalization," where the first letter of each word is capitalized and all other letters are lower case.

CNT (set total count)

CNT

The **CNT** attribute specifies an automatic count of the number of times DETAIL structures have been printed.

- A CNT field in a DETAIL structure is incremented each time the DETAIL structure containing the control is PRINTed. This provides a "running" count.
- A CNT field in a group FOOTER structure is incremented each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides a total of the number of DETAIL structures printed in the group.
- A CNT field in a page FOOTER structure is incremented each time any DETAIL structure in any BREAK structure is PRINTed. This provides a total of the number of DETAIL structures printed on the page (or report).
- A CNT field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The CNT is reset only if the RESET or PAGE attribute is also specified.

COLOR (set color)

COLOR(*rgb*)

COLOR	Specifies the print color of a BOX, LINE, or ELLIPSE control.
rgb	A LONG or ULONG integer constant containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2), or an EQUATE for a standard Windows color value.

The **COLOR** attribute specifies the print color of a BOX, LINE, or ELLIPSE control. On a BOX or ELLIPSE, the color specified is the color used for the border. EQUATEs for Windows' standard colors are contained in the EQUATES.CLW file.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              ELLIPSE,AT(60,60,200,200),COLOR(COLOR:ACTIVEBORDER) !Color  EQUATE
              BOX,AT(360,60,200,200),COLOR(00FF0000h) !Pure Red
            END
          END
```

FILL (set print fill color)

FILL(*rgb*)

FILL	Specifies the print fill color of a BOX or ELLIPSE control.
rgb	A LONG or ULONG integer constant containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **FILL** attribute specifies the print fill color of a BOX or ELLIPSE control. If omitted, the control is not filled with color.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              ELLIPSE,AT(60,60,200,200),FILL(COLOR:ACTIVEBORDER)
                                                    !Color EQUATE
              BOX,AT(360,60,200,200),FILL(00FF0000h)    !Pure Red
            END
          END
```

FONT (set default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the print font for the control.
typeface	A string constant containing the name of the font. If omitted, the print structure's FONT attribute is used (if present), or the REPORT structure's FONT attribute is used (if present), or else the printer's default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the printer's default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant, constant expression, or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute specifies the print font for the control, overriding any FONT specified on the REPORT or print structure.

The *typeface* may name any font registered in the Windows system which the printer driver supports. This includes the TrueType fonts for most printers. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              STRING('Text'),AT(0,0),FONT('Arial',14,00FF0000h)
              STRING('Text'),AT(160,160),FONT('Arial',12,,FONT:italic)
            END
          END
```

FORMAT (set LIST print format)

FORMAT(*format string***)**

FORMAT Specifies the print format for the data.

format string A string constant specifying the column or multi-column print format.

The **FORMAT** attribute specifies the print format for the data in the LIST control. The *format string* contains the information for single or multi-column formatting of the data.

The *format string* contains "field-specifiers" which map to the fields of the QUEUE. Multiple "field-specifiers" may be grouped together as a "field-group" in square brackets (**[ ]**) to display as a single unit.

Only the fields in the QUEUE for which there are "field-specifiers" are printed. This means that, if there are two fields specified in the *format string* and three fields in the QUEUE, only the two specified in the *format string* are printed in the LIST control.

Field-specifier" format: *width justification* [**(***indent***)**] [*modifiers*]

width A required integer defining the width of the field. Specified in dialog units unless overridden by the THOUS, MM, or POINTS attribute.

justification A single capital letter (**L** , **R** , **C** , or **D**) that specifies **L**eft, **R**ight, **C**enter, or **D**ecimal justification. One is required.

indent An optional integer, enclosed in parentheses, that specifies the indent from the justification. This may be negative. With left (**L**) justification, *indent* defines a left margin; with right (**R**) or decimal (**D**), it defines a right margin; and with center (**C**), it defines an indent from the center of the field.

modifiers: Optional special characters (listed below) to modify the print format of the field or group. Multiple *modifiers* may be used on one field or group.

~header~ [justification [(indent)]]
A header string enclosed in tildes, followed by optional justification and/or indent, prints the header at the top of the list. The header uses the same justification and indent as the field, if not specifically overidden.

@picture@ The *picture* formats the field for printing. The trailing @ is required to define the end of the *picture*, so that display pictures like @N12~Kr~ can be used in the format string without creating ambiguity.

#number#	The *number* enclosed in pound signs (#) indicates the QUEUE field to print. Following fields in the format string without an explicit *#number#* are taken in order from the fields following the *#number#* field. For example, #2# on the first field in the format string indicates starting with the second field in the QUEUE, skipping the first. If the number of fields specified in the format string are >= the number of fields in the QUEUE, the format "wraps around" to the start of the QUEUE.
_	An underscore underlines the field.
/	A slash causes the next field to appear on a new line (only used on a field within a group).
|	A vertical bar places a vertical line to the right of the field.

"Field-group" format: **[** *multiple field-specifiers* **]** [**(***size***)**] [*modifiers*]

multiple field-specifiers	A list of field-specifiers contained in square brackets (**[ ]**) that cause them to be treated as a single display unit.
size	An optional integer, enclosed in parentheses, that specifies the default width of the group. If omitted, the size is calculated from the enclosed fields.
modifiers	The "field-group" *modifiers* act on the entire group of fields. These are the same *modifiers* listed above.

Example:

```
TD        QUEUE,AUTO
FName       STRING(20)
LName       STRING(20)
Init        STRING(4)
Wage        REAL
          END
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              LIST,AT(0,34,366,146),FORMAT(''),FROM(TD),USE(?Show)
            END
          END
  CODE
  OPEN(CustRpt)
  SETTARGET(CustRpt)
  IF SomeCondition
    ?Show{PROP:format} = '80C~First Name~80C~Last Name~16L~Intls~60R~Wage~|'
  ELSE
    ?Show{PROP:format} = '80C~First Name~80C~Last Name~16L~Intls~60D(10)~Wage~|'
  END
```

FROM (set report listbox data source)

FROM(*source*)

FROM	Specifies the source of the data printed in a LIST control.
source	The label of a QUEUE, or any variable (normally a GROUP) containing the data items to print in the LIST.

The **FROM** attribute specifies the source of the data elements printed in a LIST control. The data elements are formatted for display according to the information in the FORMAT attribute.

If the label of a QUEUE is specified as the *source*, all fields in the QUEUE are printed. If the label of one field in a QUEUE is specified as the *source*, only that field is printed. Only the current QUEUE entry in the queue's data buffer is printed in the LIST.

If a string constant or variable is specified as the *source*, the entire string is printed in the LIST.

Example:

```
TD          QUEUE,AUTO
FName         STRING(20)
LName         STRING(20)
Init          STRING(4)
Wage          REAL
            END
CustRpt     REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail    DETAIL,AT(0,0,6500,1000)
                LIST,AT(0,34,366,146),FORMAT('80L80L16L60L'),FROM(TD),USE(?Show1)
                LIST,AT(0,200,100,146),FORMAT('80L'),FROM(Fname),USE(?Show2)
              END
            END
```

HIDE (set control non-print)

HIDE

The **HIDE** attribute specifies the control does not print unless the UNHIDE statement is used to allow it to print.

LEFT, RIGHT, CENTER, DECIMAL (set print justification)

LEFT([*indent*] **)**
RIGHT([*indent*] **)**
CENTER([*indent*] **)**
DECIMAL([*indent*] **)**

indent An integer constant specifying the amount of margin left after justification. This is in dialog units unless overridden by the THOUS, MM, or POINTS attribute.

The **LEFT**, **RIGHT**, **CENTER**, and **DECIMAL** attributes specify the justification of data printed. LEFT specifies left justification, RIGHT specifies right justification, CENTER specifies centered text, and DECIMAL specifies numeric data aligned on the decimal point.

The *indent* parameter on the CENTER attribute specifies an offset from the center. On the DECIMAL attribute, *indent* specifies the position of the decimal point.

The following controls allow LEFT or RIGHT only (without an *indent* parameter):

CHECK
GROUP
OPTION
RADIO

The following controls allow LEFT(*indent*), RIGHT(*indent*), CENTER(*indent*), or DECIMAL(*indent*):

LIST
STRING

The TEXT control allows LEFT, RIGHT, and CENTER (without an *indent* parameter).

Example:

```
Rpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail  DETAIL,AT(0,0,6500,1000)
          LIST,AT(0,20,100,146),FORMAT('800L'),FROM(Fname),USE(?Show2),LEFT(100)
        END
      END
```

MAX (set total maximum)

MAX

The **MAX** attribute specifies printing the maximum value the STRING control's USE variable has contained so far.

- A MAX field in a DETAIL structure is evaluated each time the DETAIL structure containing the control is PRINTed. This provides a "running" maximum value.
- A MAX field in a group FOOTER structure is evaluated each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides the maximum value of the variable in the group.
- A MAX field in a page FOOTER structure is evaluated each time any DETAIL structure in any BREAK structure is PRINTed. This is the maximum value of the variable in the page (or report to date).
- A MAX field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The MAX value is reset only if the RESET or PAGE attribute is also specified.

META (set .VBX to print as .WMF)

META

The **META** attribute specifies printing a .VBX custom control as a .WMF windows metafile. This will print the control as a graphic image on the report.

MIN (set total minimum)

MIN

The **MIN** attribute specifies printing the minimum value the STRING control's USE variable has contained so far.

- A MIN field in a DETAIL structure is evaluated each time the DETAIL structure containing the control is PRINTed. This provides a "running" minimum value.
- A MIN field in a group FOOTER structure is evaluated each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides the minimum value of the variable in the group.
- A MIN field in a page FOOTER structure is evaluated each time any DETAIL structure in any BREAK structure is PRINTed. This is the minimum value of the variable in the page (or report to date).
- A MIN field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The MIN value is reset only if the RESET or PAGE attribute is also specified.

PAGE (set page total reset)

PAGE

The **PAGE** attribute specifies the CNT, SUM, AVE, MIN, or MAX is reset to zero (0) when page break occurs.

PAGENO (set page number print)

PAGENO

The **PAGENO** attribute specifies the STRING control prints the current page number.

RESET (set total reset)

RESET(*breaklevel***)**

RESET Resets the CNT, SUM, AVE, MIN, or MAX to zero (0).

breaklevel The label of a BREAK structure.

The **RESET** attribute specifies the group break at which the CNT, SUM, AVE, MIN, or MAX is reset to zero (0).

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
Break1    BREAK(Pre:Key1)
            HEADER,AT(0,0,6500,1000)
              STRING('Group  Head'),AT(3000,500,1500,500),FONT('Arial',18)
            END
Detail      DETAIL,AT(0,0,6500,1000)
              STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1)
            END
            FOOTER,AT(0,0,6500,1000)
              STRING('Group  Total:'),AT(5500,500,1500,500)
              STRING(@N$11.2'),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Break1)
            END
          END
        END
```

ROUND (set round-cornered report BOX)

ROUND

The **ROUND** attribute specifies a BOX control with rounded corners.

SUM (set total)

SUM

The **SUM** attribute specifies printing the sum of the values contained in the STRING control's USE variable.

- A SUM field in a DETAIL structure is incremented each time the DETAIL structure containing the control is PRINTed. This provides a "running" total.
- A SUM field in a group FOOTER structure is incremented each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides the sum of the value contained in the variable in the group.
- A SUM field in a page FOOTER structure is incremented each time any DETAIL structure in any BREAK structure is PRINTed. This is the sum of the values contained in the variable in the page.
- A SUM field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The SUM value is reset only if the RESET or PAGE attribute is also specified.

TRN (set transparent report string)

TRN

The **TRN** attribute on a STRING control specifies the characters print transparently, without obliterating the background over which the STRING is placed. Only the dots required to create each character are printed. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

Example:

```
CustRpt REPORT,AT(1000,1000,6500,9000),THOUS
          FORM,AT(0,0,6500,9000)
            IMAGE('PIC.BMP'),USE(?I1)AT(0,0,6500,9000)          !Full page image
            STRING('String  Constant'),AT(10,0,20,20),USE(?S1),TRN
                                                  !Transparent string on the image
          END
        END
```

USE (set code reference name)

```
USE( | label    | [,number] [,equate] )
     | variable |
```

USE	Specifies a variable or field equate label for the control.
label	A field equate label to reference the control in executable code.
variable	The variable containing the value to print in the control. This label (with a ? prepended) becomes the field equate label for the control, unless the *equate* parameter is used.
number	An integer constant that specifies the number the compiler equates to the field equate label for the control.
equate	A field equate label to reference the control in executable code when the named *variable* has already been used in the same structure. This provides a mechanism to provide a unique field equate when the *variable* would not.

The **USE** attribute specifies a variable or field equate label for the control. USE with a *label* parameter simply provides a mechanism for executable source code statements to reference the control. Some controls only allow a field equate *label* as the USE parameter, not a *variable*. These controls are: IMAGE, LINE, BOX, ELLIPSE, GROUP, and RADIO. USE with a *variable* parameter supplies the control with a variable to update by operator entry. This is applicable to an OPTION, TEXT, LIST, CHECK, or CUSTOM. STRING controls may use either a field equate label or variable

All controls in a REPORT are automatically assigned numbers by the compiler. These numbers start at one (1) and increment by one (1) for each control in the REPORT. The USE attribute's *number* parameter allows you to specify the actual field number the compiler assigns to the control. This *number* also is used as the new starting point for subsequent field numbering for fields without a *number* parameter in their USE attribute. Subsequent controls without a *number* parameter in their USE attribute are incremented relative to the last *number* assigned.

Two or more controls with the same USE variable in one REPORT structure would create the same field equate label for all. Therefore, when the compiler encounters this condition, all field equate labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference (unless you know the field numbers assigned by the compiler or assign a *number* of your own). You can deliberately create this condition to display the contents of the variable in multiple controls with different display pictures, or for totaling.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail      DETAIL,AT(0,0,6500,1000)
              STRING('Group  Total:'),AT(5500,500,1500,500),USE(?Constant)
                                                              !Field equate label
              STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1)
                                                              !USE variable
            END
          END
```

Report Procedures

CLOSE (close an active report structure)

CLOSE(*report***)**

CLOSE Deactivates a REPORT structure.

report The label of a REPORT structure.

CLOSE prints the last page FOOTER, (unless the last structure printed has the ALONE attribute), and closes the REPORT. If the REPORT has the PREVIEW attribute, all the temporary metafiles are deleted.

RETURN from a procedure in which a REPORT is opened automatically closes the REPORT.

Example:

```
CLOSE(CustRpt)       !Close the report
```

ENDPAGE (force page overflow)

ENDPAGE(*report*)

ENDPAGE Forces page overflow.

report The label of a REPORT structure.

The **ENDPAGE** statement initiates page overflow and flushes the print engine's print structure buffer. If the REPORT has the PREVIEW attribute, this has the effect of ensuring that the entire report is available to view.

Example:

```
SomeReport  PROCEDURE
WMFQue      QUEUE                       !Queue to contain .WMF filenames
              STRING(64)
            END
NextEntry   BYTE(1)                     !Queue entry counter variable
Report      REPORT,PREVIEW(WMFQue)      !Report with PREVIEW attribute
DetailOne     DETAIL
                !Report controls
            . .
ViewReport  WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(''),AT(0,0,320,180),USE(?ImageField)
              BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
              BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
              BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
            END
  CODE
  OPEN(Report)
  SET(SomeFile)                         !Code to generate the report
  LOOP
    NEXT(SomeFile)
    PRINT(DetailOne)
  END
  ENDPAGE(Report)                       !Flush the buffer
  OPEN(ViewReport)                      !Open report preview window
  GET(WMFQue,NextEntry)                 !Get first queue entry
  ?ImageField{PROP:text} = WMFQue       !Load first report page
  ACCEPT
    CASE ACCEPTED()
    OF ?NextPage
      NextEntry += 1                                !Increment entry counter
      IF NextEntry > RECORDS(WMFQue) THEN CYCLE.    !Check for end of report
      GET(WMFQue,NextEntry)                         !Get next queue entry
      ?ImageField{PROP:text} = WMFQue               !Load next report page
      DISPLAY                                       ! and display it
    OF ?PrintReport
      Report{PROP:flushpreview} = ON                !Flush files to printer
      BREAK                                         ! and exit procedure
    OF ?ExitReport
      BREAK                                         !Exit procedure
  . .
  CLOSE(ViewReport)             !Close window
  FREE(WMFQue)                  !Free the queue memory
  CLOSE(Report)                 !Close report (deleting all .WMF files)
  RETURN                        ! and return to caller
```

See Also: Page Overflow, PREVIEW

OPEN (open a report structure for processing)

OPEN(*report***)**

OPEN Activates a REPORT structure.

report The label of a REPORT structure.

OPEN activates a REPORT structure. You must OPEN a REPORT before any of the structures may be printed.

Example:

```
OPEN(CustRpt)       !Open the report
```

PRINT (print a report structure)

PRINT(| *structure* | **)**
| *report* ,*number* |

PRINT Prints a report DETAIL, HEADER, or FOOTER structure.

structure The label of a DETAIL structure.

report The label of a REPORT structure.

number The number or EQUATE label of a report structure to print (only valid with a *report* parameter).

The **PRINT** statement prints a report structure to the destination specified by the user in the Windows Print... dialog. PRINT automatically activates group breaks and page overflow as needed.

Example:

```
BuildRpt  PROCEDURE
CustRpt   REPORT
            HEADER,USE(?PageHeader)     !Page header
              !structure elements
            END
CustDetail  DETAIL,USE(?Detail)         !Line item detail
              !structure elements
            END         !
          END
  CODE
  PRINT(CustDetail)                     !Print order detail line
  PrintRpt(CustRpt,?PageHeader)         !Pass report and equate to print proc

PrintRpt  PROCEDURE(RptToPrint,DetailNumber)
  CODE
  PRINT(RptToPrint,DetailNumber)        !Print its structure
```

See Also: Page Overflow, BREAK

Graphics Overview

Contents

Clarion supplies the set of “graphics primitives” defined in this chapter to allow drawing in windows and reports.

Controls always appear on top of any graphics drawn to the window. This means the graphics appear to underly any controls in the window, so they don’t get in the way of the controls the user needs to access.

The Current Target

Graphics are always drawn to the “current target.” Unless overridden with SETTARGET, the “current target” is the last window opened (and not yet closed) on the current execution thread and is the window with input focus. Drawings in a window are persistent—redraws are handled automatically by the runtime library.

Graphics can also be drawn to a report. To do this, SETTARGET must be used to nominate the REPORT as the “current target.”

Every window or report has its own current pen width, color, and style. Therefore, to consistently use the same pen (which does not use the default settings) across multiple windows, the SETPENWIDTH, SETPENCOLOR, and SETPENSTYLE statements should be issued for each window.

Graphics Coordinates

The graphics coordinate system starts with the x,y coordinates (0,0) at the top left corner of the window. The coordinates are specified in dialog units (unless overridden by the THOUS, MM, or POINTS attributes when used on graphics placed in a REPORT). A dialog unit is defined as one-quarter the average character width and one-eighth the average character height of the font specified in the window’s FONT attribute (or the system font, if no FONT attribute is specified on the window).

Graphics drawn outside the currently visible portion of the window will appear if the window is scrolled. The size of the virtual screen over which the window may scroll automatically expands to include all graphics drawn to the window. Drawing graphics outside the visible portion of the window automatically causes the scroll bars to appear (if the window has the HSCROLL, VSCROLL, or HVSCROLL attribute).

Graphics Procedures

ARC (draw an arc of an ellipse)

ARC(*x* ,*y* ,*width* ,*height* ,*startangle* ,*endangle* **)**

ARC	Draws an arc of an ellipse on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
startangle	An integer expression that specifies the starting point of the arc, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
endangle	An integer expression that specifies the ending point of the arc, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.

The **ARC** procedure places an arc of an ellipse on the current window or report.

The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The *startangle* and *endangle* parameters specify what sector of the ellipse will be drawn, as an arc.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  ARC(100,50,100,50,0,900)    !Draw 90 degree arc from 3 to 12 o'clock, as
                              ! the top-right quadrant of ellipse
```

BLANK (erase graphics)

BLANK([*x*] [,*y*] [,*width*] [,*height*] **)**

BLANK	Erases all graphics written to the specified area of the current window or report.
x	An integer expression that specifies the horizontal position of the starting point. If omitted, the default is zero.
y	An integer expression that specifies the vertical position of the starting point. If omitted, the default is zero.
width	An integer expression that specifies the width. If omitted, the default is the width of the window.
height	An integer expression that specifies the height. If omitted, the default is the height of the window.

The **BLANK** procedure erases all graphics written to the specified area of the current window or report. Controls are not erased. BLANK with no parameters erases the entire window or report.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  ARC(100,50,100,50,0,900)      !Draw arc
  BLANK                         !Then erase it
```

BOX (draw a rectangle)

BOX(*x ,y ,width ,height* [,*fill*] **)**

BOX	Draws a rectangular box on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **BOX** procedure places a rectangular box on the current window or report. The position and size of the box are specified by *x*, *y*, *width*, and *height* parameters.

The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the box. The box extends to the right and down from its starting point.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  BOX(100,50,100,50,00FF0000h)    !Red box
```

CHORD (draw a section of an ellipse)

CHORD(*x* ,*y* ,*width* ,*height* ,*startangle* ,*endangle* [,*fill*] **)**

CHORD	Draws a closed sector of an ellipse on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
startangle	An integer expression that specifies the starting point of the chord, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
endangle	An integer expression that specifies the ending point of the chord, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o'clock.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **CHORD** procedure places a closed sector of an ellipse on the current window or report. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box." The *startangle* and *endangle* parameters specify what sector of the ellipse will be drawn, as an arc. The two end points of the arc are also connected with a straight line.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  CHORD(100,50,100,50,0,900,00FF0000h)    !Red 90 degree crescent
```

ELLIPSE (draw an ellipse)

ELLIPSE(*x* ,*y* ,*width* ,*height* [,*fill*] **)**

ELLIPSE	Draws an ellipse on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **ELLIPSE** procedure places an ellipse on the current window or report. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  ELLIPSE(100,50,100,50,00FF0000h)    !Red ellipse
```

IMAGE (draw a graphic image)

IMAGE(*x* ,*y* ,*width* ,*height* ,*filename* **)**

IMAGE	Places a graphic image on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width. This may be a negative number.
height	An integer expression that specifies the height. This may be a negative number.
filename	A string constant or variable containing the name of the file to display.

The **IMAGE** procedure places a graphic image on the current window or report at the position and size specified by its *x*, *y*, *width*, and *height* parameters. This may be a bitmap (.BMP), icon (.ICO), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or Windows metafile (.WMF).

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  IMAGE(100,50,100,50,'LOGO.BMP')   !Draw graphic image
```

LINE (draw a straight line)

LINE(*x* ,*y* ,*width* ,*height* **)**

LINE	Draws a straight line on the current window or report.
x	An integer expression specifying the horizontal position of the starting point.
y	An integer expression specifying the vertical position of the starting point.
width	An integer expression specifying the width. This may be a negative number.
height	An integer expression specifying the height. This may be a negative number.

The **LINE** procedure places a straight line on the current window or report. The starting position, slope, and length of the line are specified by *x, y, width,* and *height* parameters. The *x* and *y* parameters specify the starting point of the line. The *width* and *height* parameters specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* parameter is negative, the line slopes left; if the *height* parameter is negative, the line slopes left. If either the *width* or *height* parameter is zero, the line is horizontal or vertical.

Width	Height	Result
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

The line color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The width is the current width set by SETPENWIDTH; the default width is one pixel. The line's style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  LINE(100,50,100,50)       !Draw line
```

PIE (draw a pie chart)

PIE(*x* ,*y* ,*width* ,*height* ,*slices* ,*colors* [,*depth*] [,*wholevalue*] [,*startangle*] **)**

PIE	Draws a pie chart on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
slices	A SHORT array of values that specify the relative size of each slice of the pie.
colors	A LONG array that specifies the fill color for each slice.
depth	An integer expression that specifies the depth of the three-dimensional pie chart. If omitted, the chart is two-dimensional.
wholevalue	A numeric constant or variable that specifies the total value required to create a complete pie chart. If omitted, the sum of the *slices* array is used.
startangle	A numeric constant or variable that specifies the starting point of the first slice of the pie, measured as a fraction of the *wholevalue*. If omitted (or zero), the first slice starts at the twelve o'clock position.

The **PIE** procedure creates a pie chart on the current window or report. The pie (an ellipse) is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The slices of the pie are created clockwise from the *startangle* parameter as a fraction of the *wholevalue*. Supplying a *wholevalue* parameter that is greater than the sum of all the *slices* array elements creates a pie chart with a piece missing.

The color of the lines is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The width of the lines is the current width set by SETPENWIDTH; the default width is one pixel. The line style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END

SliceSize   SHORT,DIM(4)
SliceColor  LONG,DIM(4)

  CODE
  SliceSize[1] = 90
  SliceColor[1] = 0             !Black
  SliceSize[2] = 90
  SliceColor[2] = 00FF0000h     !Red
  SliceSize[3] = 90
  SliceColor[3] = 0000FF00h     !Green
  SliceSize[4] = 90
  SliceColor[4] = 000000FFh     !Blue
  OPEN(MDIChild)
  PIE(100,50,100,50,SliceSize,SliceColor)
                            !Draw pie chart containing
                            ! four equal slices, starting at 12 o'clock
                            ! drawn counter-clockwise - Black, Red, Green, and Blue
```

POLYGON (draw a multi-sided figure)

POLYGON(*array* [,*fill*] **)**

POLYGON	Draws a multi-sided figure on the current window or report.
array	An array of SHORT integers that specify the x and y coordinates of each "corner point" of the polygon.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **POLYGON** procedure places a multi-sided figure on the current window or report. The polygon is always closed.

The *array* parameter contains the x and y coordinates of each "corner point" of the polygon. The polygon will have as many corner points as the total number of array elements divided by two. For each corner point in turn, its x coordinate is taken from the odd-numbered array element and the y coordinate from the immediately following even-numbered element.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The line's style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
Corners   SHORT,DIM(8)

  CODE
  Corners[1] = 0        !1st x position
  Corners[2] = 90       !1st y position
  Corners[3] = 90       !2nd x position
  Corners[4] = 190      !2nd y position
  Corners[5] = 100      !3rd x position
  Corners[6] = 200      !3rd y position
  Corners[7] = 50       !4th x position
  Corners[8] = 60       !4th y position
  OPEN(MDIChild)
  POLYGON(Corners,000000FFh)     !Blue filled four-sided polygon
```

ROUNDBOX (draw a box with round corners)

ROUNDBOX(*x* ,*y* ,*width* ,*height* [,*fill*] **)**

ROUNDBOX	Draws a rectangular box with rounded corners on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **ROUNDBOX** procedure places a rectangular box with rounded corners on the current window or report. The position and size of the box are specified by *x*, *y*, *width*, and *height* parameters.

The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the box. The box extends to the right and down from its starting point.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChild  WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  ROUNDBOX(100,50,100,50,00FF0000h)    !Red round-cornered box
```

SETPENCOLOR (set line draw color)

SETPENCOLOR([*color*] **)**

SETPENCOLOR Sets the current pen color.

color A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value. If omitted, the Windows system color for window text is set.

The **SETPENCOLOR** procedure sets the current pen color for use by all graphics procedures. The default color is the Windows system color for window text.

Every window has its own current pen color. Therefore, to consistently use the same pen (which does not use the default color setting) across multiple windows, the SETPENCOLOR statement should be issued for each window.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  SETPENCOLOR(000000FFh)              !Set blue pen color
  ROUNDBOX(100,50,100,50,00FF0000h)   !Red round-cornered box with blue border
```

SETPENSTYLE (set line draw style)

SETPENSTYLE([*style*] **)**

SETPENSTYLE Sets the current pen style.

style An integer constant, constant EQUATE, or variable that specifies the pen's style. If omitted, a solid line is set.

The **SETPENSTYLE** procedure sets the current line draw style for use by all graphics procedures. The default is a solid line.

Every window has its own current pen style. Therefore, to consistently use the same pen (which does not use the default style setting) across multiple windows, the SETPENSTYLE statement should be issued for each window.

EQUATE statements for the pen styles are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

PEN:solid	Solid line
PEN:dash	Dashed line
PEN:dot	Dotted line
PEN:dashdot	Mixed dashes and dots

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  SETPENCOLOR(000000FFh)            !Set blue pen color
  SETPENSTYLE(PEN:dash)             !Set dashes for line style
  ROUNDBOX(100,50,100,50,00FF0000h)
                                    !Red round-cornered box with blue dashed border
```

SETPENWIDTH (set line draw thickness)

SETPENWIDTH([*width*] **)**

SETPENWIDTH Sets the current pen width.

width An integer expression that specifies the pen's thickness, measured in dialog units (unless overridden by the THOUS, MM, or POINTS attribute on a REPORT). If omitted, the default (one pixel) is set.

The **SETPENWIDTH** procedure sets the current line draw thickness for use by all graphics procedures. The default is one pixel, which may be set with a *width* of zero (0).

Every window has its own current pen width. Therefore, to consistently use the same pen (which does not use the default width setting) across multiple windows, the SETPENWIDTH statement should be issued for each window.

Example:

```
MDIChild  WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  SETPENCOLOR(000000FFh)          !Set blue pen color
  SETPENSTYLE(PEN:dash)           !Set dashes for line style
  SETPENWIDTH(2)                  !Set two dialog unit thickness
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

SHOW (write to screen)

SHOW(*x* ,*y* ,*string* **)**

SHOW	Writes a *string* to the current window or report.
x	An integer expression that specifies the horizontal position of the starting point, in dialog units.
y	An integer expression that specifies the vertical position of the starting point, in dialog units.
string	A string constant, variable, or expression containing the formatted text to place on the current window or report.

SHOW writes the *string* text to the current window or report. The font used is the current font for the window or report.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  DISPLAY
  SHOW(100,100,FORMAT(TODAY(),@D3))         !Display the date
  SHOW(20,20,'Press Any Key to Continue')   !Display a message
```

TYPE (write string to screen)

TYPE(*string***)**

TYPE	Writes a *string* to the current window or report.
string	A string constant, variable, or expression.

TYPE writes a *string* to the current window or report. The *string* appears on the window or report at the current cursor position, and "wraps around" if the *string* length extends beyond the right edge. The font used is the current font for the window or report. The SHOW statement may be used to position the cursor before output from TYPE.

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
          END
  CODE
  OPEN(MDIChild)
  DISPLAY
  TYPE(Cus:Notes)         !Type the notes field
```

Graphics Functions

PENCOLOR (return line draw color)

PENCOLOR( )

The **PENCOLOR** function returns the current pen color set by SETPENCOLOR.

Return Data Type: LONG

Example:

```
Proc1       PROCEDURE
MDIChild1   WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              !window controls
            END
  CODE
  OPEN(MDIChild1)
  SETPENCOLOR(000000FFh)        !Set blue pen color
  Proc2                         !Call another procedure

Proc2       PROCEDURE
MDIChild2   WINDOW('Child Two'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              !window controls
            END
ColorNow    LONG
  CODE
  ColorNow = PENCOLOR()           !Get current pen color
  OPEN(MDIChild2)
  SETPENCOLOR(ColorNow)           !Set same pen color
  SETPENSTYLE(PEN:dash)           !Set dashes for line style
  SETPENWIDTH(2)                  !Set two dialog unit thickness
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

PENSTYLE (return line draw style)

PENSTYLE( )

The **PENSTYLE** function returns the current line draw style set by SETPENSTYLE.

EQUATE statements for the pen styles are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

PEN:solid	Solid line
PEN:dash	Dashed line
PEN:dot	Dotted line
PEN:dashdot	Mixed dashes and dots

Return Data Type: LONG

Example:

```
Proc1     PROCEDURE
MDIChild1 WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
             !window controls
          END
  CODE
  OPEN(MDIChild1)
  SETPENCOLOR(000000FFh)        !Set blue pen color
  SETPENSTYLE(PEN:dash)         !Set dashes for line style
  Proc2                         !Call another procedure

Proc2     PROCEDURE
MDIChild2 WINDOW('Child  Two'),AT(0,0,320,200),MDI,MAX,HVSCROLL
             !window controls
          END
ColorNow  LONG
StyleNow  LONG
  CODE
  ColorNow = PENCOLOR()           !Get current pen color
  StyleNow = PENSTYLE()           !Get current pen style
  OPEN(MDIChild2)
  SETPENCOLOR(ColorNow)           !Set same pen color
  SETPENSTYLE(StyleNow)           !Set same pen style
  SETPENWIDTH(2)                  !Set two dialog unit thickness
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

PENWIDTH (return line draw thickness)

PENWIDTH( )

The **PENWIDTH** function returns the current line draw thickness set by SETPENWIDTH. The return value is in dialog units (unless overridden by the THOUS, MM, or POINTS attributes on a REPORT).

Return Data Type: LONG

Example:

```
Proc1     PROCEDURE
MDIChild1 WINDOW('Child  One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
             !window controls
          END

  CODE
  OPEN(MDIChild1)
  SETPENCOLOR(000000FFh)        !Set blue pen color
  SETPENSTYLE(PEN:dash)         !Set dashes for line style
  SETPENWIDTH(2)                !Set two dialog unit thickness
  Proc2                         !Call another procedure

Proc2     PROCEDURE
MDIChild2 WINDOW('Child  Two'),AT(0,0,320,200),MDI,MAX,HVSCROLL
             !window controls
          END
ColorNow  LONG
StyleNow  LONG
WidthNow  LONG
  CODE
  ColorNow = PENCOLOR()           !Get current pen color
  StyleNow = PENSTYLE()           !Get current pen style
  WidthNow = PENWIDTH()           !Get current pen width
  OPEN(MDIChild2)
  SETPENCOLOR(ColorNow)           !Set same pen color
  SETPENSTYLE(StyleNow)           !Set same pen style
  SETPENWIDTH(WidthNow)           !Set same pen width
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

Data File Structures

Contents

FILE (declare a data file structure)

```
label    FILE,DRIVER( ) [,CREATE] [,RECLAIM] [,OWNER( )] [,ENCRYPT] [,NAME()] [,PRE( )]
                                  [,BINDABLE] [,THREAD] [,EXTERNAL] [,DLL] [,OEM]
label      [INDEX( )]
label      [KEY( )]
label      [MEMO( )]
label      [BLOB]
[label]    RECORD
             fields
           END
         END
```

FILE	Declares a data file.
DRIVER	Specifies the data file type. The DRIVER attribute is required on all FILE structure declarations.
CREATE	Allows the file to be created with the CREATE statement during program execution.
RECLAIM	Specifies reuse of deleted record space.
OWNER	Specifies the password for data encryption.
ENCRYPT	Encrypt the data file.
NAME	Set DOS filename specification.
PRE	Declare a label prefix for the structure.
BINDABLE	Specify all variables in the RECORD structure may be used in dynamic expressions.
THREAD	Specify memory for the record buffer is separately allocated for each execution thread, when the file is opened on the thread.
EXTERNAL	Specify the FILE is defined, and the memory for its record buffer is allocated, in an external library.
DLL	Specify the FILE is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
OEM	Specify string data is converted from OEM ASCII to ANSI when read from disk and ANSI to OEM ASCII before writing to disk.
INDEX	Declare a static file access index which must be built at run time.
KEY	Declare a dynamically updated file access index.
MEMO	Declare a variable length text field up to 64K in length.

BLOB Declare a variable length memo field which may be greater than 64K in length.

RECORD Declare a record structure for the *fields*. A RECORD structure is required in all FILE structure declarations.

fields Data elements in the RECORD structure.

FILE declares a data file structure which is an exact description of a data file residing on disk. The label of the FILE structure is used in file processing statements and functions to effect operations on the disk file. The FILE structure must be terminated by a period or the END statement.

All attributes of the FILE, KEY, INDEX, MEMO, data declaration statements, and the data types which a FILE may contain, are dependent upon the support of the file driver. Anything in the FILE declaration which is not supported by the file system specified in the DRIVER attribute will cause a file driver error message when the FILE is opened. Attribute and/or data type exclusions for a specific file system are listed in the file driver's documentation.

At run-time, the RECORD structure is assigned memory for a data buffer where records from the disk file may be processed by executable statements. A RECORD structure is required in a FILE structure. Memory for a data buffer for any MEMO fields is allocated only when the FILE is opened, and de-allocated when the FILE is closed.

A FILE with the BINDABLE attribute declares all the variables within the RECORD structure as available for use in a dynamic expression, without requiring a separate BIND statement for each (allowing BIND(file) to enable all the fields in the file). The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A FILE with the THREAD attribute declares a separate record buffer (and file control block) for each execution thread that OPENs the FILE. If the thread does not OPEN the file, no record buffer is allocated for the file on that thread. A FILE with the EXTERNAL attribute is declared and may be referenced in Clarion code, but is not allocated memory. The memory for the FILE's record buffer is allocated by the external library. This allows a Clarion program access to FILEs declared as public in external libraries.

Example:

```
Names FILE,DRIVER('Clarion')    !Declare a file structure
Rec     RECORD                  !Required record structure
Name      STRING(20)            ! containing one or more data elements
      . .                       !End file and record declaration
```

CREATE (allow data file creation)

CREATE

The **CREATE** attribute of a FILE declaration allows a disk file to be created by the CREATE statement from within the PROGRAM where the FILE is declared. This adds some overhead, as all the file information must be contained in the excutable program.

Example:

```
Names FILE,DRIVER('Clarion'),CREATE      !Declare a file, allow create
Rec     RECORD
Name      STRING(20)
      . .
```

DRIVER (specify data file type)

DRIVER(*filetype* [,*driver string*]**)**

DRIVER	Specifies the file system the file uses.
filetype	A string constant containing the name of the file manager (Btrieve, Clarion, etc.).
driver string	A string constant or variable containing any additional instructions to the file driver.

The **DRIVER** attribute specifies which file driver is used to access the data file. DRIVER is a required attribute of all FILE declarations.

Clarion programs use file drivers for physical file access. A file driver acts as a translator between a Clarion program and the file system, eliminating different access commands for each file system. File drivers allow access to files from different file systems without changes in the Clarion syntax.

The specific implementation method of each Clarion file access command is dependent on the file driver. Some commands may not be available in a file driver due to limitations in the file system. Each file driver is documented separately. Any unsupported file access commands, FILE declaration attributes, data types, and/or file system idiosyncracies are listed there.

Example:

```
Names   FILE,DRIVER('Clarion')     !Begin file declaration
Record    RECORD
Name        STRING(20)
       . .
```

NAME (set filename)

NAME([| *constant* | | *variable* | **])**

NAME Specifies the DOS filename of the file.

constant A string constant.

variable The label of a static string variable. This may be declared as Global data, Module data, or Local data with the STATIC attribute.

The **NAME** attribute on a FILE statement specifies the DOS filename for the file driver. If the *constant* or *variable* does not contain a drive and path, the current drive and directory are assumed. If the extension is omitted, the directory entry assumes the file driver's default value.

Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files. Therefore, a NAME may also be placed on a KEY, INDEX, or MEMO. A NAME attribute without a *constant* or *variable* defaults to the label of the declaration statement on which it is placed (including any specified prefix).

NAME(*constant*) may be used on any field declared within the RECORD structure. This provides the file driver with the name of a field as it may be used in that driver's file system.

Example:

```
Cust    FILE,PRE(Cus),NAME(CustName)           !Filename in CustName variable
CustKey   KEY('Name'),NAME('c:\data\cust.idx')  !Declare key, cust.idx
Record    RECORD
Name        STRING(20)                          !Default NAME to 'Cus:Name'
       . .
```

See Also: FILE, KEY, INDEX

ENCRYPT (encrypt data file)

ENCRYPT

The **ENCRYPT** attribute is used in conjunction with the OWNER attribute to disguise the information in a data file. ENCRYPT is only valid with an OWNER attribute. Even with a "hex-dump" utility, the data in an encrypted file is extremely difficult to decipher.

Example:

```
Names   FILE,DRIVER('Clarion'),OWNER('Clarion'),ENCRYPT
Record    RECORD
Name        STRING(20)
          . .
```

See Also: OWNER

OWNER (declare password for data encryption)

OWNER(*password***)**

OWNER Specifies a file encryption password.

password A string constant or variable.

The **OWNER** attribute specifies the *password* which is used by the ENCRYPT attribute to encrypt the data.

An OWNER attribute without an accompanying ENCRYPT attribute is allowed by some file systems. The exact implementation of this attribute is file driver dependent.

Example:

```
Customer  FILE,DRIVER('Clarion'),OWNER('abCdeF'),ENCRYPT
                !Encrypt data password "abCdeF"
Record       RECORD
Name           STRING(20)
             . .
```

See Also: ENCRYPT

RECLAIM (reuse deleted record space)

RECLAIM

The **RECLAIM** attribute specifies that the file driver adds new records to the file in the space previously used by a record that has been deleted, if available. Otherwise, the record is added at the end of the file. Implementation of RECLAIM is file driver specific and may not be supported in all file systems.

Example:

```
Names   FILE,DRIVER('Clarion'),RECLAIM  !Reuse deleted record space
Record    RECORD
Name        STRING(20)
          . .
```

PRE (set file label)

PRE([*prefix*] **)**

PRE Provides a label prefix for complex data structures.

prefix Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alpha character (or underscore) and must not be a reserved word. By convention, a *prefix* is 1-3 characters, although it can be longer.

The **PRE** attribute provides a label prefix for the file. It is used to distinguish between identical variable names that occur in different structures. When a data element from the file is referenced in executable statements, assignments, and parameter lists, the *prefix* is attached to its label by a colon (Pre:Label).

Another method to distinguish between identical variable names that occur in different structures does not use the PRE attribute, but instead uses the Field Qualification syntax. When referenced in executable statements, assignments, and parameter lists, the label of the structure containing the field is attached to the field label by a colon (GroupName:Label). In the case of a file that also contains a RECORD structure (that has a label), the individual fields are addressed as FileLabel:RecordLabel:FieldName. If the file contains a RECORD structure without a label, the individual fields are addressed as FileLabel:FieldName.

Example:

```
MasterFile  FILE,DRIVER('Clarion'),PRE(Mst)     !Declare master file layout
Record        RECORD
AcctNumber      LONG
            . .
Detail      FILE,DRIVER('Clarion'),PRE(Dtl)     !Declare detail file layout
Record        RECORD
AcctNumber      LONG
            . .
            GROUP,PRE(Mem)                      !Declare memory variables
Message       STRING(30)
            END
  CODE
  IF Dtl:AcctNumber <> Mst:AcctNumber           !Is it a new account
    Mem:Message = 'New Account'                 ! display message
    DO MatchMaster                              ! get new record
  END

  IF Detail:Record:AcctNumber <> Mssterfile:Record:AcctNumber  !Same expression
    Mem:Message = 'New Account'                 ! display message
    DO MatchMaster                              ! get new record
  END
```

See Also: Reserved Words, Field Qualification

BINDABLE (set runtime expression string RECORD variables)

BINDABLE

The **BINDABLE** attribute on a FILE statement declares a RECORD structure whose constituent variables are all available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

The BIND(group) form of the BIND statement must still be used in the executable code before the individual fields in the RECORD structure may be used.

Example:

```
Names     FILE,DRIVER('Clarion'),BINDABLE   !Bindable Record structure
Record      RECORD
Name          STRING(20)
FileName      STRING(8),NAME('FName')       !Dynamic name: FName
Dot           STRING(1)                     !Dynamic name: Dot
Extension     STRING(3),NAME('EXT')         !Dynamic name: EXT
            . .
  CODE
  OPEN(Names)
  BIND(Names)
```

See Also: BIND, UNBIND, EVALUATE

THREAD (set thread-specific record buffer)

THREAD

The **THREAD** attribute declares a FILE which is allocated memory for its record buffer (and file control block) separately for each execution thread in the program. This makes the values contained in the record buffer dependent upon which thread is executing.

Whenever a new execution thread is started, the FILE must be OPENed again to receive a new instance of the record buffer.

Example:

```
            PROGRAM
            MAP
              Thread1
              Thread2
            END
Names       FILE,DRIVER('Clarion'),PRE(Nam),THREAD    !Threaded file
NbrNdx        INDEX(Nam:Number),OPT
Rec           RECORD
Name            STRING(20)
Number          SHORT
            . .
  CODE
  START(Thread1)
  START(Thread2)

Thread1   PROCEDURE
  CODE
  OPEN(Names)                    !OPEN creates new record buffer instance
  GET(Names,1)                   ! containing the 1st record in the file

Thread2   PROCEDURE
  CODE
  OPEN(Names)                    !OPEN creates another new record buffer instance
  GET(Names,5)                   ! containing the 5th record in the file
```

See Also: START, Data Declarations and Memory Allocation

EXTERNAL (set file defined externally)

EXTERNAL(*member* **)**

EXTERNAL	Specifies the FILE is defined in an external library.
member	A string constant. Valid only on a FILE declaration. It contains the filename (without extension) of the MEMBER module containing the FILE definition without the EXTERNAL attribute. If the FILE is defined in a PROGRAM module, an empty *member* string (‘’) is required.

The **EXTERNAL** attribute specifies that the FILE on which it is placed is defined in an external library. Therefore, a FILE with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory for a record buffer. The memory for the FILE’s record buffer is allocated by the external library. This allows the Clarion program access to FILEs declared as public in external libraries.

When using EXTERNAL(*member*) to declare a FILE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the FILE without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the FILE with the EXTERNAL attribute. This ensures that there is only one record buffer allocated for the FILE and all the libraries and the .EXE will reference the same memory when referring to data elements from that FILE.

The FILE declarations in all libraries (or .EXEs) that reference common files must be EXACTLY the same (with the appropriate addition of the EXTERNAL attribute). If they are not exactly the same, data corruption could occur. The actual consequence of incompatible FILE declarations is dependent upon the file driver for that file system. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer’s responsibility to ensure that consistency is maintained.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same files would have one .DLL containing the actual FILE definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common FILEs with the EXTERNAL attribute.

Example:

```
  PROGRAM
  MAP
   MODULE('LIB.LIB')
     AddCount                     !External library procedure
  . .

TotalCount  LONG,EXTERNAL         !A variable declared in an external library

Cust    FILE,PRE(Cus),EXTERNAL('')    !A File defined in a PROGRAM module
CustKey   KEY('Cus:Name')             ! whose .LIB is linked into this program
Record    RECORD
Name        STRING(20)
        . .

Contact   FILE,PRE(Con),EXTERNAL('LIB01') !A File defined in a MEMBER module
ContactKey  KEY('Con:Name')               ! whose .LIB is linked into this program
Record      RECORD
Name          STRING(20)
          . .

! The LIB.CLW file contains:
  PROGRAM
  MAP
   MODULE('LIB01')
     AddCount  !
  . .

TotalCount  LONG                          !The TotalCount variable definition

Cust      FILE,PRE(Cus)                   !The Cust File definition where the
CustKey     KEY('Cus:Name')               ! record buffer is allocated
Record      RECORD
Name          STRING(20)
          . .

  CODE
  !Executable code ...

! The LIB01.CLW file contains:
 MEMBER('LIB')

Contact     FILE,PRE(Con)         !The Contact File definition where the
ContactKey    KEY('Con:Name')     ! record buffer is allocated
Record        RECORD
Name            STRING(20)
            . .

AddCount PROCEDURE
  CODE
  TotalCount += 1
```

DLL (set file defined externally in .DLL)

DLL([*flag*] **)**

DLL	Declares a FILE defined externally in a .DLL.
flag	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the FILE on which it is placed is defined in a .DLL. A FILE with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the FILE.

The FILE declarations in all libraries (or .EXEs) that reference common FILEs must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL to declare a FILE shared by .DLLs and .EXE, only one .DLL should define the FILE without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the FILE with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the FILE and all the .DLLs and the .EXE will reference the same memory when referring to that FILE.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same FILEs would have one .DLL containing the actual file definition that only contains FILE and global data definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
Cust    FILE,PRE(Cus),EXTERNAL(''),DLL    !File defined in PROGRAM module of a .DLL
CustKey   KEY('Cus:Name')
Record    RECORD
Name        STRING(20)
        . .
```

See Also: EXTERNAL

OEM (set international string support)

OEM

The **OEM** attribute specifies that the FILE on which it is placed contains non-English language string data. These strings are automatically translated from the OEM ASCII character set data contained in the file to the ANSI character set for display in Windows. All string data in the record is automatically translated from the ANSI character set to the OEM ASCII character set before the record is written to disk.

The specific OEM ASCII character set used for the translation comes from the DOS code page loaded by the *country*.SYS file. This makes the data file specific to the language used for that code page, and means the data may not be useable on a computer with a different code page loaded. This attribute may not be supported by all file systems; consult the specific file driver's documentation.

Example:

```
Cust    FILE,DRIVER('TopSpeed'),PRE(Cus),OEM   !Contains international strings
CustKey   KEY('Cus:Name')
Record    RECORD
Name        STRING(20)
          . .

Screen  WINDOW('Window')
          ENTRY(@S20),USE(Cus:Name)
          BUTTON('&OK'),USE(?Ok),DEFAULT
          BUTTON('&Cancel'),USE(?Cancel)
        END

  CODE
  OPEN(Cust)                          !Open Cust file
  SET(Cust)
  NEXT(Cust)                          !Get record, ASCII strings are automatically
                                      ! translated to ANSI character set
  OPEN(Screen)                        !Open window and display ANSI data
  ACCEPT
    CASE FIELD()
    OF ?Ok
      CASE EVENT()
      OF EVENT:Accepted
        PUT(Cust)                     !Put record, ANSIstrings are automatically
                                      ! translated to the OEM ASCII character set
                                      ! per the loaded DOS code page
        BREAK
      END
    END
  END
  CLOSE(Screen)
  CLOSE(Cust)
```

File Structure Statements

INDEX (declare static file access index)

label **INDEX([-/+][*field*],...,[-/+][*field*]) [,NAME( )] [,NOCASE] [,OPT]**

INDEX	Declares a static index into the data file.
-/+	The - (*minus sign*) preceding an index component *field* specifies descending order for that component. If omitted, or + (*plus sign*) the component is sorted in ascending order.
field	The label of a field in the RECORD structure of the FILE in which the INDEX is declared. The *field* is an index component. A field declared with the DIM attribute (an array) may not be used as an index component.
NAME	Specifies the disk file specification for the INDEX.
OPT	Excludes, from the INDEX, those records with null values (zero or blank) in all index component fields.
NOCASE	Specifies case insensitive sort order.

INDEX declares a "static key" for a FILE structure. An INDEX is updated only by the BUILD statement. It is used to access records in a different logical order than the "physical order" of the file. An INDEX may be used for either sequential file processing or direct random access. An INDEX always allows duplicate entries. An INDEX may have more than one component *field*. The order of the components determines the sort sequence of the index. The first component is the most general, and the last component is the most specific. Generally, a data file may have up to 255 indexes (and/or keys) and each index may be up to 255 bytes, but the exact numbers are file driver dependent. An INDEX declared without a *field* creates a "dynamic index." A dynamic index may use any field (or fields) in the RECORD as components (except arrays). The component fields of a dynamic index are defined at run time in the second parameter of the BUILD statement. The same dynamic index declaration may be built and rebuilt using different component fields each time.

Example:

```
Names     FILE,DRIVER('Clarion'),PRE(Nam)
NameNdx     INDEX(Nam:Name),NOCASE        !Declare the name index
NbrNdx      INDEX(Nam:Number),OPT         !Declare the number index
DynamicNdx  INDEX()                       !Declare a dynamic index
Rec         RECORD
Name          STRING(20)
Number        SHORT
            . .
```

See Also: KEY, BUILD

KEY (declare dynamic file access index)

label **KEY([-/+]***field*,...,[**-/+**][*field*]**)** [,**DUP**] [,**NAME( )**] [,**NOCASE**] [,**OPT**] [,**PRIMARY**]

KEY	Declares a dynamically maintained index into the data file.
-/+	The - (*minus sign*) preceding a key component *field* specifies descending order for that component. If omitted, or + (*plus sign*), the component is sorted in ascending order.
field	The label of a field in the RECORD structure of the FILE in which the KEY is declared. The *field* is a key component. A field declared with the DIM attribute (an array) may not be used as a key component.
NAME	Specifies the disk file specification of the KEY.
DUP	Allows multiple records with duplicate values in their key component fields.
NOCASE	Specifies case insensitive sort order.
OPT	Excludes, from the KEY, those records with null (zero or blank) values in all key component fields.
PRIMARY	Specifies the KEY is the file's relational primary key (a unique key containing all records in the file).

A **KEY** is an index into the data file which is automatically updated whenever records are added, changed, or deleted. It is used to access records in a different logical order than the "physical order" of the file. A KEY may be used for either sequential file processing or direct random access.

A KEY may have more than one component *field*. The order of the components determines the sort sequence of the key. The first component is the most general, and the last component is the most specific. Generally, a data file may have up to 255 keys (and indexes) and each key may be up to 255 bytes, but the exact numbers are file driver dependent.

Example:

```
Names   FILE,DRIVER('Clarion'),PRE(Nam)
NameKey   KEY(Nam:Name),NOCASE,DUP        !Declare the name key
NbrKey    KEY(Nam:Number),OPT             !Declare the number key
Rec       RECORD
Name        STRING(20)
Number      SHORT
          . .
  CODE
  Nam:Name = 'Clarion Software'       !Initialize key field
  GET(Names,Nam:NameKey)              !Get the record
  SET(Nam:NbrKey)                     !Set sequential by number
```

See Also: SET, GET, INDEX

MEMO (declare a text field)

label **MEMO(***length***)** [,**BINARY**] [,**NAME( )**]

MEMO	Declares a fixed-length string which is stored variable-length on disk.
length	A numeric constant that determines the maximum number of characters. The range is from 1 to 65,520 bytes in 16-bit applications, unlimited in 32-bit applications.
BINARY	Declares the MEMO a storage area for binary data.
NAME	Specifies the disk filename for the MEMO field. (Use of this parameter is file driver dependent.)

MEMO declares a fixed-length string field which is stored variable-length on disk. The *length* parameter defines the maximum size of a memo. A MEMO must be declared before the RECORD structure. Memory is allocated for a MEMO field's buffer when the file is opened, and is deallocated when the file is closed.

Generally, up to 255 MEMO fields may be declared in a FILE structure. The exact number of MEMO fields and their manner of storage on disk is file driver dependent. MEMO fields are usually displayed in TEXT fields in SCREEN and REPORT structures.

Example:

```
Names   FILE,DRIVER('Clarion'),PRE(Nam)
NameKey   KEY(Nam:Name)
NbrKey    KEY(Nam:Number)
Notes     MEMO(4800)              !Memo, 4800 bytes
Rec       RECORD
Name        STRING(20)
Number      SHORT
        . .
```

BLOB (declare a variable-length memo field)

label **BLOB** [,**BINARY**] [,**NAME( )**]

BLOB	Declares a variable-length string which may be greater than 64K (in both 16 and 32-bit applications).
BINARY	Declares the BLOB a storage area for binary data.
NAME	Specifies the disk filename for the BLOB field. (Use of this parameter is file driver dependent.)

BLOB (Binary Large OBject) declares a string field which is completely variable-length and may be greater than 64K in size (in both 16 and 32-bit applications). A BLOB must be declared before the RECORD structure. Memory for a BLOB is dynamically allocated and de-allocated as necessary. Generally, up to 255 BLOB fields may be declared in a FILE structure. The exact number of BLOB fields and their manner of storage on disk is file driver dependent.

A BLOB may not be accessed “as a whole;” you must use “string slicing” syntax to access one piece (up to 64K) at a time. A BLOB may not be used in the same manner as a variable (may not be named as a control’s USE attribute, etc.).You can use PROP:Handle to get the windows handle to the BLOB entity. This provides the only mechanism to assign one BLOB to another: get the handle of both BLOB entities and then assign one BLOB’s handle to the other BLOB’s handle. The SIZE function returns the number of bytes contained in the BLOB field for the current record in memory. You can also get (and set) the size of a BLOB using PROP:BlobSize.

Example:

```
Names     FILE,DRIVER('TopSpeed')
NbrKey      KEY(Names:Number)
Notes       BLOB              !Can be larger than 64K
Rec         RECORD
Name          STRING(20)
Number        SHORT
            . .
ArcNames  FILE,DRIVER('TopSpeed')
NbrKey      KEY(ArcNames:Number)
Notes       BLOB
Rec         RECORD
Name          STRING(20)
Number        SHORT
            . .
  CODE
  OPEN(Names)
  CREATE(ArcNames)
  SET(Names)
  LOOP
    NEXT(Names)
    IF ERRORCODE() THEN BREAK.
    ArcNames:Rec = Names:Rec          !Assign record data to Archive file
    ArcNames:Notes{PROP:Handle} = Names:Notes{PROP:Handle}  !Assign BLOB to Archive
    ADD(ArcNames)
  END
```

RECORD (declare record structure)

```
[label]  RECORD [,PRE( )] [,NAME( )]
           fields
         END
```

RECORD	Declares the beginning of the data structure within the FILE declaration.
fields	Multiple variable declarations.
PRE	Specify a label prefix for the structure.
NAME	Specifies an external name for the RECORD structure. (Use of this parameter is file driver dependent.)

The **RECORD** statement declares the beginning of the data structure within the FILE declaration. A RECORD structure is required in a FILE declaration. Each *field* is an element of the RECORD structure. The length of a RECORD structure is the sum of the length of its fields. When the label of a RECORD structure is used in an assignment statement, expression, or parameter list, it is treated as a GROUP data type.

At run time, static memory is allocated as a data buffer for the RECORD structure. The *fields* in the record buffer are available whether the file is open or closed.

If the *fields* contain variable declarations with initial values, that initial value is only used to determine the size of the variable, the record buffer is not initialized to the value. For example, a STRING('abc') field declaration creates a three-byte string, but it's value is not automatically initialized to 'abc' unless the program's executable code assigns it that value.

Records from the data file on disk are read into the data buffer with the NEXT, PREVIOUS, or GET statements. Data in the *fields* are processed, then written to the data file as a single RECORD unit by the ADD, PUT, or DELETE statements.

Example:

```
Names   FILE,DRIVER('Clarion')      !Declare a file structure
Record    RECORD                    ! begin record declaration
Name        STRING(20)              !   declare name field
Number      SHORT                   !   declare number field
          . .                       !End file, end record declaration
```

INDEX, KEY and MEMO Attributes

BINARY (MEMO contains binary data)

BINARY

The **BINARY** attribute of a MEMO declaration specifies the MEMO field will receive data that is not just ASCII characters. This attribute is normally used to store small graphic images for display in an IMAGE field on screen.

Example:

```
Names   FILE,DRIVER('Clarion'),PRE(Nam)
NameKey   KEY(Nam:Name)
NbrKey    KEY(Nam:Number)
Picture   MEMO(48000),BINARY          !Binary memo - 48,000 bytes
Rec       RECORD
Name        STRING(20)
Number      SHORT
        . .
```

See Also: MEMO, IMAGE

DUP (allow duplicate KEY entries)

DUP

The **DUP** attribute of a KEY declaration allows multiple records with the same key value to occur in a file. If the DUP attribute is omitted, attempting to ADD or PUT records with duplicate key values will cause the "Creates Duplicate Key" error, and the record will not be written to the file. During sequential processing using the KEY, records with duplicate key values are accessed in the physical order their entries appear in the KEY file. The GET and SET statements access the first record in a set of duplicates. The DUP attribute is unnecessary on INDEX declarations because an INDEX always allows duplicate entries.

Example:

```
Names   FILE,DRIVER('Clarion'),PRE(Nam)
NameKey   KEY(Nam:Name),DUP           !Declare name key, allow duplicate names
NbrKey    KEY(Nam:Number)               !Declare number key, no duplicates allowed
Rec       RECORD
Name        STRING(20)
Number      SHORT
        . .
```

NOCASE (case insensitive KEY or INDEX)

NOCASE

The **NOCASE** attribute of a KEY or INDEX declaration makes the sorted sequence of alphabetic characters insensitive to the ASCII upper/lower case sorting convention. All alphabetic characters in key fields are converted to upper case as they are written to the KEY. This case conversion has no affect on the case of the stored data. The NOCASE attribute has no effect on non-alphabetic characters.

Example:

```
Names   FILE,DRIVER('Clarion'),PRE(Nam)
NameKey   KEY(Nam:Name),NOCASE        !Declare name key, make case insensitive
NbrKey    KEY(Nam:Number)             !Declare number key
Rec       RECORD
Name        STRING(20)
Number      SHORT
        . .
```

See Also: INDEX, KEY

OPT (exclude null KEY or INDEX entries)

OPT

The **OPT** attribute excludes entries in the KEY or INDEX for records with "null" values in all fields comprising the KEY or INDEX. For the purpose of this attribute, a "null" value is defined as zero in a numeric field or all blank spaces in a string field.

Example:

```
Names   FILE,DRIVER('Clarion'),PRE(Nam)   !Declare a file structure
NameKey   KEY(Nam:Name),OPT               !Declare name key, exclude blanks
NbrKey    KEY(Nam:Number),OPT             !Declare number key, exclude zeroes
Rec       RECORD
Name        STRING(20)
Number      SHORT
        . .
```

See Also: INDEX, KEY

PRIMARY (set relational primary key)

PRIMARY

The **PRIMARY** attribute specifies the KEY is unique, includes all records in the file, and does not allow "null" values in any of the fields comprising the KEY. This is the definition of a file's "Primary Key" per the relational database theory as expressed by E. F. Codd.

Example:

```
Names   FILE,DRIVER('TopSpeed'),PRE(Nam)    !Declare a file structure
NameKey   KEY(Nam:Name),OPT                !Declare name key, exclude blanks
NbrKey    KEY(Nam:Number),OPT,PRIMARY         !Declare number key, exclude zeroes
Rec       RECORD
Name        STRING(20)
Number      SHORT
          . .
```

See Also: KEY

NAME (set external name)

NAME([| *constant* | | *variable* | **])**

NAME Specifies an "external" name for the file driver.

constant A string constant.

variable The label of a static string variable. This may be declared as Global data, Module data, or Local data with the STATIC attribute.

The **NAME** attribute on a KEY or INDEX or MEMO statement specifies an "external" name for the key or memo for the file driver. Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files, which is specified in the NAME attribute.

NAME(*constant*) may be used on any field declared within the RECORD structure. This provides the file driver with the name of a field as it may be used in that driver's file system.

A NAME attribute without a *constant* or *variable* defaults to the label of the declaration statement on which it is placed (including any specified prefix).

Example:

```
Cust    FILE,PRE(Cus),NAME(CustName)                !Filename in CustName variable
CustKey   KEY('Cus:Name'),NAME('c:\data\cust.idx')  !Declare key, cust.idx
Record    RECORD
Name        STRING(20)
        . .
```

See Also: FILE, KEY, INDEX

File Commands

BUILD (build keys and indexes)

```
          | key    |
BUILD(    |index   | [, components [,filter ] ] )
          | file   |
```

BUILD	Builds keys and indexes.
key	The label of a KEY declaration.
index	The label of an INDEX declaration.
file	The label of a FILE declaration.
components	A string constant or variable containing the list of the component fields on which to BUILD the dynamic INDEX. If the file has the CREATE attribute, field labels may be used in the *components* parameter. Without the CREATE attribute, the contents of each field's NAME attribute must be used. The fields must be separated by commas, with leading plus (+) or minus (-) to indicate ascending or descending sequence (if supported by the file driver).
filter	A string constant, variable, or expression containing a logical expression with which to filter out unneeded records from the dynamic *index*. This requires that you name *components* for the *index*. You must BIND all variables used in the *filter* expression.

The **BUILD** statement builds keys and indexes. BUILD(*key*), BUILD(*index*), and BUILD(*file*) require exclusive access to the file. Therefore, the file must closed, LOCKed, or opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write). BUILD(*index,components*) does not require exclusive access to the file.

BUILD(*key*) or BUILD(*index*)
: Builds only that KEY or INDEX. The file must be closed, LOCKed, or opened with *access mode* set to 12h or 22h.

BUILD(*file*)
: Builds all the KEYs declared for the file. The file must be closed, LOCKed, or opened with *access mode* set to 12h or 22h.

BUILD(*index,components*)
: Allows you to BUILD a dynamic INDEX. This form of BUILD does not require exclusive access to the file, however, the file must be open (with any valid *access mode*). The dynamic INDEX is created as a temporary file, exclusive to the user who BUILDs it. The temporary

file is automatically deleted when the file is closed.

BUILD(*index*,*components*,*filter*)
: Allows you to BUILD a dynamic INDEX containing only those records whch meet the *filter* criteria. The *filter* must be in a form that is supported by the file driver.

Errors Posted:

37 File Not Open
40 Creates Duplicate Key
63 Exclusive Access Required
76 Invalid Index String

Example:

```
Names  FILE,DRIVER('TopSpeed'),PRE(Nam)   !Declare a file structure
NameKey   KEY(Nam:Name),OPT               !Declare name key
NbrNdx    INDEX(Nam:Number),OPT           !Declare number index
DynNdx    INDEX()                         !Declare a dynamic index
Rec       RECORD
Name        STRING(20),NAME('Nam:Name')
Number      SHORT,NAME('Nam:Number')
        . .

  CODE
  OPEN(Names,12h)             !Open file, exclusive read/write

  BUILD(Names)                !Build all keys on Names file

  BUILD(Nam:NbrNdx)           !Build the number index

  BUILD(Nam:DynNdx,'+Nam:Number,+Nam:Name')
                              !Build dynamic index ascending number, ascending name

  BIND('Nam:Name',Nam:Name)   !BIND the filter variable
  BUILD(Nam:DynNdx,'+Nam:Name','UPPER(Nam:Name[1]) = A')
                              !Build dynamic index of names that start with A
  UNBIND('Nam:Name')          !UNBIND the filter variable
```

See Also: OPEN, SHARE

CLOSE (close a data file)

CLOSE(*file***)**

CLOSE Closes a FILE.

file The label of a FILE.

The **CLOSE** statement closes a FILE. Generally, this flushes DOS buffers and frees any memory used by the open file other than the RECORD structure's data buffer. The exact action CLOSE takes is file driver dependent.

Example:

```
CLOSE(Customer)     !Close the customer file
```

COPY (copy a data file)

COPY(*file*,*new file***)**

COPY Duplicates a FILE.

file The label of the FILE to copy.

new file A string constant or a STRING variable containing a DOS directory file specification. If the file specification does not contain a drive and path, the current drive and directory are assumed. If only the path is specified, the filename and extension of the original *file* are used for the *new file*.

The **COPY** statement duplicates a FILE and enters the specification for the *new file* in the DOS directory. The *file* to be copied must be closed, or the "File Already Open" error is posted. If the file specification of the *new file* is identical to the original *file*, the COPY statement is ignored.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are file driver dependent. If any error is posted, the file is not copied.

Errors Posted:

02 File Not Found
03 Path Not Found
05 Access Denied
52 File Already Open

Example:

```
COPY(Names,'A:\')               !Copy Names file to floppy
COPY(CompText,Filename)         !Copy the text file to another file
```

CREATE (create an empty data file)

CREATE(*file*)

CREATE Creates an empty data file.

file The label of the FILE to be created.

The **CREATE** statement adds an empty data file to the DOS directory. If the *file* already exists, it is deleted and recreated as an empty file. The *file* must be closed, or the "File Already Open" error is posted. CREATE does not open the file for access.

Errors Posted:
03 Path Not Found
04 Too Many Open Files
05 Access Denied
52 File Already Open
54 No Create Attribute

Example:

```
CREATE(Master)       !Create a new master file
CREATE(Detail)       !Create a new detail file
```

EMPTY (empty a data file)

EMPTY(*file*)

EMPTY Deletes all records from a FILE.

file The label of a FILE.

EMPTY deletes all records from the specified *file*. EMPTY requires exclusive access to the file. Therefore, the file must be opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write).

Errors Posted: 63 Exclusive Access Required

Example:

```
OPEN(Master,18)      !Open the master file
EMPTY(Master)        ! and start a new one
```

See Also: OPEN, SHARE

FLUSH (flush DOS buffers)

FLUSH(*file*)

FLUSH	Terminates a STREAM operation, flushing the DOS buffers.
file	The label of a FILE.

The **FLUSH** statement terminates a STREAM operation. It flushes the DOS buffers, which updates the DOS directory entry for that *file*. Support for this statement is dependent upon the file system and its specific action is described in the file driver documentation.

Example:

```
STREAM(History)           !Use DOS buffering
SET(Current)              !Set to top of current file
LOOP
  NEXT(Current)
  IF ERRORCODE() THEN BREAK.
  His:Record = Cur:Record
  ADD(History)
END
FLUSH(History)            !End streaming, flush buffers
```

See Also: STREAM

LOCK (exclusive file access)

LOCK(*file* [,*seconds*]**)**

LOCK Locks a data file.

file The label of a FILE opened for shared access.

seconds A numeric constant or variable which specifies the maximum wait time in seconds.

The **LOCK** statement locks a *file* against access by other workstations in a multi-user environment. Generally, this excludes other users from writing to or reading from the *file*. The specific action LOCK takes is file driver dependent.

LOCK(*file*) Attempts to lock the *file* until it is successful. If it is already locked by another workstation, LOCK will wait until the other workstation unlocks it.

LOCK(*file*,*seconds*)
Posts the "File Is Already Locked" error after unsuccessfully trying to lock the file for the specified number of *seconds*.

The most common problem to avoid when locking files is referred to as "deadly embrace." This condition occurs when two workstations attempt to lock the same set of files in two different orders and both are using the LOCK(*file*) form of LOCK. One workstation has already locked a file that the other is trying to LOCK, and vice versa. This problem may be avoided by using the LOCK(*file*,*seconds*) form of LOCK, and always locking files in the same order.

Errors Posted: 32 File Is Already Locked

Example:

```
LOOP                        !Loop to avoid "deadly embrace"
  LOCK(Master,1)            !Lock the master file, try 1 second
  IF ERRORCODE() = 32       !If someone else has it
    BEEP(0,100)             ! pause for 1 second
    CYCLE                   ! and try again
  END
  LOCK(Detail,1)            !Lock the detail file, try 1 second
  IF ERRORCODE() = 32       !If someone else has it
    UNLOCK(Master)          ! unlock the locked file
    BEEP(0,100)             ! pause for 1 second
    CYCLE                   ! and try again
. .
```

OPEN (open a data file)

OPEN(*file* [,*access mode*]**)**

OPEN Opens a FILE structure for processing.

file The label of a FILE declaration.

access mode A numeric constant, variable, or expression which determines the level of access granted to both the user opening the file, and other users in a multi-user system. If omitted, the default value is 22h (Read/Write + Deny Write).

The **OPEN** statement opens a FILE structure for processing and sets the *access mode*. Support for various *access modes* are file driver dependent. All files must be explicitly opened before they may be accessed. The *access mode* is a bitmap which tells the operating system what access to grant the user opening the file and what access to deny to others using the file. The actual values for each access level are:

	Dec.	Hex.	Access
User Access:	0	0h	Read Only
	1	1h	Write Only
	2	2h	Read/Write
Other's Access:	0	0h	Any Access (FCB compatibility mode)
	16	10h	Deny All
	32	20h	Deny Write
	48	30h	Deny Read
	64	40h	Deny None

Errors Posted:

02 File Not Found
04 Too Many Open Files
05 Access Denied
52 File Already Open
75 Invalid Field Type Descriptor

Example:

```
ReadOnly    EQUATE(0)       !Access mode equates
WriteOnly   EQUATE(1)
ReadWrite   EQUATE(2)
DenyAll     EQUATE(10h)
DenyWrite   EQUATE(20h)
DenyRead    EQUATE(30h)
DenyNone    EQUATE(40h)
  CODE
  OPEN(Names,ReadWrite+DenyNone)    !Open fully shared access
```

See Also: SHARE

PACK (remove deleted records)

PACK(*file*)

PACK Removes deleted records from a data file and rebuilds its keys.

file The label of a FILE declaration.

The **PACK** statement removes deleted records from a data file and rebuilds its keys. The resulting data files are as compact as possible. PACK requires at least twice the disk space that the file, keys, and memos occupy to perform the process. New files are created from the old, and the old files are deleted only after the process is complete. PACK requires exclusive access to the file. Therefore, the file must be opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write).

Errors Posted: 63 Exclusive Access Required

Example:

```
OPEN(Trans,12h)          !Open the file in exclusive mode
PACK(Trans)              ! and pack it
```

See Also: OPEN, SHARE

REMOVE (erase the data file)

REMOVE(*file*)

REMOVE Deletes a FILE.

file The label of the FILE to be removed.

The **REMOVE** statement erases a file specification from the DOS directory in the same manner as the DOS Delete command. The *file* must be closed, or the "File Already Open" error is posted. If any error is posted, the file is not removed.

Errors Posted: 02 File Not Found
05 Access Denied
52 File Already Open

Example:

```
REMOVE(OldFile)     !Delete the old file
REMOVE(Changes)     !Delete the changes file
```

RENAME (change data file directory name)

RENAME(*file*,*new file*)

RENAME	Renames a FILE.
file	The label of the FILE to be renamed.
new file	A string constant or a STRING variable containing a DOS directory file specification. If the file specification does not contain a drive and path, the current drive and directory are assumed. If only the path is specified, the filename and extension of the original *file* are used for the *new file*. Files cannot be renamed to a new drive.

The **RENAME** statement changes the file specification to the specification for the *new file* in the directory. The *file* to be renamed must be closed, or the "File Already Open" error is posted. If the file specification of the *new file* is identical to the original *file*, the RENAME statement is ignored. If any error is posted, the file is not renamed.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are file driver dependent.

Errors Posted:

02 File Not Found
03 Path Not Found
05 Access Denied
52 File Already Open

Example:

```
RENAME(Text,'text.bak')       !Make it the backup
RENAME(Master,'\newdir')      !Move it to another directory
```

SHARE (open a data file)

SHARE(*file* [,*access mode*]**)**

SHARE Opens a FILE structure for processing.

file The label of a FILE declaration.

access mode A numeric constant, variable, or expression which determines the level of access granted to both the user opening the file, and other users in a multi-user system. If omitted, the default value is 42h (Read/Write, Deny None).

The **SHARE** statement opens a FILE structure for processing and sets the *access mode*. The SHARE statement is the same as the OPEN statement, with the exception of the default value of *access mode*. The *access mode* is a bitmap which tells the operating system what access to grant the user opening the file and what access to deny to others using the file. The actual values for each access level are:

```
                    Dec.    Hex.    Access
User Access:        0       0h      Read Only
                    1       1h      Write Only
                    2       2h      Read/Write

Other's Access:     0       0h      Any Access (FCB compatibility mode)
                    16      10h     Deny All
                    32      20h     Deny Write
                    48      30h     Deny Read
                    64      40h     Deny None
```

Errors Posted:

02 File Not Found
04 Too Many Open Files
05 Access Denied
52 File Already Open
75 Invalid Field Type Descriptor

Example:

```
ReadOnly    EQUATE(0)         !Access mode equates
WriteOnly   EQUATE(1)
ReadWrite   EQUATE(2)
DenyAll     EQUATE(10h)
DenyWrite   EQUATE(20h)
DenyRead    EQUATE(30h)
DenyNone    EQUATE(40h)
  CODE
  SHARE(Master,ReadOnly+DenyWrite)    !Open read only mode
```

See Also: OPEN

STREAM (enable DOS buffering)

STREAM(*file*)

STREAM Disables automatic FILE flushing.

file The label of a FILE.

Some file systems flush the DOS buffers on each disk write. The **STREAM** statement disables this automatic flushing operation. DOS buffers are allocated by the BUFFERS= statement in the Config.Sys file. They store disk writes until the buffers are full, then write the buffers to disk all at once. The directory entries for the *file* are updated only when the buffers are written to disk (flushed). A STREAM operation is terminated by closing the file, which automatically flushes the buffers, or by issuing a FLUSH statement.

Support for this statement is dependent upon the file system and is described in its file driver's documentation.

Example:

```
STREAM(History)             !Use DOS buffering
SET(Current)                !Set to top of current file
LOOP
  NEXT(Current)
  IF ERRORCODE() THEN BREAK.
  His:Record = Cur:Record
  ADD(History)
END
FLUSH(History)              !End streaming, flush buffers
```

See Also: FLUSH

UNLOCK (unlock a locked data file)

UNLOCK(*file*)

UNLOCK Unlocks a previously locked data file.

file The label of a FILE declaration.

The **UNLOCK** statement unlocks a previously LOCKed data file. It will not unlock a file locked by another user. If the *file* is not locked, or is locked by another user, UNLOCK is ignored. UNLOCK posts no errors. The specific action UNLOCK takes is file driver dependent.

Example:

```
LOOP                        !Loop to avoid "deadly embrace"
  LOCK(Master,1)            !Lock the master file, try for 1 second
  IF ERRORCODE() = 32       !If someone else has it
    BEEP(0,100)             ! pause for 1 second
    CYCLE                   ! and try again
  END
  LOCK(Detail,1)            !Lock the detail file, try for 1 second
  IF ERRORCODE() = 32       !If someone else has it
    UNLOCK(Master)          ! unlock the locked file
    BEEP(0,100)             ! pause for 1 second
    CYCLE                   ! and try again
. .                         !End if, end loop
```

Record Access Commands

ADD (add a new file record)

ADD(*file* [,*length*]**)**

ADD	Writes a new record to a FILE.
file	The label of a FILE declaration.
length	An integer constant, variable, or expression which contains the number of bytes to write to the *file*. The *length* must be greater than zero and not greater than the length of the RECORD. If omitted or out of range, *length* defaults to the length of the RECORD structure.

The **ADD** statement writes a new record from the RECORD structure data buffer to the data file. All KEYs associated with the *file* are also updated during each ADD. If an error is posted, no record is added to the file. The specific action ADD takes is file driver dependent.

If there is no room for the record on disk, the "Access Denied" error is posted.

Errors Posted:

05 Access Denied
37 File Not Open
40 Creates Duplicate Key

Example:

```
ADD(Customer)                        !Add a new customer record
  IF ERRORCODE() THEN STOP(ERROR()).  ! and check for errors
```

APPEND (add a new file record)

APPEND(*file* [,*length*]**)**

APPEND Writes a new record to a FILE.

file The label of a FILE declaration.

length An integer constant, variable, or expression which contains the number of bytes to write to the *file*. The *length* must be greater than zero and not greater than the length of the RECORD. If omitted or out of range, *length* defaults to the length of the RECORD structure.

The **APPEND** statement writes a new record from the RECORD structure data buffer to the data file. No KEYs associated with the *file* are updated during an APPEND. After APPENDing records, the KEYs must be rebuilt with the BUILD command. APPEND is usually used in batch adding a number of records at one time.

If an error is posted, no record is added to the file. The specific action APPEND takes is file driver dependent. If there is no room for the record on disk, the "Access Denied" error is posted.

Errors Posted:

05 Access Denied
37 File Not Open

Example:

```
LOOP                                    !Process an input file
  NEXT(InFile)                          ! getting each record in turn
    IF ERRORCODE() THEN BREAK.          ! break loop on error
  Cus:Record = Inf:Record               !Copy the data to Customer file
  APPEND(Customer)                      ! and APPEND a customer record
    IF ERRORCODE() THEN STOP(ERROR()).  !  check for errors
END
BUILD(Customer)                         !Re-build Keys
```

See Also: BUILD

DELETE (delete a file record)

DELETE(*file*)

DELETE Removes a record from a FILE.

file The label of a FILE declaration.

The **DELETE** statement removes the last record accessed by NEXT, PREVIOUS, GET, ADD, or PUT. The key entries for that record are also removed from the KEYs. DELETE does not clear the record buffer. Therefore, data values from the record just deleted still exist and are available for use until the record buffer is overwritten.

If no record was previously accessed, or the record is held by another workstation, DELETE posts the "Record Not Available" error and no record is deleted. The specific action DELETE takes is file driver dependent.

Errors Posted: 05 Access Denied
33 Record Not Available

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)
NameKey     KEY(Cus:Name),OPT
NbrKey      KEY(Cus:Number),OPT
Rec         RECORD
Name          STRING(20)
Number        SHORT
            . .
  CODE
  Cus:Number = 12345               !Initialize key field
  GET(Customer,Cus:NbrKey)         !Get that record
    IF ERRORCODE() THEN STOP(ERROR()).
  DELETE(Customer)                 !Delete the customer record
```

See Also: ADD, GET, HOLD, NEXT, PREVIOUS, PUT

GET (read a file record by direct access)

```
          | file,key                    |
GET(      | file,filepointer [, length ] |)
          | key,keypointer              |
```

GET Retrieves a specific record from a FILE.

file The label of a FILE declaration.

key The label of a KEY or INDEX declaration.

filepointer A numeric constant, variable, or expression for the value returned by the POINTER(*file*) function. The specific value is file driver dependent.

keypointer A numeric constant, variable, or expression for the value returned by the POINTER(*key*) function. The specific value is file driver dependent.

length An integer constant, variable, or expression which contains the number of bytes to read from the *file*. The *length* must be greater than zero and not greater than the RECORD length. If omitted or out of range, *length* defaults to the length of the RECORD structure.

The **GET** statement locates a specific record in the data file and reads it into the RECORD structure data buffer. Direct access to the record is achieved by relative record position within the file, or by matching key values.

GET(*file,key*) Gets the first record from the file (as listed in the *key*) which contains values matching the values in the component fields of the *key*.

GET(*file,filepointer* [,*length*])
Gets a record from the file based on the *filepointer* relative position within the *file*. If *filepointer* is zero, the current record pointer is cleared and no record is retrieved.

GET(*key,keypointer*)
Gets a record from the file based on the *keypointer* relative position within the *key*.

The values for *filepointer* and *keypointer* are file driver dependent. They could be: record number; relative byte position within the file; or, some other kind of "seek position" within the file. If the *filepointer* or *keypointer* value is out of range, or there are no matching *key* values in the data file, the "Record Not Found" error is posted.

Errors Posted:
35 Record Not Found
37 File Not Open
43 Record Is Already Held

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)
NameKey     KEY(Cus:Name),OPT
NbrKey      KEY(Cus:Number),OPT
Rec         RECORD
Name          STRING(20)
Number        SHORT
            . .
  CODE

  Cus:Name = 'Clarion'              !Initialize key field
  GET(Customer,Cus:NameKey)         ! get record with matching value
    IF ERRORCODE() THEN STOP(ERROR()).

  GET(Customer,3)                   !Get 3rd rec in physical file order
    IF ERRORCODE() THEN STOP(ERROR()).

  GET(Cus:NameKey,3)                !Get 3rd rec in keyed order
    IF ERRORCODE() THEN STOP(ERROR()).
```

See Also: POINTER, DUPLICATE

HOLD (exclusive file record access)

HOLD(*file* [,*seconds*]**)**

HOLD Arms record locking.

file The label of a FILE opened for shared access.

seconds A numeric constant or variable which specifies the maximum wait time in seconds.

The **HOLD** statement arms record locking for a following GET, NEXT, or PREVIOUS statement in a multi-user environment. The GET, NEXT, or PREVIOUS flags the record as "held" when it successfully gets the record. Generally, this excludes other users from writing to, but not reading, the record. The specific action HOLD takes is file driver dependent.

HOLD(*file*) Arms the process so that the following GET, NEXT, or PREVIOUS attempts to hold the record until it is successful. If it is held by another workstation, GET, NEXT, or PREVIOUS will wait until the other workstation releases it.

HOLD(*file,seconds*) Arms the process for the following GET, NEXT, or PREVIOUS to post the "Record Is Already Held" error after unsuccessfully trying to hold the record for *seconds*.

A user may HOLD one record at a time in each file. If a second record is accessed in the same file, the previously held record in that file is automatically released. A common problem to avoid is "deadly embrace." This occurs when two workstations attempt to hold the same set of records in two different orders and both are using the HOLD(*file*) form of HOLD. One workstation has already held a record that the other is trying to HOLD, and vice versa. You can avoid this problem by using the HOLD(*file,seconds*) form of HOLD, and trapping for the "Record Is Already Held" error.

Example:

```
LOOP                        !Loop to avoid "deadly embrace"
  HOLD(Master,1)            !Arm Hold on master file, try for 1 second
  GET(Master,1)             !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    BEEP(0,100); CYCLE      ! pause for 1 second and try again
  END
  HOLD(Detail,1)            !Lock the detail file, try for 1 second
  GET(Detail,1)             !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    RELEASE(Master)         ! release the held record
    BEEP(0,100); CYCLE      ! pause for 1 second and try again
  END
  BREAK
END
```

See Also: RELEASE, GET, NEXT, PREVIOUS

NEXT (read next file record in sequence)

NEXT(*file*)

NEXT Reads the next record in sequence from a FILE.

file The label of a FILE declaration.

NEXT reads the next record in sequence from a data file and places it in the RECORD structure data buffer. The SET statement determines the sequence in which records are read. The first NEXT following a SET reads the record at the position specified by the SET statement. Subsequent NEXT statements read subsequent records in that sequence. The sequence is not effected by any GET, ADD, PUT, or DELETE.

Executing NEXT without a preceding SET, or attempting to read past the end of file posts the "Record Not Available" error.

Errors Posted:
33 Record Not Available
37 File Not Open
43 Record Is Already Held

Example:

```
SET(Cus:NameKey)                  !Beginning of file in keyed sequence
LOOP                              !Read all records through end of file
  NEXT(Customer)                  ! read a record sequentially
    IF ERRORCODE() THEN BREAK.    ! break on end of file
  DO PostTrans                    ! call transaction posting routine
END
```

See Also: SET, PREVIOUS, EOF, HOLD

NOMEMO (read file record without reading memo)

NOMEMO(*file*)

NOMEMO Arms “memoless” record retrieval.

file The label of a FILE.

The **NOMEMO** statement arms “memoless” record retrieval for the next GET, NEXT, or PREVIOUS statement encountered. The following GET, NEXT, or PREVIOUS gets the record but does not get any associated MEMO field(s) for the record. Generally, this speeds up access to the record when the contents of the MEMO field(s) are not needed by the procedure.

Example:

```
SET(Master)
LOOP
  NOMEMO(Master)                !Arm “memoless” access
  NEXT(Master)                  !Get record without memo
    IF ERRORCODE() THEN BREAK.
  Queue = Mst:Record            !Fill memory queue
  ADD(Queue)
    IF ERRORCODE() THEN STOP(ERROR()).
. .
DISPLAY(?ListBox)               !Display the queue
```

See Also: GET, NEXT, PREVIOUS

PREVIOUS (read previous file record in sequence)

PREVIOUS(*file*)

PREVIOUS Reads the previous record in sequence from a FILE.

file The label of a FILE declaration.

PREVIOUS reads the previous record in sequence from a data file and places it in the RECORD structure data buffer. The SET statement determines the sequence in which records are read. The first PREVIOUS following a SET reads the record at the position specified by the SET statement. Subsequent PREVIOUS statements read subsequent records in reverse sequence. The sequence is not effected by any GET, ADD, PUT, or DELETE.

Executing PREVIOUS without a preceding SET, or attempting to read past the beginning of file posts the "Record Not Available" error.

Errors Posted:

33 Record Not Available
37 File Not Open
43 Record Is Already Held

Example:

```
SET(Trn:DateKey)                !End/Beginning of file in keyed sequence
LOOP                            !Read all records in reverse order
  PREVIOUS(Trans)               ! read a record sequentially
    IF ERRORCODE() THEN BREAK.  ! break at beginning of file
  DO LastInFirstOut             ! call last in first out routine
END
```

See Also: SET, NEXT, BOF, HOLD

PUT (write record back to file)

PUT(*file* [,*filepointer*] [,*length*]**)**

PUT Writes a record back to a FILE.

file The label of a FILE declaration.

filepointer A numeric constant, variable, or expression for the value returned by the POINTER(*file*) function. The specific value is file driver dependent.

length An integer constant, variable, or expression containing the number of bytes to write to the *file*. This must be greater than zero and not greater than the RECORD length. If omitted or out of range, the RECORD length is used.

The **PUT** statement writes the current values in the RECORD structure data buffer to a previously accessed record in the *file*.

PUT(*file*) Writes back the last record accessed with NEXT, PREVIOUS, GET, or ADD. If the values in the key variables were changed, the KEYs are updated.

PUT(*file,filepointer*)
Writes the record to the *filepointer* location in the *file* and the KEYs are updated.

PUT(*file,filepointer,length*)
Writes *length* bytes to the *filepointer* location in the *file* and the KEYs are updated.

If a record was not accessed with NEXT, PREVIOUS, GET, ADD, or was deleted, the "Record Not Available" error is posted. PUT also posts the "Creates Duplicate Key" error. If any error is posted, the record is not written to the file.

Errors Posted:
05 Access Denied
33 Record Not Available
40 Creates Duplicate Key

Example:

```
SET(Trn:DateKey)                  !End/Beginning of file in keyed sequence
LOOP                              !Read all records in reverse order
  PREVIOUS(Trans)                 ! read a record sequentially
    IF ERRORCODE() THEN BREAK.    ! break at beginning of file
  DO LastInFirstOut               !Call last in first out routine
  PUT(Trans)                      !Write transaction record back to the file
    IF ERRORCODE() THEN STOP(ERROR()).
END
```

See Also: NEXT, PREVIOUS, GET, ADD

RELEASE (release a held file record)

RELEASE(*file*)

RELEASE Releases the held record.

file The label of a FILE declaration.

The **RELEASE** statement releases a previously held record. It will not release a record held by another user. If the record is not held, or is held by another user, RELEASE is ignored.

Example:

```
LOOP                        !Loop to avoid "deadly embrace"
  HOLD(Master,1)            !Arm Hold on master file, try for 1 second
  GET(Master,1)             !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    BEEP(0,100)             ! pause for 1 second
    CYCLE                   ! and try again
  END
  HOLD(Detail,1)            !Hold the detail file, try for 1 second
  GET(Detail,1)             !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    RELEASE(Master)         ! release the held record
    BEEP(0,100)             ! pause for 1 second
    CYCLE                   ! and try again
  END
  BREAK
END
```

See Also: HOLD

REGET (reget file record)

REGET(*file,string***)**

REGET	Regets a specific record in the FILE.
file	The label of a FILE declaration.
string	The string returned by the POSITION function.

The **REGET** reads the record identified by the *string* returned by the POSITION function. The value contained in the *string* returned by the POSITION function, and its length, are file driver dependent.

Errors Posted: 33 Record Not Available

Example:

```
RecordQue     QUEUE,PRE(Dsp)
QueFields       LIKE(Trn:Record),PRE(Dsp)
              END
SavPosition   STRING(260)
  CODE
  SET(Trn:DateKey)                            !Top of file in keyed sequence
  LOOP                                        !Read all records in file
    NEXT(Trans)                               ! read a record sequentially
      IF ERRORCODE() THEN BREAK.
    RecordQue = Trn:Record                    !Move record into queue
    ADD(RecordQue)                            ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20 OR EOF(Trans) !20 records in queue or end of file?
      SavPosition = POSITION(Trn:DateKey)     !Save record position
      DO DisplayQue                           !Display the queue
      FREE(RecordQue)                         ! and free it
      REGET(Trans,SavPosition)                ! and get the record again
  . .
```

See Also: POSITION, RESET

RESET (reset file record sequence position)

RESET(*sequence,string***)**

RESET	Resets the sequential processing pointer to a specific record in the FILE.
sequence	The label of a FILE, KEY, or INDEX declaration.
string	The string returned by the POSITION function.

RESET restores the record pointer to the record identified by the *string* returned by the POSITION function. Once RESET has restored the record pointer, either NEXT or PREVIOUS will read that record.

The value contained in the *string* returned by the POSITION function, and its length, are file driver dependent. RESET is used in conjunction with POSITION to temporarily suspend and resume sequential file processing.

Example:

```
RecordQue     QUEUE,PRE(Dsp)
QueFields       LIKE(Trn:Record),PRE(Dsp)
              END
SavPosition   STRING(260)
  CODE
  SET(Trn:DateKey)                          !Top of file in keyed sequence
  LOOP                                      !Read all records in file
    NEXT(Trans)                             ! read a record sequentially
      IF ERRORCODE() THEN BREAK.
    RecordQue = Trn:Record                  !Move record into queue
    ADD(RecordQue)                          ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20 OR EOF(Trans) !20 records in queue or end of file?
      SavPosition = POSITION(Trn:DateKey)   !Save record position
      DO DisplayQue                         !Display the queue
      FREE(RecordQue)                       ! and free it
      RESET(Trn:DateKey,SavPosition)        !Reset the record pointer
      NEXT(Trans)                           ! and get the record again
  . .
```

See Also: POSITION, NEXT, PREVIOUS

SET (initiate sequential file processing)

```
              | file              |
SET(          | file,key          | )
              | file,filepointer  |
              | key               |
              | key,key           |
              | key,keypointer    |
              | key,key,filepointer |
```

SET Initializes sequential processing of a FILE.

file The label of a FILE declaration. This parameter specifies processing in the physical order in which records occur in the data file.

key The label of a KEY or INDEX declaration. When used in the first parameter position, *key* specifies processing in the sort sequence of the KEY or INDEX.

filepointer A numeric constant, variable, or expression for the value returned by the POINTER(file) function. The specific value is file driver dependent.

keypointer A numeric constant, variable, or expression for the value returned by the POINTER(key) function. The specific value is file driver dependent.

SET initializes sequential processing of a data file. SET does not get a record, but only sets up processing order and starting point for the following NEXT or PREVIOUS statements. The first parameter determines the order in which records are processed. The second and third parameters determine the starting point within the file. If the second and third parameters are omitted, processing begins at the beginning (or end) of the file.

SET(*file*) Specifies physical record order processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file.

SET(*file,key*) Specifies physical record order processing and positions to the first record which contains values matching the values in the component fields of the *key*.

SET(*file,filepointer*)
Specifies physical record order processing and positions to the *filepointer* record within the *file*.

SET(*key*) Specifies keyed sequence processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file in that sequence.

SET(*key,key*) Specifies keyed sequence processing and positions to the first or last record which contains values matching the values in the component fields of the *key*. Both *key* parameters must be the same.

SET(*key,keypointer*)

Specifies keyed sequence processing and positions to the *keypointer* record within the *key*.

SET(*key,key,filepointer*)

Specifies keyed sequence processing and positions to a record which contains values matching the values in the component fields of the *key* at the exact record number specified by *filepointer*. Both *key* parameters must be the same.

When *key* is the second parameter, processing begins at the first or last record containing values matching the values in the component fields of the KEY or INDEX. If an exact match is found, NEXT will read the first matching record while PREVIOUS will read the last matching record. If no exact match is found, the record with the next greater value is read by NEXT, the record with next lesser value is read by PREVIOUS.

The values for *filepointer* and *keypointer* are file driver dependent. They could be a record number, the relative byte position within the file, or some other kind of "seek position" within the file. These parameters are used to begin processing at a specific record within the file.

For all file drivers, an attempt to SET past the end of the file will set the EOF function to true, and an attempt to SET before the beginning of the file will set the BOF function to true.

Example:

```
SET(Customer)               !Physical file order, beginning of file

Cus:Name = 'Smith'
SET(Customer,Cus:NameKey)   !Physical file order, first record where Name = 'Smith'

SavePtr = POINTER(Customer)
SET(Customer,SavePtr)       !Physical file order, physical record number = SavePtr

SET(Cus:NameKey)            !NameKey order, beginning of file (relative to the key)

SavePtr = POINTER(Cus:NameKey)
SET(Cus:NameKey,SavePtr)    !NameKey order, key-relative record number = SavePtr

Cus:Name = 'Smith'
SET(Cus:NameKey,Cus:NameKey)
                            !NameKey order, first record where Name = 'Smith'

Cus:Name = 'Smith'
SavePtr = POINTER(Customer)
SET(Cus:NameKey,Cus:NameKey,SavePtr)
                            !NameKey order, Name = 'Smith' and rec number = SavePtr
```

See Also: NEXT, PREVIOUS, KEY, RECORD, POINTER

SKIP (bypass file records in sequence)

SKIP(*file*,*count***)**

SKIP	Bypasses records during sequential file processing.
file	The label of a FILE declaration.
count	A numeric constant or variable. The *count* specifies the number of records to bypass. If the value is positive, records are skipped in forward (NEXT) sequence. If *count* is negative, records are skipped in reverse (PREVIOUS) sequence.

The **SKIP** statement is used to bypass records during sequential file processing. It bypasses records, in the sequence specified by the SET statement, by moving the file pointer *count* records. SKIP is more efficient than NEXT or PREVIOUS for skipping past records because it does not move records into the RECORD structure data buffer.

If SKIP reads past the end or beginning of file, the EOF( ) and BOF( ) functions return true. If no SET has been issued, SKIP is ignored.

Example:

```
SET(Itm:InvoiceKey)                  !Start at beginning of Items file
LOOP                                 !Process all records
  NEXT(Items)                        ! Get a record
    IF ERRORCODE() THEN BREAK.
  IF Itm:InvoiceNo <> SavInvNo       ! Check for first item in order
    Hea:InvoiceNo = Itm:InvoiceNo    !   Initialize key field
    GET(Header,Hea:InvoiceKey)       !   Get the associated header record
      IF ERRORCODE() THEN STOP(ERROR()).
    IF Hea:InvoiceStatus = 'Cancel'  !   Is it a canceled order?
      SKIP(Items,Hea:ItemCount-1)    !     SKIP rest of the items
      CYCLE                          !     and process next order
  . .
  DO ItemProcess                     ! process the item
  SavInvNo = Itm:InvoiceNo           ! save the invoice number
END
```

WATCH (automatic file concurrency check)

WATCH(*file*)

WATCH Arms automatic optimistic concurrency checking.

file The label of a FILE declaration.

The **WATCH** statement arms automatic optimistic concurrency checking by the file driver for a following GET, NEXT, or PREVIOUS statement in a multi-user environment. Generally, the file driver retains a copy of the retrieved record on the GET, NEXT, or PREVIOUS when it successfully gets the record. When the retrieved record is PUT to the *file*, the record on disk is compared to the original record retrieved. An error is returned by the PUT statement if the record has been changed by another user. The specific action WATCH takes is file driver dependent.

Example:

```
SET(Itm:InvoiceKey)                         !Start at beginning of Items file
LOOP                                        !Process all records
  WATCH(Items)                              !Arm concurrency check
  NEXT(Items)                               ! Get a record
    IF ERRORCODE() THEN BREAK.
  DO ItemProcess                            ! process the item
  PUT(Items)                                ! and put it back
    IF ERRORCODE() THEN STOP(ERROR()).      !Stop on any error, including
                                            ! record changed by another user
END
```

File Functions

BOF (beginning of file function)

BOF(*file*)

BOF Flags the beginning of the FILE during sequential processing.

file The label of a FILE declaration.

The **BOF** function returns a non-zero value (true) when the first record in relative file sequence has been read by PREVIOUS or passed by SKIP. Otherwise, the return value is zero (false).

The BOF function is most often used as a LOOP UNTIL condition. Since a LOOP condition is evaluated at the top of the LOOP, BOF returns true after the last record has been read and processed in reverse order.

The BOF function may not be supported by all file drivers (or may be inefficient). Check the driver documentation before using this function.

Return Data Type: LONG

Example:

```
SET(Trn:DateKey)                    !End/Beginning of file in keyed sequence
LOOP UNTIL BOF(Trans)               !Process file backwards
  PREVIOUS(Trans)                   ! read a record sequentially
    IF ERRORCODE() THEN STOP(ERROR()).
  DO LastInFirstOut                 ! call last in first out routine
END
```

See Also: PREVIOUS, SKIP, LOOP

BYTES (return size in bytes)

BYTES(*file*)

BYTES	Returns number of bytes in FILE, or most recently read.
file	The label of a FILE.

The **BYTES** function returns the size of a FILE in bytes or the number of bytes in the last record accessed. Following an OPEN statement, BYTES returns the size of the file. After the *file* has been accessed by GET, NEXT, ADD, or PUT, the BYTES function returns the number of bytes accessed in the RECORD. The BYTES function may be used to return the number of bytes read in a variable length record.

Return Data Type: LONG

Example:

```
OPEN(DosFile)                              !Open the file
IF (BYTES(DosFile) % 80) > 0               !Check for short record
  SavPtr = INT(BYTES(DosFile) % 80) + 1    ! compute short record pointer
ELSE
  SavPtr = BYTES(DosFile) / 80             ! compute last record pointer
END
GET(DosFile,SavPtr)                        !Get the last record
LastRec = BYTES(DosFile)                   !Save size of the short record
```

DUPLICATE (check for duplicate key entries)

```
DUPLICATE( | key  | )
           | file |
```

DUPLICATE Checks duplicate entries in unique keys.

key The label of a KEY declaration.

file The label of a FILE declaration.

The **DUPLICATE** function returns a non-zero value (true) if writing the current record to the data file would post the "Creates Duplicate Key" error. With a *key* parameter, the specified KEY is checked. With a *file* parameter, all KEYs declared without a DUP attribute are checked.

The DUPLICATE function assumes that the contents of the RECORD structure data buffer are duplicated at the current record pointer location. Therefore, when using DUPLICATE prior to ADDing a record, the record pointer should be cleared with: GET(*file*,0).

Return Data Type: LONG

Example:

```
IF Action = 'ADD' THEN GET(Vendor,0).          !If adding, clear the file pointer
IF DUPLICATE(Vendor)                           !If this vendor already exists
  SCR:MESSAGE = 'Vendor Number already assigned' ! display message
  SELECT(?)                                    ! and stay on the field
END
```

See Also: GET

EOF (end of file function)

EOF(*file*)

EOF Flags the end of the FILE during sequential processing.

file The label of a FILE declaration.

The **EOF** function returns a non-zero value (true) when the last record in relative file sequence has been read by NEXT or passed by SKIP. Otherwise, the return value is zero (false). The EOF function is most often used as a LOOP UNTIL condition. Since a LOOP condition is evaluated at the top of the LOOP, EOF returns true after the last record has been read and processed.

The EOF function may not be supported by all file drivers (or may be inefficient). Check the driver documentation before using this function.

Return Data Type: LONG

Example:

```
SET(Trn:DateKey)          !Beginning of file in keyed sequence
LOOP UNTIL EOF(Trans)     !Process all records
  NEXT(Trans)             ! read a record sequentially
    IF ERRORCODE() THEN STOP(ERROR()).
  DO LastInFirstOut       ! call last in first out routine
END
```

See Also: NEXT, SKIP, LOOP

POINTER (return relative record position)

POINTER(| *file* | **)**
| *key* |

POINTER Returns relative record position.

file The label of a FILE declaration. This specifies physical record order within the file.

key The label of a KEY or INDEX declaration. This specifies the entry order within the KEY or INDEX file.

POINTER returns the relative record position within the data file (in *file* sequence), or the relative record position within the KEY or INDEX file (in *key* sequence) of the last record accessed. The value returned by the POINTER function is file driver dependent. It may be a record number, the relative byte position within the file, or some other kind of "seek position" within the file.

Return Data Type: LONG

Example:

```
SavePtr# = POINTER(Customer)     !Save file pointer
```

See Also: SET

POSITION (return file record sequence position)

POSITION(*sequence***)**

POSITION Identifies a record's unique position in the FILE.

sequence The label of a FILE, KEY, or INDEX declaration.

POSITION returns a STRING which identifies a record's unique position within the *sequence*. POSITION returns the position of the last record accessed in the file (the record currently in the file's record buffer). POSITION is used in conjunction with RESET to temporarily suspend and resume sequential file processing.

The value contained in the returned STRING and the length of that STRING are file driver dependent. As a general rule, for file systems that have record numbers, the size of the STRING returned by POSITION(file) is 4 bytes. The return string from POSITION(key) is 4 bytes plus the sum of the sizes of the fields in the key. For file systems that do not have record numbers the size of the STRING returned by POSITION(file) is the sum of the sizes of the fields in the Primary Key (the first KEY on the FILE that does not have the DUP or OPT attribute). The return string from POSITION(key) is the sum of the sizes of the fields in the Primary Key plus the sum of the sizes of the fields in the key.

Return Data Type: STRING

Example:

```
RecordQue     QUEUE,PRE(Dsp)
QueFields       LIKE(Trn:Record),PRE(Dsp)
              END
SavPosition   STRING(260)
  CODE
  SET(Trn:DateKey)                        !Top of file in keyed sequence
  LOOP                                    !Read all records in file
    NEXT(Trans)                           ! read a record sequentially
      IF ERRORCODE() THEN BREAK.
    RecordQue = Trn:Record                !Move record into queue
    ADD(RecordQue)                        ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20 OR EOF(Trans)
                                          !20 records in queue?
      SavPosition = POSITION(Trn:DateKey) !Save record position
      DO DisplayQue                       !Display the queue
      FREE(RecordQue)                     ! and free it
      RESET(Trn:DateKey,SavPosition)      !Reset the record pointer
      NEXT(Trans)                         ! and get record
  . .
```

See Also: RESET, REGET

RECORDS (return number of file or key records)

RECORDS(| *file* | **)**
| *key* |

RECORDS Returns the number of records.

file The label of a FILE declaration.

key The label of a KEY or INDEX declaration.

The **RECORDS** function returns the number of records in a *file* or *key*. Since the OPT attribute of a KEY or INDEX excludes "null" entries, RECORDS may return a smaller number for the KEY or INDEX than the FILE.

Return Data Type: LONG

Example:

```
SaveCount = RECORDS(Master)          !Save the record count
SaveNameCount = RECORDS(Nam:NameKey) !Number of records with names filled in
```

See Also: KEY, INDEX, OPT

SEND (send message to file driver)

SEND(*file*,*message***)**

SEND Sends a message to the file driver.

file The label of a FILE declaration. The FILE's DRIVER attribute identifies the file driver to receive the *message*.

message A string constant or variable containing the information to supply to the file driver.

The **SEND** function allows the program to pass any parameters specific to a file driver during program execution. Specific examples of valid SEND *messages* are listed in the file driver's documentation.

Return Data Type: STRING

Example:

```
FileCheck = SEND(ClarionFile,'RECOVER=120')
                              !Arm recovery process for a Clarion data file
```

Transaction Processing

A database has integrity when its data records contain valid data (data integrity) and its key fields accurately express the relationships between records and files (referential integrity). Database integrity can only be maintained through careful design and programming. If a particular data element's value is important, its entry field edit code must be written to detect and exclude bad data. If a Primary Key (Parent) record must not be deleted while related Foreign Key (Child) records exist, the program code must prevent the deletion.

Transaction Processing is one of the Clarion language tools that help programmers maintain database integrity. Transaction Processing is also commonly called "Transaction Tracking," "Transaction Logging," or an implementation of "commit boundaries." No matter what it is called, it is a programming technique that can significantly contribute to your maintenance of database integrity.

Clarion supports multiple file systems through its file driver technology. Some file systems do not support Transaction Processing, and others implement it using slightly different methods. Therefore, this discussion is about "generic" Transaction Processing. Each file driver's documentation should be consulted regarding support for, and implementation differences in its Transaction Processing.

Transaction Definition

A transaction may be defined as:

> A single logical event where multiple records are written to disk, and during which, any failure in those disk writes would compromise the integrity of the database.

A transaction can involve several records in one file, or one or more records in multiple files. The most important requirement of a transaction is: all of the records must be successfully written to disk, or none of them should be written at all. Therefore, Transaction Processing is "all or nothing."

Transaction Frame

A transaction always has an explicit beginning and ending, which defines the "transaction frame." One of the cardinal rules of Transaction Processing is to keep the transaction frame as small as possible. There are several reasons for this. Any system failure during a transaction would leave database integrity compromised. Therefore, the period of time during which database integrity could be compromised should be kept as small as possible to reduce the chances of this happening. A second reason is that for some

file systems in multi-user environments, exclusive access to the files is required for Transaction Processing. This means other users are denied access to the files during the transaction frame. Obviously, this is a strong argument for keeping the transaction frame small.

During the transaction frame, changes to the files included in the transaction are "tracked" or "logged" in Pre-Image files. Pre-Image files store just enough information about the transaction to restore the files to the state they were in before the transaction began, if necessary.

A transaction ends when it is either "rolled back" or "committed."

- If some error occurs during the transaction, the changes to the database stored in the Pre-Image files are rolled back. Once the changes have been rolled back, the state of the database is once again as it was before the transaction began.
- If no errors occur during the transaction, it is successful. Therefore, the transaction is committed and the changes are allowed to remain in the database.

Either way, the internal integrity of the database is still intact at the end of the transaction.

If a transaction is interrupted by a power outage or general system failure, the partially completed transaction must be rolled back, else the database integrity would be compromised. Each file system detects the need to roll back an incomplete transaction differently. File systems with the Client/Server type of architecture usually detect incomplete transactions when the file system's Server module is loaded on the network file server. Other types of file systems usually detect them when the data files that were in the transaction set are opened. However detected, **any incomplete transaction is rolled back automatically as soon as it is detected, either by the file system's engine or the Clarion driver for that file system.**

LOGOUT, COMMIT, ROLLBACK

There are three Clarion language statements which implement Transaction Processing: LOGOUT, COMMIT, and ROLLBACK.

The LOGOUT statement begins a transaction. It lists all the files that will be included in the transaction set—they all must use the same file driver. There is a very good reason for this.

Internally in each file system, the mechanics of transaction processing support have been designed in such a manner that there are no "windows of vulnerability." A "window of vulnerability" is an opening in the internal code where, if disaster (power outage or general system failure) were to

strike during that window, database integrity could be compromised without possibility of recovery. Each file system's support for transaction processing has been specifically designed to eliminate any such windows.

If you were allowed to include files from multiple file systems in a transaction set, a "window of vulnerability" would be created between the file drivers. Each file driver is a separate entity which internally handles transaction processing for its own files. If disaster were to strike during the COMMIT of such a transaction, any file driver in the transaction set that had not yet committed its transaction would think it had left an incomplete transaction. Incomplete transactions are automatically detected and rolled back. If this were to happen, the database integrity would be corrupted because part of the transaction would have been committed and part rolled back. Therefore, **all files in the transaction set must use the same file driver**.

COMMIT terminates a successful transaction. Only an explicit COMMIT statement terminates a successful transaction, there is no such thing as an "implied commit." If a file is closed, either explicitly (CLOSE) or implicitly (RUN, CHAIN, RUNSMALL, etc.) before the transaction is committed, the transaction is assumed to be incomplete. In most file systems, COMMIT deletes the Pre-Image files that would allow the transaction to be rolled back.

ROLLBACK terminates an unsuccessful transaction. It uses the information stored in the Pre-Image files to restore the files in the transaction set to their former state. In most file systems, ROLLBACK deletes the Pre-Image files after it has rolled back the transaction.

Multi-User Considerations

There are some considerations that must be taken into account when using Transaction Processing in a multi-user environment. The first thing to realize is that an Uninterruptable Power Supply (UPS) is absolutely required on the network's file server. This is because network operating systems "lie" to applications when they are told to write a record to disk. The operating system tells the application that the disk write was successful, when in fact, the record is still in the operating system's cache/buffer and has not yet physically made it to the disk.

Without a UPS on the file server, any power outage while the record is waiting in the cache/buffer disk could cause a loss of database integrity. The waiting period in the cache/buffer could be as long as thirty seconds, depending upon how the network is configured. This is an unconscionable length of time to allow as a "window of vulnerability." Therefore, a UPS on the file server is a requirement for Transaction Processing in a multi-user environment.

Some file systems require exclusive access (LOCK) for all the files in the transaction set. This is not a problem, because the LOGOUT statement automatically locks the files if necessary. The necessity of a file lock is determined by the *access mode* with which the file was opened. Any file opened with "Other's Access" set for Deny None or Deny Read is a shared resource for disk writes, and will be locked if the file system requires file locks. If the files are locked by LOGOUT, either ROLLBACK or COMMIT will UNLOCK them. This makes the Transaction Processing code exactly the same, whether used on single-user or multi-user systems.

It is important to know whether or not LOGOUT is automatically locking the files for you. If the files are automatically locked, and any program which accesses those same files also uses the HOLD statement, you could encounter the second form of "deadly embrace" discussed in the essay on Sharing Files. Your code must be written to detect any held records in the transaction, roll back the transaction, and allow the user a chance to re-try the transaction. You also need to do the same kind of concurrency checking discussed in the essay on Multi-User Considerations. If you are changing existing records in the transaction, you must detect any changes made by others to the records in the transaction.

COMMIT (terminate successful transaction)

COMMIT

The **COMMIT** statement terminates an active transaction. Execution of a COMMIT statement assumes that the transaction was completely successful and no ROLLBACK is necessary. Once COMMIT has been executed, ROLLBACK of the transaction is impossible.

COMMIT informs the file driver involved in the transaction that the temporary files containing the information necessary to restore the database to its previous state may be deleted. The file driver then performs the actions necessary to its file system to successfully terminate a transaction.

Example:

```
  LOGOUT(.1,OrderHeader,OrderDetail)    !Begin Transaction
    DO ErrHandler                       ! always check for errors
  ADD(OrderHeader)                      !Add Parent record
    DO ErrHandler                       ! always check for errors
  LOOP X# = 1 TO RECORDS(DetailQue)     !Process stored detail records
    GET(DetailQue,X#)                   ! Get one from the QUEUE
      DO ErrHandler                     ! always check for errors
    Det:Record = DetailQue              ! Assign to record buffer
    ADD(OrderDetail)                    ! and add it to the file
      DO ErrHandler                     ! always check for errors
  END
  COMMIT                                !Terminate successful transaction

ErrHandler ROUTINE                      !Error routine
  IF NOT ERRORCODE() THEN EXIT.         !Bail out if no error
  ROLLBACK                              !Rollback the aborted transaction
  BEEP                                  !Alert the user
  MESSAGE('Transaction Error - ' & ERROR())
  RETURN                                ! and get out
```

See Also: LOGOUT, ROLLBACK

LOGOUT (begin transaction)

LOGOUT(*timeout* ,*file* [,*file*,...,*file*]**)**

LOGOUT	Initiates transaction processing.
timeout	A numeric constant or variable which specifies the number of seconds to attempt to begin the transaction for a *file* before aborting the transaction and returning an error.
file	The label of a FILE declaration. There may be multiple *file* parameters, separated by commas, in the parameter list. All *files* that will be in the transaction set must be listed.

The **LOGOUT** statement initiates transaction processing for a specified set of *files*. All *files* in the transaction set must have the same file driver. LOGOUT informs the file driver that a transaction is beginning. The file driver then performs the actions necessary to that file system to initiate transaction processing for the specified set of *files*. If the file system requires that the *files* be locked for transaction processing, LOGOUT automatically locks the *files.*

Only one LOGOUT transaction may be active at a time. A second LOGOUT statement without a prior COMMIT or ROLLBACK halts the program with an error message, returning the user to DOS.

Errors Posted: 32 File Is Already Locked

Example:

```
    LOGOUT(.1,OrderHeader,OrderDetail)      !Begin Transaction
      DO ErrHandler                         ! always check for errors
    ADD(OrderHeader)                        !Add Parent record
      DO ErrHandler                         ! always check for errors
    LOOP X# = 1 TO RECORDS(DetailQue)       !Process stored detail records
      GET(DetailQue,X#)                     ! Get one from the QUEUE
        DO ErrHandler                       ! always check for errors
      Det:Record = DetailQue                ! Assign to record buffer
      ADD(OrderDetail)                      ! and add it to the file
        DO ErrHandler                       ! always check for errors
    END
    COMMIT                                  !Terminate successful transaction

ErrHandler ROUTINE                          !Error routine
    IF NOT ERRORCODE() THEN EXIT.           !Bail out if no error
    ROLLBACK                                !Rollback the aborted transaction
    BEEP                                    !Alert the user
    MESSAGE('Transaction Error - ' & ERROR())
    RETURN                                  ! and get out
```

See Also: COMMIT, ROLLBACK

ROLLBACK (terminate unsuccessful transaction)

ROLLBACK

The **ROLLBACK** statement terminates an active transaction. Execution of a ROLLBACK statement assumes that the transaction was unsuccessful and the database must be restored to the state it was in before the transaction began.

ROLLBACK informs the file driver involved in the transaction that the temporary files containing the information necessary to restore the database to its previous state must be used to restore the database. The file driver then performs the actions necessary to its file system to roll back the transaction.

Example:

```
  LOGOUT(.1,OrderHeader,OrderDetail)      !Begin Transaction
    DO ErrHandler                         ! always check for errors
  ADD(OrderHeader)                        !Add Parent record
    DO ErrHandler                         ! always check for errors
  LOOP X# = 1 TO RECORDS(DetailQue)       !Process stored detail records
    GET(DetailQue,X#)                     ! Get one from the QUEUE
      DO ErrHandler                       ! always check for errors
    Det:Record = DetailQue                ! Assign to record buffer
    ADD(OrderDetail)                      ! and add it to the file
      DO ErrHandler                       ! always check for errors
  END
  COMMIT                                  !Terminate successful transaction

ErrHandler ROUTINE                        !Error routine
  IF NOT ERRORCODE() THEN EXIT.           !Bail out if no error
  ROLLBACK                                !Rollback the aborted transaction
  BEEP                                    !Alert the user
  MESSAGE('Transaction Error - ' & ERROR())
  RETURN                                  ! and get out
```

See Also: LOGOUT, COMMIT

Null Data Processing

The concept of a null "value" in a field of a FILE or VIEW indicates that the user has never entered data into the field. Null actually means "value not known" for the field. This is completely different from a blank or zero value, and makes it possible to detect the difference between a field which has never had data, and a field which has a (true) blank or zero value.

In expressions, null does not equal blank or zero. Therefore, any expression which compares the value of a field from a FILE or VIEW with another value will always evaluate as unknown if the field is null. This is true even if the value of both elements in the expression are unknown (null) values. For example, the conditional expression Pre:Field1 = Pre:Field2 will evaluate as true only if both fields contain known values. If both fields are null, the result of the expression is also unknown.

```
Known = Known       !Evaluates as True or False
Known = Unknown     !Evaluates as unknown
Unknown = Unknown   !Evaluates as unknown
Unknown <> 10       !Evaluates as unknown
1 + Unknown         !Evaluates as unknown
```

The only four exceptions to this rule are boolean expressions using OR and AND where only one portion of the entire expression in unknown and the other protion of the expression meets the expression criteria:

```
Unknown OR True       !Evaluates as True
True OR Unknown       !Evaluates as True
Unknown AND False     !Evaluates as False
False AND Unknown     !Evaluates as False
```

Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver. Some file drivers support the null field concept (SQL drivers, for the most part), while others do not. Consult the documentation for the specific file driver to determine whether or not your file system's driver supports nulls.

NULL (return null file field)

NULL(*field*)

NULL	Determines null "value" of a *field*.
field	The label (including prefix) of a field in a FILE or VIEW structure.

The **NULL** function returns a non-zero value (true) if the *field* is null, and zero (false) if the *field* contains any known value (including blank or zero). Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)   !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record      RECORD
AcctNumber    LONG
OrderNumber   LONG
Name          STRING(20)
Addr          STRING(20)
CSZ           STRING(35)
            . .

Header    FILE,DRIVER('Clarion'),PRE(Hea)   !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
Record      RECORD
AcctNumber    LONG
OrderNumber   LONG
ShipToName    STRING(20)
ShipToAddr    STRING(20)
ShipToCSZ     STRING(35)
            . .

  CODE
  OPEN(Header)
  OPEN(Customer)
  SET(Hea:AcctKey)
  LOOP
    NEXT(Header)
      IF ERRORCODE() THEN BREAK.
    IF NULL(Hea:ShipToName)               !Check for null ship-to address
      Cus:AcctNumber = Hea:AcctNumber
      GET(Customer,Cus:AcctKey)           !Get Customer record
        IF ERRORCODE() THEN CLEAR(Cus:Record).
      Hea:ShipToName  = Cus:Name          ! and assign customer address
      Hea:ShipToAddr  = Cus:Addr          ! as the ship-to address
      Hea:ShipToCSZ  = Cus:CSZ
    END
    PUT(Header)                           !Put Header record back
  END
```

SETNULL (set file field null)

SETNULL(*field*)

SETNULL Assigns null "value" to a *field*.

field The label (including prefix) of a field in a FILE or VIEW structure.

The **SETNULL** statement assigns a null "value" to a *field* in a FILE or VIEW structure. Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record      RECORD
AcctNumber    LONG
OrderNumber   LONG
Name          STRING(20)
Addr          STRING(20)
CSZ           STRING(35)
            . .

Header    FILE,DRIVER('Clarion'),PRE(Hea)   !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber    LONG
OrderNumber   LONG
ShipToName    STRING(20)
ShipToAddr    STRING(20)
ShipToCSZ     STRING(35)
            . .

  CODE
  OPEN(Header)
  OPEN(Customer)
  SET(Hea:AcctKey)
  LOOP
    NEXT(Header)
      IF ERRORCODE() THEN BREAK.
    Cus:AcctNumber = Hea:AcctNumber
    GET(Customer,Cus:AcctKey)                !Get Customer record
      IF ERRORCODE() THEN CLEAR(Cus:Record).
    IF NOT NULL(Hea:ShipToName) AND Hea:ShipToName = Cus:Name
                                             !Check ship-to address
      SETNULL(Hea:ShipToName)                ! and assign null "values"
      SETNULL(Hea:ShipToAddr)                ! to ship-to address
      SETNULL(Hea:ShipToCSZ)
    END
    PUT(Header)                              !Put Header record back
  END
```

SETNONNULL (set file field non-null)

SETNONNULL(*field*)

SETNONNULL	Assigns non-null value (blank or zero) to a *field*.
field	The label (including prefix) of a field in a FILE or VIEW structure.

The **SETNONNULL** statement assigns a non-null value (blank or zero) to a *field* in a FILE or VIEW structure. Support for null “values” in a FILE or VIEW is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus)   !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
            . .

Header      FILE,DRIVER('Clarion'),PRE(Hea)   !Declare header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey      KEY(Hea:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
            . .

  CODE
  OPEN(Header)
  OPEN(Customer)
  SET(Hea:AcctKey)
  LOOP
    NEXT(Header)
      IF ERRORCODE() THEN BREAK.
    Cus:AcctNumber = Hea:AcctNumber
    GET(Customer,Cus:AcctKey)                    !Get Customer record
      IF ERRORCODE() THEN CLEAR(Cus:Record).
    IF NULL(Hea:ShipToName) OR Hea:ShipToName = Cus:Name
                                                 !Check same ship-to address
      Hea:ShipToName = 'Same as Customer Address' ! flag the record
      SETNONNULL(Hea:ShipToAddr)                 ! and blank out ship-to address
      SETNONNULL(Hea:ShipToCSZ)
    END
    PUT(Header)                                  !Put Header record back
  END
```

See Also: NULL, SETNULL

Internationalization

Environment Files

An environment file contains internationalization settings for an application. On program initialization, the Clarion run-time library attempts to locate an environment file with the same name and location as your application's program file (*appname*.ENV). If an environment file is not found, the run-time library defaults to standard English/ASCII. You can also use these settings to specify internationalization issues for the Clarion environment by creating a CW.ENV file (the Database Manager uses these settings when displaying data files).

The .ENV file is compatible with the .INI files used by Clarion for DOS (both versions 3 and 3.1) if the CLACHARSET is set to OEM, because Clarion for DOS .INI files are generally written using OEM ASCII, not the ANSI character set.

The LOCALE procedure can be used to load environment files at run-time to dynamically change the international settings. LOCALE can also be used to set individual entries. International support is dependent on support in the File Driver (generally for the OEM attribute); consult the File Driver documentation for information on international support in specific drivers.

The following settings can be set in an environment file:

CLACHARSET=WINDOWS
CLACHARSET=OEM

This determines the character set used by the entries in the .ENV file. WINDOWS is the default if this setting is omitted from the environment file. Use the OEM setting if you are using a DOS editor to edit the .ENV file, or if it has to be compatible with Clarion for DOS. Otherwise, specify WINDOWS or omit the entry. This should always be the first setting in the environment file.

CLACOLSEQ=WINDOWS
CLACOLSEQ=*"string"*

Specifies a specific collating sequence for use at run-time. This collating sequence is used for building KEY and INDEX files, as well as for sorting QUEUEs and all string/character comparisons.

If the WINDOWS setting is used, then the default collation sequence is defined by Windows' Country setting (in the Control Panel). If this entry is omitted from the environment file, then the default ANSI ordering is used, not the windows default.

Using the WINDOWS setting, the ordering can 'interleave' characters of differing case (AaBbCc ...), so code such as

```
CASE SomeString[1]
OF 'A' TO 'Z'
```

includes 'a' TO 'y' as well. Use the ISUPPER and ISLOWER functions in preference to this kind of code if WINDOWS (or other non-default) collation sequences are used.

In addition to the WINDOWS setting, you may specify a *string* of characters (in double quotes) to explicitly define the collation sequence to use. Only those characters that need to have their sort order specified need be included; all other characters not listed remain in their same relative order. For example, if CLACOLSEQ="CA" is specified for the standard English sort (ABCD ...) the resulting sort order is "CBAD." This is a change from the Clarion for DOS versions of this setting that needed exactly 222 characters, but it is backward compatible.

NOTE: You should always read and write files using the same collation sequence. Using a different sequence may result in keys becoming out of order and records becoming inaccessible. Specifying CLACOLSEQ=WINDOWS means that the collation sequence may change if the user changes the Country in Windows' Control Panel.

CLAAMPM=WINDOWS
CLAAMPM=*"AMstring","PMstring"*

This specifies the text used to indicate AM or PM as a part of a time display field. The WINDOWS setting specifies use of the AM/PM strings set up in the Windows Control Panel. The *AMstring* and *PMstring* settings are the same as in Clarion for DOS, except that they take notice of the setting of CLACHARSET.

CLAMONTH="Month1","Month2", ... ,"Month12"

Specifies the text returned by functions and picture formats involving the month full name.

CLAMON="AbbrevMonth1","AbbrevMonth2", ... ,"AbbrevMonth12"

Specifies the text returned by functions and picture formats involving the abbreviated month name.

CLADIGRAPH="*DigraphChar1Char2, ...* "

This allows *Digraph* characters to collate correctly. A *Digraph* is a single logical character that is a combination of two characters (*Char1* and *Char2*). The *Digraph* is collated as the two characters that combine to create

it. They are more common in non-English languages. For example, with CLADIGRAPH="ÆAe,æae" specified, the word “Jæger” sorts before “Jager” (since “Jae” comes before “Jag”).

Multiple *DigraphChar1Char2* combinations may be defined, separated by commas. This setting takes notice of the CLACHARSET setting.

CLACASE=WINDOWS
CLACASE=*”UpperString”,”LowerString”*

Allows you to specify upper and lower case letter pairs.

The WINDOWS setting uses the default upper/lower case pair sets as defined by the Windows Country setting (in the Control Panel). If this entry is omitted from the environment file, then the default ANSI ordering is used, not the windows default.

The *UpperString* and *LowerString* parameters specify a set of uppercase characters and each one’s lowercase equivalent. The length of the *UpperString* and *LowerString* parameters must be equal. CLACASE takes notice of the setting of CLACHARSET. ANSI characters less than 127 are not affected.

CLABUTTON=”OK”,”&Yes”,”&No”,”&Abort”,”&Ignore”,”&Retry”,”Cancel”,”&Help”

This defines the text used by the buttons of the MESSAGE function. The text is specified as a list of comma separated strings in the following order: OK, YES, NO, ABORT, RETRY, IGNORE, CANCEL, HELP. The default is as specified above.

CLAMSG*errornumber=”ErrorMessage”*

This allows run-time error messages to be overridden with translated strings. The *errornumber* is a standard Clarion error code number appended to CLAMSG. *ErrorMessage* is the string value used to replace that error number’s default message. For example, CLAMSG2=”No File Found” makes ”No File Found” the return value of the ERROR() function when ERRORCODE() = 2.

Example:

```
CLACHARSET=WINDOWS
CLACOLSEQ=”AÄÅÆaàáâãåæBbCÇçDdEÉeèéêëFfGgHhIìíîïJjKkLlMmNÑnñOÖoòóôõPpQqRrSsßTtUÜuùúûüVvWwXxYyZzÿ”
CLAAMPM=”AM”,”PM”
CLAMONTH=”January”,”February”,”March”,”April”,”May”,”June”,”July”,”August”,”September”,”October”,”November”,”December”
CLAMON=”Jan”,”Feb”,”Mar”,”Apr”,”May”,”Jun”,”Jul”,”Aug”,”Sep”,”Oct”,”Nov”,”Dec”
CLADIGRAPH="ÆAe,æae"
CLACASE=”ÄÅÆÇÉÑÖÜ”,”äåæçéñöü”
CLABUTTON=”OK”,”&Si”,”&No”,”&Abortar”,”&Ignora”,”&Volveratratar”,”Cancelar”,”&Ayuda”
CLAMSG2=“No File Found”
```

CONVERTANSITOOEM (convert ANSI strings to ASCII)

CONVERTANSITOOEM(*string* **)**

CONVERTANSITOOEM
Translates ANSI strings to OEM ASCII.

string The label of the string to convert. This may be a single variable or a any structure that is treated as a GROUP (RECORD, QUEUE, etc.).

The **CONVERTANSITOOEM** statement translates either a single string or the strings within a GROUP from the ANSI (Windows display) character set into the OEM character set (ASCII with extra characters defined by the active code page).

This procedure is not required on data files if the OEM attribute is set on the file.

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus)   !Declare file without OEM attribute
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
              . .
Win WINDOW,SYSTEM
      STRING(@s20),USE(Cus:Name)
    END
  CODE
  OPEN(Customer)
  SET(Customer)
  NEXT(Customer)
  CONVERTOEMTOANSI(Cus:Record)  !Convert all strings from ASCII to ANSI
  OPEN(Win)
  ACCEPT
    !Process window controls
  END
  CONVERTANSITOOEM(Cus:Record)  !Convert back to ASCII from ANSI
  PUT(Customer)
```

See Also: CONVERTOEMTOANSI

CONVERTOEMTOANSI (convert ASCII strings to ANSI)

CONVERTOEMTOANSI(*string* **)**

CONVERTOEMTOANSI
Translates OEM ASCII strings to ANSI.

string The label of the string to convert. This may be a single variable or a any structure that is treated as a GROUP (RECORD, QUEUE, etc.).

The **CONVERTOEMTOANSI** statement translates either a single string or the strings within a GROUP from the the OEM character set (ASCII with extra characters defined by the active code page) into ANSI (Windows display) character set.

This procedure is not required on data files if the OEM attribute is set on the file.

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus)   !Declare file without OEM attribute
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
              . .
Win WINDOW,SYSTEM
      STRING(@s20),USE(Cus:Name)
    END
  CODE
  OPEN(Customer)
  SET(Customer)
  NEXT(Customer)
  CONVERTOEMTOANSI(Cus:Record)  !Convert all strings from ASCII to ANSI
  OPEN(Win)
  ACCEPT
    !Process window controls
  END
  CONVERTANSITOOEM(Cus:Record)  !Convert back to ASCII from ANSI
  PUT(Customer)
```

See Also: CONVERTANSITOOEM

ISALPHA (return alphabetic character)

ISALPHA(*string* **)**

ISALPHA	Returns whether the *string* passed to it contains an alphabetic character.
string	The label of the character string to test. If the *string* contains more than one character, only the first character is tested.

The ISALPHA function returns TRUE if the *string* passed to it is alphabetic (an upper or lower case letter) and false otherwise. This is independent of the language and collation sequence.

Return Data Type: LONG

Example:

```
SomeString  STRING(1)
  CODE
  SomeString = 'A'    !ISALPHA returns true
  IF ISALPHA(SomeString)
    X#= MESSAGE('Alpha string')
  END
  SomeString = '1'    !ISALPHA returns false
  IF ISALPHA(SomeString)
    X#= MESSAGE('Alpha string')
  ELSE
    X#= MESSAGE('Not Alpha string')
  END
```

See Also: ISUPPER, ISLOWER

ISLOWER (return lower case character)

ISLOWER(*string* **)**

ISLOWER Returns whether the *string* passed to it contains a lower case alphabetic character.

string The label of the string to test. If the *string* contains more than one character, only the first character is tested.

The ISLOWER function returns TRUE if the *string* passed to it is a lower case letter and false otherwise. This is independent of the language and collation sequence.

Return Data Type: LONG

Example:

```
SomeString  STRING(1)
  CODE
  SomeString = 'a'    !ISLOWER returns true
  IF ISLOWER(SomeString)
    X#= MESSAGE('Lower case string')
  END
  SomeString = 'A'    !ISLOWER returns false
  IF ISLOWER(SomeString)
    X#= MESSAGE('Lower case string')
  ELSE
    X#= MESSAGE('Not lower case string')
  END
```

See Also: ISUPPER, ISALPHA

ISUPPER (return upper case character)

ISUPPER(*string* **)**

ISUPPER Returns whether the *string* passed to it contains an upper case alphabetic character.

string The label of the string to test. If the *string* contains more than one character, only the first character is tested.

The ISUPPER function returns TRUE if the *string* passed to it is an upper case letter and false otherwise. This is independent of the language and collation sequence.

Return Data Type: LONG

Example:

```
SomeString  STRING(1)
  CODE
  SomeString = 'A'    !ISUPPER returns true
  IF ISUPPER(SomeString)
    X#= MESSAGE('Upper case string')
  END
  SomeString = 'a'    !ISUPPER returns false
  IF ISUPPER(SomeString)
    X#= MESSAGE('Upper case string')
  ELSE
    X#= MESSAGE('Not upper case string')
  END
```

See Also: ISLOWER, ISALPHA

LOCALE (load environment file)

LOCALE(| *file* **|)**
| *setting, value* |

LOCALE Allows the user to load a specific environment file (.ENV) at run-time and also to set individual environment settings.

file A string constant or variable containing the name (including extension) of the environment file (.ENV) to load, or the keyword WINDOWS. This may be a fully-qualified DOS pathname.

setting A string constant or variable containing the name of the environment variable to set. Valid choices are listed under the *Environment Files* section.

value A string constant or variable containing the environment variable setting.

The **LOCALE** procedure allows the user to load a specific environment file (.ENV) at run-time and also to set individual environment settings. This allows an application to load another file to override the default *appname*.ENV file, or to specify individual environment file settings when no environment file exists.

The WINDOWS keyword as the *file* parameter specifies use of Windows' default values for CLACOLSEQ, CLACASE and CLAAMPM. When specifying individual *settings*, the *value* parameter does not require double quotes around each individual item in the *value* string, unlike the syntax required in an .ENV file.

Errors Posted:

02 File Not Found
05 Access Denied

Example:

```
LOCALE('MY.ENV')                        !Load an environment file
LOCALE('WINDOWS')                       !Set default CLACOLSEQ, CLACASE and CLAAMPM
LOCALE('CLABUTTON','OK,&Si,&No,&Abortar,&Ignora,&Volveratratar,Cancelar,&Ayuda')
                                        !Set CLABUTTON to Spanish
LOCALE('CLACOLSEQ','AÄÅÆaàáâãåæBbCÇcçDdEÉeèéêëFfGgHhIìíîïJjKkLlMmNÑnñOÖoòóôõPpQqRrSsßTtUÜuùúûüVvWwXxYyZzÿ')
                                        !Set the collating sequence
LOCALE('CLACASE','ÄÅÆÇÉÑÖÜ,äåæçéñòü')  !Set upper/lower case pairs
LOCALE('CLAMSG2','No File Found')       !Set ERROR() message for ERRORCODE()=2
```

See Also: Environment Files

Data File Processing

Custom database applications, by definition, store data in files. Getting data into those files, and processing it for some kind of meaningful output, is the primary purpose of any database application. This essay is a discussion of the Clarion language tools which allow the programmer to access and process data files.

File Access Methods

Generally speaking, records are put into data files at the end of the file in the sequence in which they are added (this is not always true, but is usually true). This creates the "physical, record-number order" of the file—the physical order in which the records appear within the file. This physical order does not necessarily correspond to any meaningful or useful sequence.

There are two ways to access records within a file: sequential access, and random access. Sequential access means you retrieve a number of records in some specified sequence, processing each record in order. Random access means you retrieve and process one specific record. Both of these access methods are used in almost every business database application.

If you only need to access records sequentially in their physical, record-number order, nothing more than the data file is needed. If you need to randomly access a record, and you know exactly which position it occupies in the file (its record number), the same thing is true. However, for most applications, these constraints would be too limiting.

KEY and INDEX

The Clarion KEY and INDEX declarations create alternate sort orders for the records in the file. These allow sequential or random access to a data file in some order other than the physical, record-number order. The order is determined by the component fields that make up the KEY or INDEX. Each KEY or INDEX component may be in ascending or descending order.

The main difference between KEY and INDEX lies in the fact that a KEY is dynamically maintained. Every time a record is added, changed, or deleted, the KEY is also updated. Since it is always kept current, a KEY should be used for sort orders that are frequently used in the application.

An INDEX is not maintained and must be rebuilt immediately before it is used to ensure that it accurately reflects the current state of the file. The BUILD statement is used to rebuild an INDEX. Because of the time factor in rebuilding, and the fact that exclusive file access is required for the BUILD, an INDEX should be used for sort orders that are infrequently used.

One special form of INDEX is the ”dynamic” INDEX. This is an INDEX whose component fields are not declared in the file definition. The component fields of a “dynamic” INDEX are declared at run-time in the BUILD statement. Unlike a “static” INDEX, you may BUILD a “dynamic” INDEX with the file open in any access mode. The advantage should be immediately obvious: end-user-definable sort orders.

```
Sample    FILE,DRIVER('Clarion'),PRE(Sam)
Field1Key   KEY(Sam:Field1)           !KEY on Field 1
Field2Ndx   INDEX(Sam:Field2)         !Static INDEX on Field2
DynNdx      INDEX()                   !Dynamic INDEX
Record      RECORD
Field1        LONG
Field2        STRING(10)
Field3        DECIMAL(7,2)
          . .
  CODE
  OPEN(Sample,42h)                          !Open read/write deny none
  LOCK(Sample)                              !Lock for exclusive access
  BUILD(Sam:Field2Ndx)                      !  to Build the INDEX
  UNLOCK(Sample)                            !    then Unlock the file
  BUILD(Sam:DynNdx,'-Sam:Field1,+Sam:Field2') !Build the dynamic INDEX
```

In this example, the KEY on Sam:Field1 will always be current, the INDEX on Sam:Field2 is built when the file is opened and LOCKed (exclusive access is required). The “dynamic” INDEX is built at run-time in descending Sam:Field1 and ascending Sam:Field2 sort order. Of course, it could be built in any sort order possible for the file.

Other than their maintenance, KEY and INDEX are functionally equivalent. They share the same type of file format and may be used interchangeably in all executable file access statements which require a KEY or INDEX parameter. To simplify this discussion, wherever the phrase “KEY and/or INDEX” would be appropriate, it is replaced with the generic term *index*. All references to *index* apply equally to both KEYs and INDEXes, unless otherwise noted.

Sequential File Access

There are three Clarion statements which perform sequential file access: SET, NEXT, and PREVIOUS. The SET statement initializes the sequential processing—it does not read a record. The NEXT and PREVIOUS statements read the records in ascending (NEXT) or descending (PREVIOUS) order within the sequence established by SET.

The SET statement is the ruling element in sequential file processing. A SET statement must come before NEXT or PREVIOUS to initialize the starting point and sequence in which the records will be read. Usually, the SET statement is the last executable statement before the LOOP structure which sequentially processes the records in the file. The NEXT or PREVIOUS is then the first statement within the LOOP, as in this example code using the previous example file definition:

```
  SET(Sam:Field1Key)              !Set to top of file in KEY order
  LOOP                            !Loop until end of the Sample file
    NEXT(Sample)                  !Read each record in turn
    IF ERRORCODE() THEN BREAK.    !Break at end of file
      !record processing statements
  END                             !End loop
```

There are seven forms of the SET statement listed in the Language Reference Manual. These essentially break down into two categories: three starting points for physical record-number order access, and four starting points for *indexed* order access.

Physical Order	*Indexed* Order
Top/Bottom of File	Top/Bottom of File
Physical Record Number	*Index* Record Number
Index Value	*Index* Value
	Index Value and Physical Record Number

SET initializes the sequential processing record pointer, and it employs a type of "fuzzy logic." When you SET to the Top/Bottom of the file, the record pointer is not actually pointing at either. If you issue a NEXT after the SET, you read records forward from the beginning of the file. If you issue a PREVIOUS instead, you read records backwards from the end of the file. Once you have issued the NEXT or PREVIOUS to begin reading records in one direction, you cannot go back across the Top/Bottom of the file without another SET.

The same "fuzzy logic" is active when you SET to an *index* value. If SET finds a record containing an exact match to that *index* value, it points to that specific record. In this case, either NEXT or PREVIOUS would read the same record. If, however, there is no exact match to the *index* value, SET points "between" the last record in sequence containing a value less than (or greater than, in a descending *index*) the *index* value and the next record in sequence containing a value greater than (or less than, in a descending *index*) the *index* value. In this case, NEXT and PREVIOUS would not read the same record. NEXT would read the following record in the *index* sequence, PREVIOUS would read the prior record in the *index* sequence.

The advantage of this "fuzzy logic" lies in its use with a multiple component *index*, as in this example.

```
Sample    FILE,DRIVER('Clarion'),PRE(Sam)
FieldsKey   KEY(Sam:Field1,Sam:Field2),DUP  !KEY on Field 1 and Field 2
Record      RECORD
Field1        LONG
Field2        STRING(10)
Field3        DECIMAL(7,2)
            . .
  CODE
  OPEN(Sample,42h)                  !Open read/write deny none
  CLEAR(Sam:Record)                 !Clear the record buffer
  Sam:Field1 = 10                   !Initialize first KEY component
  SET(Sam:FieldsKey,Sam:FieldsKey)  !KEY sequence, start at 10-blank
  LOOP                              !Process each record
    NEXT(Sample)                    !   one at a time
    IF ERRORCODE() THEN BREAK.      !Break at end of file
    IF Sam:Field1 <> 10             !Check for end of group
```

```
      BREAK                          !  if so, get out of process loop
    END
      !record processing statements
  END                                !End process loop
```

This code first clears the record buffer, assigning zeroes to Sam:Field1 and Sam:Field3, and blanks to Sam:Field2. The first component field of Sam:FieldsKey is initialized to the value that must be in the records you need to process. The SET statement sets up sequential processing in *indexed* order, starting at the *index* value—in this case a value of 10 in Sam:Field1 and blanks in Sam:Field2.

```
Sample File Records:  Index Record #  Field1  Field2  Field3
                                  1       5     ABC   14.52
                                  2       5     DEF   14.52
Record Pointer After SET >>
                                  3      10     ABC   14.52
                                  4      10     ABC   29.04
                                  5      10     DEF   14.52
                                  6      15     ABC   14.52
                                  7      15     DEF   14.52
```

SET leaves the record pointer positioned as shown above because there is no exact match. Record 2's value 5-DEF is less than 10-blank, and record 3's value 10-ABC is greater than 10-blank, therefore the record pointer is left "between" the two. The first time through the LOOP, NEXT reads record number 3. The IF statement terminates the processing loop after NEXT reads record 6.

There is a distinct difference between the Physical Record Number and the *Index* Record Number. The Physical Record Number is the relative physical position within the data file as returned by the POINTER(Label of a File) function. The *Index* Record Number is the relative record position within the *index* sequence as returned by the POINTER(Label of an *Index*) function.

In physical order, the same file might look like this (of course, the physical and *index* record numbers are not stored in the data file):

```
Sample File:  Physical Record #  Index Record #  Field1  Field2  Field3
                              1               3      10     ABC   14.52
                              2               6      15     ABC   14.52
                              3               5      10     DEF   14.52
                              4               2       5     DEF   14.52
                              5               4      10     ABC   29.04
                              6               7      15     DEF   14.52
                              7               1       5     ABC   14.52
```

The forms of SET that use Record Numbers as the starting point look very similar, therefore you need to be very clear about which you are using (Physical vs. *Index*).

```
SET(Sample,1)     !Physical Order, SETs to physical rec 1, index rec 3

SET(Sam:FieldsKey,1)
                  !Index order, SETs to index rec 1, physical rec 7
```

```
Sam:Field1 = 10
Sam:Field2 = 'ABC'
SET(Sam:FieldsKey,Sam:FieldsKey,5)
                  !Index order, SETs to index rec 4, physical rec 5
```

This last form of SET allows you to SET to a specific record within a sequence of records which contain duplicate *index* field values. It searches the duplicate *index* entries for an *index* entry which points to the Physical Record Number specified as the third parameter. This is useful in files where there are multiple records with duplicate *index* values and you need to begin processing at one specific record within those duplicates.

Random File Access

There is only one Clarion statement which performs random access to individual records within a file—the GET statement. Unlike SET, GET either reads the record you attempt to retrieve, or returns an error. There is no "fuzzy logic" with GET.

There are three forms of the GET statement. They allow you to retrieve a record based on an *index* value, Physical Record Number, or *Index* Record Number.

```
Sam:Field1 = 15
Sam:Field2 = 'ABC'
GET(Sample,Sam:FieldsKey)     !GETs index rec 6, physical rec 2
GET(Sample,1)                 !GETs physical rec 1, index rec 3
GET(Sam:FieldsKey,1)          !GETs index rec 1, physical rec 7
```

The first GET example retrieves the first record in the *index* order which contains the values in the *index* component fields at the time the GET is issued. The second example retrieves the first record in the file in Physical Record Number order. The third retrieves the first record in the file in *Index* Record Number order.

GET always looks for an exact match to the *index* value and returns an error if it does not find one. Therefore, all component fields of a multiple component *index* must be initialized before issuing a GET.

GET is completely independent of SET/NEXT or SET/PREVIOUS sequential processing. This means that a GET into a file which is being sequentially processed does not change the record pointer for sequential processing.

```
SET(Sam:FieldsKey)               !Set to top of file
LOOP                             !Process each record in index order
  NEXT(Sample)                   !Gets each sequential record
  IF ERRORCODE() THEN BREAK.     !Break at end of file
     !sequential record processing statements
  GET(Sam:FieldsKey,1)           !Gets the first record in index order
     !random access record processing statements
END
```

This example code processes through the entire file in *index* order. After each sequential record is processed, the first record in *index* order is retrieved and processed. This does not affect the sequence, therefore NEXT will progress through the file, despite the GET of the first record every time through the loop.

Summary

- Sequential Access and Random Access are the two methods used to retrieve records from a file.
- The Clarion KEY and INDEX declarations define alternate sort orders of the file in which they are declared.
- A KEY is dynamically maintained and is always ready for use. An INDEX is not maintained and must be built before use.
- A "dynamic" INDEX allows sort orders to be defined at run-time.
- The SET statement initializes the order and starting point of sequential processing. A SET is required before the first NEXT or PREVIOUS.
- SET employs "fuzzy logic" to determine the starting point. It either points to a specific record, or "between" records at the position where it determined no record fit the starting point parameters it was given.
- Physical and *Index* Record Numbers are very different and must not be confused with each other.
- The GET statement performs random record access within a file.
- GET is completely independent of the SET/NEXT and SET/PREVIOUS sequential record processing.

Multi-User Considerations

The world of database applications programming is rapidly heading towards networking. Stand-alone applications are expanding into multi-user environments as more companies connect their PCs to Local Area Networks (LANs). Mainframe applications in large companies are being "right-sized" and re-written for LAN operation. With the emergence of multi-threading, multi-tasking operating systems for PCs, even standalone computers need applications that are written with multi-user shared-access considerations in mind. This essay is a discussion of the Clarion language tools provided to write applications specifically designed for use in multi-user environments.

Opening Files

Before any data file can be processed, it must first be opened. The OPEN and SHARE statements provide this function. OPEN and SHARE are functionally equivalent, the only difference between the two is the default value of the second (access mode) parameter of each.

The access mode specifies the type of access the user opening the file receives, and the type of access allowed to other users of the file. These two values are added together to create the DOS Access Code for the file. The access mode values are:

	Access	Dec.	Hex.
User's Access:	Read Only	0	0h
`	Write Only	1	1h
	Read/Write	2	2h
Other's Access:	Deny All	16	10h
	Deny Write	32	20h
	Deny Read	48	30h
	Deny None	64	40h

The OPEN statement's default access mode is Read/Write Deny Write (22h), which only allows exclusive (single-user) disk write access to the user opening the file. The SHARE statement's default access mode is Read/Write Deny None (42h), allowing non-exclusive (multi-user) access to anybody who opens the file. Either OPEN or SHARE may open the file in any of the possible access modes.

```
OPEN(file)          !Open Read/Write Deny Write
OPEN(file,22h)      !Open Read/Write Deny Write
SHARE(file,22h)     !Open Read/Write Deny Write

SHARE(file)         !Open Read/Write Deny None
SHARE(file,42h)     !Open Read/Write Deny None
OPEN(file,42h)      !Open Read/Write Deny None

OPEN(file,40h)      !Open Read Only Deny None
SHARE(file,40h)     !Open Read Only Deny None
```

These examples demonstrate the three most commonly used access modes. For multi-user applications, the most common access mode is Read/Write

Deny None (42h), which permits all users complete access to the file. Read Only Deny None (40h) is usually used in multi-user situations where the user will not update the file (a lookup-only file) but there may be some other user who may need to write to that file.

Concurrency Checking

The biggest consideration to keep in mind about multi-user access to files is the possibility that several users could be updating the same record at the same time. A process known as "concurrency checking" prevents the data file from being corrupted by multiple user updates to the same record. Concurrency checking means determining that the record on disk, which is about to be overwritten, still contains the same values it did when it was retrieved for update.

Obviously, there is no need for any kind of concurrency checking when a record is being added. If the file has a unique KEY, two users adding the same record twice is impossible because the second ADD returns a "Creates Duplicate Key" error without adding the record. If duplicate KEYs are allowed, there is no generic way for the program code to check for inadvertent (incorrect) duplicates as opposed to deliberate (correct) duplicate records. There is also no need for concurrency checking when a record is being deleted. Once the first user has deleted the record, it is gone. Any subsequent user that attempts to delete that record will not be able to get it in the first place.

Concurrency checking is necessary when a user is making a change to a record. The process of changing a record is: get the record, make the changes, and write the changes back to the file. The problem is, during the time it takes the first user to make changes to the record, a second user (a faster typist) could: get the same record, make some change, and write the changed record back to disk. When the time comes for the first user to write his/her changes to disk, the record on disk is no longer the same as when it was first retrieved. Does the first user simply overwrite the second (faster) user's changes? If both users are changing different data elements within that record and both changes are valid, overwriting the second user's changes cannot be allowed. Even if they are both making the same change, the first user needs to know that someone else has already made that change.

For the simplest concurrency checking method, your program code should:

1. Save a copy of the record before any changes are made.
2. Re-read the record immediately before writing the changes to disk, and compare it with the saved original.
3. If the two are the same, allow the user's changes to be written to disk. If not, alert the user and display the record, as changed by the other user.

Assume the following global declarations and compiler equates:

```
Sample    FILE,DRIVER('Clarion'),PRE(Sam)   !A data file declaration
Field1Key   KEY(Sam:Field1)
Record      RECORD
Field1        LONG
Field2        STRING(10)
          . .
Action    LONG                            !Record update action variable
AddRec    EQUATE(1)
ChangeRec EQUATE(2)
```

Assume that some procedure allows the user to select a record from the file, define the expected Action (Add, Change, or Delete the record), then calls an update procedure. The update procedure operates only on that selected record and accomplishes the Action the user set in the previous procedure. The update procedure code would look similar to this:

```
Update      PROCEDURE                 !An update procedure
Screen      WINDOW
              !data entry screen declarations go here
            END
SaveQue     QUEUE,PRE(Sav)              !Record save queue is a copy
SaveRecord    LIKE(Sam:Record),PRE(Sav) !  of the file's record buffer
            END                         !    with a different prefix
SavRecPtr   LONG                        !Record pointer save variable
  CODE
  OPEN(Screen)
  Sav:SaveRecord = Sam:Record           !Save copy of record
  ADD(SaveQue,1)                        !  to QUEUE entry 1
  SavRecPtr = POINTER(Sample)           !Save record number
  DISPLAY                               !Display the record on screen
  ACCEPT                                  !Screen field process loop
    CASE ACCEPTED()
        !Individual screen field edit code goes here
    OF ?OKButton                        !Screen completion field
      IF Action = ChangeRec             !If changing an existing record
        Sav:SaveRecord = Sam:Record     !Save changes made
        ADD(SaveQue,2)                  !  to QUEUE entry 2
        GET(SaveQue,1)                  !Get original record from QUEUE
        GET(Sample,SavRecPtr)           !Get record from FILE again
        IF ERROR()
          IF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
            Action = AddRec             ! change Action to add it back
            GET(SaveQue,2)              !Get this user's changes
            Sam:Record = Sav:SaveRecord ! put them in record buffer
          ELSE
            STOP(ERROR())               !Stop on any other error
          END
        ELSIF Sav:SaveRecord <> Sam:Record !Compare for other's changes
          Sav:SaveRecord = Sam:Record   ! Save new disk record
          ADD(SaveQue,1)                !    to QUEUE entry 1
          DISPLAY                       ! Display other's changes
          BEEP                          ! Alert the user
          IF MESSAGE('Changed by another station').
          SELECT(1)                     !   and start over
          CYCLE                         !    at first field
        ELSE                            !If nobody changed it
          GET(SaveQue,2)                ! Get this user's changes
          Sam:Record = Sav:SaveRecord  ! put them back in record buffer
      . .
```

```
      EXECUTE Action                    !Execute disk write
        ADD(Sample)                     !If Action = 1 (AddRec)
        PUT(Sample)                     !If Action = 2 (ChangeRec)
        DELETE(Sample)                  !If Action = 3 (DeleteRec)
      END
      ErrorCheck                   !A generic error checking procedure
      FREE(SaveQue)                !Free memory used by queue entries
      BREAK                        !  and break out of process loop
  . .                              !End loop and case
```

This example code demonstrates the simplest type of concurrency checking. It saves the original record in memory QUEUE entry one, and the pointer to that record in a LONG variable. After that, the user is allowed to make the changes to the screen data. The code to check for other user's changes is contained in the CASE FIELD() OF ?OKButton. This would be the field which the user completes when he/she is done making changes and is ready to write the record to disk.

To check for other user's changes, the code first saves this user's changes to a second memory QUEUE entry, then gets the saved original record from the QUEUE. The saved record pointer is used to get the record from the data file again. If the record is not found in the file, someone else has deleted it. Therefore, since this user is changing it, simply add the changed record back into the file. If the record was not deleted, it is compared against the original saved copy. If they are not the same, the changed record is saved to the same memory QUEUE entry (one) which contained the original record. Then the user is alerted to the problem and sent back to the first field on the screen to re-enter the changes (if necessary). If the record is still the same, the user's changes are retrieved from the second memory QUEUE entry and put into the record buffer for the disk write. This method is fairly straight-forward and logical. However, it uses three extra chunks of memory the size of the record buffer: the memory QUEUE's buffer, and the two entries in that QUEUE (plus each QUEUE entry's 28-byte overhead). If you are dealing with a file that has many fields, the record buffer could be very large and this could use a significant amount of memory.

Another method of concurrency checking does not copy and save the original record, but instead calculates a Checksum or Cyclical Redundancy Check (CRC) value. The calculation is performed on the record before changes are made, then the record is retrieved from disk and the calculation is performed again. If the two values are not the same, the record has been changed. This method still requires a save area for the user's changes, because the record must be read again for the second calculation, and all disk reads are placed in the record buffer. Without a save area, the user's changes would be overwritten.

Here is an example of a 16-bit CRC function, and its prototype for the MAP structure. This is similar to the CRC calculations used in some serial communications protocols. An array of BYTE fields is passed to the function, it calculates a 16-bit CRC value for that array, and returns it to a USHORT (16-bit unsigned) variable.

```
  MAP                          !The function prototype for the MAP.
    CRC16(*BYTE[]),USHORT      !CRC16 expects an array of BYTEs to be
  END                          !  passed to it, returns a USHORT value
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CRC16   FUNCTION(Array)              !16 Bit CRC Check
CRC     ULONG                        !Work variable
  CODE
  LOOP X# = 1 TO MAXIMUM(Array,1)   !Loop through whole array
    CRC = BOR(CRC,Array[X#])         !Concatenate an array byte to CRC
    LOOP 8 TIMES                     !Loop through each bit
      CRC = BSHIFT(CRC,1)            !Shift CRC left one bit
      IF BAND(CRC,1000000h)          !Was CRC 24th bit on before shift?
        CRC = BXOR(CRC,102100h)      ! XOR shifted value with CRC mask
  . . .    !End both loops
  RETURN(BAND(BSHIFT(CRC,-8),0000FFFFh))  !Shift and mask return value
```

Using this CRC check function, the previous example code would be changed to look like:

```
Update      PROCEDURE                    !An update procedure
Screen      WINDOW
              !data entry screen declarations go here
            END
Sav:SaveRecord  LIKE(Sam:Record),PRE(Sav),STATIC
                                         !Record buffer save area
                                         !  with a different prefix
PassArray   BYTE,DIM(SIZE(Sam:Record),OVER(Sam:Record)
                                         !Declare array OVER Sam:Record
SavRecPtr   LONG                         !Record pointer save variable
SavCRC      USHORT                       !CRC value save variable
  CODE
  OPEN(Screen)
  SavCRC = CRC16(PassArray)              !Save original CRC value
  SavRecPtr = POINTER(Sample)            !Save record number
  DISPLAY                                !Display the record on screen
  ACCEPT                                   !Screen field process loop
    CASE ACCEPTED()
      !Individual screen field edit code goes here
    OF ?OKButton                         !Screen completion field
      IF Action = ChangeRec              !If changing an existing record
        Sav:SaveRecord = Sam:Record      !Save changes made
        GET(Sample,SavRecPtr)            !Get record from FILE again
        IF ERROR()
          IF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
            Action = AddRec              ! change Action to add it back
            Sam:Record = Sav:SaveRecord ! put them in record buffer
          ELSE
            STOP(ERROR())                !Stop on any other error
          END
        ELSIF SavCRC <> CRC16(PassArray)  !Compare CRCs for changes
          SavCRC = CRC16(PassArray)      ! Save new CRC value
          DISPLAY                        ! Display other's changes
          BEEP                           ! Alert the user
          IF MESSAGE('Changed by another station').
          SELECT(1)                      !   and start over
          CYCLE                          !    at first field
        ELSE                             !If nobody changed it
          Sam:Record = Sav:SaveRecord    ! put changes back in buffer
      . .
      EXECUTE Action                     !Execute disk write
        ADD(Sample)                      !If Action = 1 (AddRec)
        PUT(Sample)                      !If Action = 2 (ChangeRec)
```

```
      DELETE(Sample)                  !If Action = 3 (DeleteRec)
    END
    ErrorCheck                      !A generic error checking procedure
    BREAK                           !  and break out of process loop
. .                                 !End loop and case
```

You can see that the update procedure code using this method is a bit smaller, and easier to follow logically. There are two new data declarations: SavCRC is declared to save the original CRC calculation, and PassArray is an array declared OVER the file's record buffer. The PassArray declaration simply provides a way to pass the CRC16 function the entire record as an array of BYTEs, it does not allocate any memory.

A valid question at this point would be, "Why is the Sav:SaveRecord declared in this code with a STATIC attribute?" There are four reasons:

- A save area is still needed to temporarily keep the user's changes while concurrency checking calculations are being performed.
- Variables declared locally in a PROCEDURE or FUNCTION are assigned memory on the stack when the PROCEDURE or FUNCTION is called.
- The save area for a large record buffer could easily take more memory than is available on the stack. Therefore, the save area should be in static memory.
- In a PROCEDURE or FUNCTION, only data structures (SCREEN, PULLDOWN, REPORT, FILE, and QUEUE) and variables with ther STATIC attribute are assigned static memory.

Of course, if the save area were declared in the global data section (between the keywords PROGRAM and CODE), or a MEMBER module's data section (between the keywords MEMBER and PROCEDURE, or FUNCTION), it would not need to be declared with the STATIC attribute—it would automatically be assigned static memory.

HOLD and RELEASE

A tool to prevent other users from making changes to a record while it is being updated is the HOLD statement. HOLD tells the following GET, NEXT, or PREVIOUS statement to get the record and set a flag that tells any other user attempting to get that record that it is in use—a "record lock." The record remains held until it is: explicitly released with a RELEASE statement; implicitly released by a PUT, or DELETE, of that record; or, implicitly released by retrieving another record from the same file.

The Clarion language supports multiple file systems through its file driver technology. Each file system may implement record locking in a different

manner. Therefore, the actual effect of HOLD is dependent upon the file driver, which takes whatever action is appropriate to the file system. In some file systems, a HOLD on a record allows other users to read the record, but not to write to it. In others, HOLD blocks other users from any access to the record. Some file systems release the HOLD automatically if the system crashes, others don't and leave it flagged as held. The specific action of HOLD is described in each file driver's documentation.

If you HOLD a record when it is retrieved, and RELEASE it when you write it back, you can eliminate the need for the type of concurrency checking previously described. Is this a good idea, though? Depending upon the actual implementation of HOLD in the file system being used, the answer may be either Yes or No.

If your file system blocks other users from all access to the record, or there is the possibility a "system crash" could leave it in a HOLD state, the answer is probably No. This does not mean that you should not use HOLD at all. It does mean that you should not use HOLD where a record would be held during user input (an indeterminate amount of time). More likely, you would use HOLD during the concurrency check described above. This is to make sure that nobody changes the record between the second GET (for concurrency checking) and the PUT that writes the user's changes to disk.

Using HOLD in this manner only changes the code in the CASE ACCEPTED() OF ?OKButton.

```
OF ?OKButton                        !Screen completion field
  IF Action = ChangeRec             !If changing an existing record
    Sav:SaveRecord = Sam:Record     !Save changes made
    HOLD(Sample,1)                  !Hold while checking for changes
    GET(Sample,SavRecPtr)           !Get record from FILE again
    IF ERROR()
      IF ERROR() = 'RECORD ALREADY HELD' !Has someone else got it?
        BEEP                        ! Alert the user
        SHOW(25,1,'Held by another station')
        SELECT(1)                   !   and start over
        CYCLE                       !    at first field
      ELSIF ERROR() = 'RECORD NOT FOUND' !Did someone else delete it?
        Action = AddRec             ! change Action to add it back
        Sam:Record = Sav:SaveRecord ! put them back in record buffer
      ELSE
        STOP(ERROR())               !Stop on any other error
      END
    ELSIF SavCRC <> CRC16(PassArray) !Compare CRC values for changes
      RELEASE(Sample)               !Release the hold
      SavCRC = CRC16(PassArray)     ! Save new CRC value
      DISPLAY                       ! Display other's changes
      BEEP                          ! Alert the user
      IF MESSAGE('Changed by another station').
      SELECT(1)                     !   and start over
      CYCLE                         !    at first field
    ELSE                            !If nobody changed it
      Sam:Record = Sav:SaveRecord ! put these changes back in buffer
  . .
  EXECUTE Action                    !Execute disk write
    ADD(Sample)                     !If Action = 1 (AddRec)
```

```
      PUT(Sample)                   !If Action = 2 (ChangeRec)
      DELETE(Sample)                !If Action = 3 (DeleteRec)
    END
    ErrorCheck                  !A generic error checking procedure
    BREAK                       !  and break out of process loop
```

This change puts a HOLD on the record only long enough to determine if it is the same record the user started with and write the changes to disk. If someone else has a HOLD on the record, the user is alerted to that fact and allowed to try again. If the record continually comes up as held by another station, then it has probably been left in a HOLD state by a system crash. In that case, the hold should be released by whatever action is appropriate for that file system. Code could be written to handle this eventuality, but it would be specific to the file system and this is "generic" example code.

If the prevention of record update conflicts is a "mission-critical" concern, then HOLD could be used to keep control of the record during user data entry. One trade-off with this use of HOLD is the nuisance of dealing with records that are left locked when users' systems crash while records are held. Correcting that situation could involve some manual work with file system utilities, or could simply be a matter of specific coding considerations for the file system being used. Another concern with using HOLD this way comes when the file system being used does not allow other users to read the held records. The held records would seem to "disappear" then "reappear" from time to time as users HOLD records. Either way, this method should probably not be used unless the application really requires it.

To utilize this technique, the HOLD would have to be in the procedure which actually retrieves the record from the file. In most cases, that procedure would display some kind of scrolling list of records, usually displayed in a LIST box. The following example code demonstrates this.

```
  OF ?List                          !LIST
    CASE EVENT()
    OF EVENT:Accepted               !An existing record was selected
      GET(TableQue,CHOICE())        !Get record number from the QUEUE
      HOLD(Sample,1)                !Arm the HOLD
      GET(Sample,Que:RecPointer)    !  and get the record from the file
      IF ERROR() = 'RECORD ALREADY HELD' !Has someone else got it?
        BEEP                        ! Alert the user
        IF MESSAGE('Held by another station').
        SELECT(?List)             !   to try again
        CYCLE
      ELSE                          !If no one else has it
        Action = ChangeRec          !Set up disk action for change
        Update                      ! and call the update procedure
      END
      !Code to handle other keycodes goes here
    END
```

This technique grossly simplifies the update procedure code, as in this example:

```
Update  PROCEDURE    !An update procedure
Screen  SCREEN
          !data entry screen declarations go here
        END
```

```
CODE
OPEN(Screen)
DISPLAY                             !Display the record on screen
ACCEPT                              !Screen field process loop
  CASE FIELD()
    !Individual screen field edit code goes here
  OF ?OKButton                      !Screen completion field
    CASE EVENT()
    OF EVENT:Accepted
      EXECUTE Action                !Execute disk write
        ADD(Sample)                 !If Action = 1 (AddRec)
        PUT(Sample)                 !If Action = 2 (ChangeRec)
        DELETE(Sample)              !If Action = 3 (DeleteRec)
      END
      ErrorCheck                    !A generic error checking procedure
      BREAK                         !  and break out of process loop
. . .                               !End loop and case
```

The HOLD statement only allows each user to HOLD one record in each file. If you need to update multiple records in one file and you must be sure that no other user makes changes to those records while they are being updated, then you must LOCK the file.

LOCK and UNLOCK

The LOCK statement prevents other users from accessing any records in a file, until you UNLOCK it. Just like HOLD, the effect of LOCK is dependent upon the file driver which takes whatever action is appropriate to the file system. In some file systems, a system crash automatically unlocks the file, and in others it is left locked. The specific action LOCK takes is described in each file driver's documentation.

Because other users are completely barred from accessing records in the LOCKed file, LOCK is not commonly used. The most common use of LOCK would be to BUILD an INDEX prior to using it (and that is not even necessary if it is a "dynamic" INDEX). The type of "batch update processing" that would require a file to be LOCKed for a significant period of time is generally best left until after hours, when all users are gone. Other than to BUILD an INDEX, a file LOCK is usually only needed during Transaction Processing, which is the subject of a separate essay.

If an application truly demands a file LOCK, then the period of time during which the other users are denied access should be kept to an absolute minimum. The code between the file LOCK and its subsequent UNLOCK statement should not require any user input. This means that the code should be written such that an end-user cannot go to lunch leaving a file LOCKed. Specifically, LOCK should come immediately before the BUILD occurs, and the file should be UNLOCKed as soon as it is complete.

```
ReportProc    PROCEDURE
Sample    FILE,DRIVER('Clarion'),PRE(Sam)  !A data file declaration
Field1Key   KEY(Sam:Field1)
Field2Ndx   INDEX(Sam:Field2)   !An INDEX
```

```
Record      RECORD
Field1        LONG
Field2        STRING(10)
            . .
Report      REPORT
              !Report declaration statements go here
            END
  CODE
  OPEN(Sample,42h)                      !Open Read/Write Deny None
  LOCK(Sample,1)                        !Lock the file
  IF ERROR() = 'FILE IS ALREADY LOCKED' !Check for other locks
    BEEP                                !Alert the user
    IF MESSAGE('Locked by another station').
    RETURN                              !  and get out
  END
  BUILD(Sam:Field2Ndx)                  !Build the index
  UNLOCK(Sample)                        !Unlock the file
  OPEN(Report)
  SET(Sam:Field2Ndx)                    !Use the index
  LOOP
    NEXT(Sample)
    IF ERRORCODE() THEN BREAK.
      !Report processing code goes here
  END
```

This code opens the file in *access mode* 42h (Read/Write Deny None) for fully shared access. The LOCK is attempted for one second. If it is successful, the BUILD immediately executes. If the LOCK was unsuccessful, the user is alerted and returned to the procedure that called the report. Once the BUILD is complete, UNLOCK once again allows other users access to the file, and the report is run based on the sort order of the INDEX.

"Deadly Embrace"

There are two forms of "deadly embrace." The first occurs when two users attempt to LOCK the same set of files in separate orders of sequence. The scenario is:

User A locks file A
User B locks file B at the same time
User A attempts to LOCK file B and cannot because User B has it LOCKed
User B attempts to LOCK file A and cannot because User A has it LOCKed

This leaves both users "hung up" attempting to gain control of the files. The solution to this dilemma is the adoption of a simple coding convention: Always LOCK files in the same order (alphabetical works just fine) and trap for other users' LOCKs. This example demonstrates the principle:

```
LOOP
  LOCK(FileA,1)                    !Attempt LOCK for 1 second
  IF ERROR() = 'FILE IS ALREADY LOCKED'
    BEEP(0,100)                    !Wait 1 second
    CYCLE                          !  and try again
  END
  LOCK(FileB,1)                    !Attempt LOCK for 1 second
  IF ERROR() = 'FILE IS ALREADY LOCKED'
    UNLOCK(FileA)                  !Unlock the locked file
    BEEP(0,100)                    !Wait 1 second
    CYCLE                          !  and try again
  END
  BREAK                            !Break from loop when both locked
END
```

This code will eventually LOCK both files. If FileA is already locked by another user, the loop will pause for one second while BEEP puts out a zero-frequency beep (no sound is generated), then try again. The one second pause allows the other user a chance to complete their action. If the first LOCK is successful, the LOCK on FileB is attempted. If FileB is already locked by another user, FileA is immediately unlocked for other user's use, then the pause and re-try sequence occurs again. The BREAK from the LOOP in this example is only allowed after both files are successfully LOCKed.

Mixing the use of HOLD and LOCK can result in the second form of "deadly embrace." In some file systems, LOCK and HOLD are completely independent, therefore it is possible for one user to HOLD a record in a file, and another user to LOCK that same file. The user with the HOLD cannot write the record back, or even RELEASE it, and the user with the LOCK cannot write to that held record.

This situation may be resolved in one of two ways:

- You may choose to never mix HOLD and LOCK on the same file. This limits you to the use of HOLD only (the most common solution), or LOCK only. This solution must be used in all applications that write to a common set of files.
- You may choose to always trap for held records while the file is LOCKed. This implies that you know how you want to deal with the "deadly embrace" record when it is detected.

The first solution is by far the more commonly used. The second takes you into an area of programming that is probably better served by Transaction Processing, which would include held record trapping.

Summary

- A File must be opened before its records may be accessed.
- The *access mode* determines the type of access DOS grants to the user opening the file and any other users.
- Multi-user programming must always take into consideration the possibility of multiple users accessing the same record at the same time.
- Concurrency checking is done to ensure that user updates don't overwrite other user's changes to the records.
- HOLD is most commonly used in conjunction with concurrency checking to ensure that no other user changes the record while it is being compared for changes.
- LOCK is most commonly used to gain exclusive control of a file while you BUILD an INDEX.
- The "deadly embrace" is a programming consideration most easily dealt with through the adoption of consistent program coding conventions.

View Structures

Contents

VIEW (declare a "virtual" file)

```
label   VIEW(primary file) [,FILTER( )]
            [PROJECT( )]
            [JOIN( )
                [PROJECT( )]
                [JOIN( )
                    [PROJECT( )]
                 END]
             END]
        END
```

VIEW	Declares a "virtual" file as a composite of related files.
label	The name of the VIEW.
primary file	The label of the primary FILE of the VIEW.
FILTER	Declares an expression used to filter valid records for the VIEW.
PROJECT	Specifies the fields from the *primary file*, or the secondary related file specified by a JOIN structure, that the VIEW will retrieve. If omitted, all fields from the file are retrieved.
JOIN	Declares a secondary related file.

VIEW declares a "virtual" file as a composite of related data files. The data elements declared in a VIEW do not physically exist in the VIEW, because the VIEW structure is a logical construct. VIEW is a separate method of addressing data physically residing in multiple, related FILE structures. At run-time, the VIEW structure is not assigned memory for a data buffer, so the fields used in the VIEW are placed in their respective FILE structure's record buffer.

A VIEW structure must be explicitly OPENed before use, and all primary and secondary related files used in the VIEW must have been previously OPENed. File I/O operations that operate directly on the primary or any secondary related file in the VIEW are not permitted while the VIEW is OPEN.

The VIEW data structure allows sequential access, only. A SET statement on the VIEW's primary file must be issued before the OPEN(view) to set the VIEW's processing order and starting point, then NEXT(view) or PREVIOUS(view) allow sequential access to the VIEW. The REGET statement is also available for VIEW, but only to specify the primary and secondary related file records that should be current in their respective record buffers after the VIEW is CLOSEd. If no REGET statement is issued immediately before the CLOSE(view) statement, the primary and secondary

related file record buffers are set to no current record. The processing sequence of the primary and secondary related files is undefined after the VIEW is CLOSEd. Therefore, SET or RESET must be used to establish sequential file processing order, if necessary, after closing the VIEW.

The VIEW data structure is designed to facilitate database access on client-server systems. It accomplishes two relational operations at once: the relational "Join" and "Project" operations. On client-server systems, these operations are performed on the file server, and only the result of the operation is sent to the client. This can dramatically improve performance of network applications.

A relational "Join" retrieves data from multiple files, based upon the relationships defined between the files. The JOIN structure in a VIEW structure defines the relational "Join" operation. There may be multiple JOIN structures within a VIEW, and they may be nested within each other to perform multiple-level "Join" operations.

A relational "Project" operation retrieves only specified data elements from the files involved, not their entire record structure. Only those fields explicitly declared in PROJECT statements in the VIEW structure are retrieved. Therefore, the relational "Project" operation is automatically implemented by the VIEW structure. The contents of fields that are not contained in the PROJECT are undefined.

The FILTER attribute restricts the VIEW to a sub-set of records. The FILTER expression may include any of the fields explicitly declared in the VIEW structure and restrict the VIEW based upon the contents of any of the fields. This makes the FILTER operate across all levels of the "Join" operation.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
City            STRING(20)
State           STRING(20)
Zip             STRING(20)
          . .

Header    FILE,DRIVER('Clarion'),PRE(Hea) !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber    LONG
OrderNumber   LONG
ShipToName    STRING(20)
ShipToAddr    STRING(20)
ShipToCity    STRING(20)
ShipToState   STRING(20)
ShipToZip     STRING(20)
          . .

Detail    FILE,DRIVER('Clarion'),PRE(Dtl) !Declare detail file layout
OrderKey    KEY(Dtl:OrderNumber)
Record      RECORD
OrderNumber   LONG
Item          LONG
Quantity      SHORT
          . .

Product   FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey     KEY(Pro:Item)
Record      RECORD
Item          LONG
Description   STRING(20)
Price         DECIMAL(9,2)
          . .

ViewOrder VIEW(Customer)                          !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber)      !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber)  !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item)        !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
```

FILTER (set view filter expression)

FILTER(*expression***)**

FILTER Specifies a filter *expression* used to evaluate records to include in the VIEW.

expression A string constant containing a logical expression.

The **FILTER** attribute specifies a filter *expression* used to evaluate records to include in the VIEW.

The *expression* may reference any field in the VIEW, at all levels of JOIN structures. The entire *expression* must evaluate as true for a record to be included in the VIEW. The *expression* may contain any valid Clarion language logical expression. You must BIND all variables used in the *expression*.

Example:

```
 !Get only orders for customer 9999 since order number 100
ViewOrder VIEW(Customer),FILTER('Cus:AcctNumber = 9999 AND Hea:OrderNumber > 100')
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber)        !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber)    !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item)          !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END

  CODE
  OPEN((Customer,22h)
  OPEN((Header,22h)
  OPEN((Product,22h)
  OPEN(Detail,22h)
  BIND('Cus:AcctNumber',Cus:AcctNumber)
  BIND('Hea:AcctNumber',Hea:AcctNumber)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
  LOOP
    NEXT(ViewOrder)
    IF ERRORCODE() THEN BREAK.
    !Process the valid record
  END
  UNBIND('Cus:AcctNumber',Cus:AcctNumber)
  UNBIND('Hea:AcctNumber',Hea:AcctNumber)
  CLOSE(Header)
  CLOSE(Customer)
  CLOSE(Product)
  CLOSE(Detail)
```

PROJECT (set view fields)

PROJECT(*fields* **)**

PROJECT	Declares the fields retrieved for the VIEW.
fields	A comma delimited list of fields (including prefixes) from the primary file of the VIEW, or the secondary related file named in the JOIN structure, containing the PROJECT declaration.

The **PROJECT** statement in a VIEW structure declares *fields* retrieved for a relational "Project" operation. A relational "Project" operation retrieves only the specified *fields* from the file, not the entire record structure. Only those fields explicitly declared in PROJECT declarations in the VIEW structure are retrieved.

A PROJECT statement may be declared in the VIEW, or within one of its component JOIN structures. If there is no PROJECT declaration in the VIEW or JOIN structure, all fields in the relevant file are retrieved.

Example:

```
Detail    FILE,DRIVER('Clarion'),PRE(Dtl) !Declare detail file layout
OrderKey    KEY(Dtl:OrderNumber)
Record      RECORD
OrderNumber   LONG
Item          LONG
Quantity      SHORT
Description   STRING(20)    !Line item comment
          . .

Product   FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey     KEY(Pro:Item)
Record      RECORD
Item          LONG
Description   STRING(20)    !Product description
Price         DECIMAL(9,2)
          . .

ViewOrder VIEW(Detail)
            PROJECT(Det:OrderNumber,Det:Item,Det:Description)
            JOIN(Pro:ItemKey,Det:Item)
              PROJECT(Pro:Description,Pro:Price)
            END
          END
```

JOIN (declare a "join" operation)

```
JOIN(secondary key,linking fields)
        [PROJECT( )]
        [JOIN( )
              [PROJECT( )]
         END]
END
```

JOIN	Declares a secondary file for a relational "Join" operation.
secondary key	The label of a KEY which defines the secondary FILE and its access key.
linking fields	A comma-delimited list of fields in the related file that contain the values the *secondary key* uses to access records.
PROJECT	Specifies the fields from the secondary related file specified by a JOIN structure that the VIEW will retrieve. If omitted, all fields from the file are retrieved.

The **JOIN** structure declares a secondary file for a relational "Join" operation. A relational "Join" retrieves data from multiple files, based upon the relationships defined between the files. There may be multiple JOIN structures within a VIEW, and they may be nested within each other to perform multiple-level "Join" operations.

The *secondary key* defines the access key for the secondary file. The *linking fields* name the fields in the file to which the secondary file is related, that contain the values used to retrieve the related records. For a JOIN directly within the VIEW, these fields come from the VIEW's primary file. For a JOIN nested within another JOIN, these fields come from the secondary file of the JOIN in which it is nested. Non-linking fields in the *secondary key* are allowed as long as they appear in the list of the key's component fields after all the linking fields.

When data is retrieved, if there are no matching secondary file records for a primary file record, null values are supplied in the fields specified in the PROJECT. This type of relational "Join" operation is known as an "outer join." The FILTER attribute of the VIEW can be used to accomplish all other forms of the relational "Join" operation.

Example:

```
Header      FILE,DRIVER('Clarion'),PRE(Hea) !Declare header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey      KEY(Hea:AcctNumber,Hea:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
ShipToCity      STRING(20)
ShipToState     STRING(20)
ShipToZip       STRING(20)
            . .
Detail      FILE,DRIVER('Clarion'),PRE(Dtl) !Declare detail file layout
OrderKey      KEY(Dtl:AcctNumber,Dtl:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Item            LONG
Quantity        SHORT
            . .
Product     FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey       KEY(Pro:Item)
Record        RECORD
Item            LONG
Description     STRING(20)
Price           DECIMAL(9,2)
            . .
ViewOrder   VIEW(Header)    !Declare VIEW structure
              PROJECT(Hea:AcctNumber,Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:AcctNumber,Hea:OrderNumber) !Join Detail file
                PROJECT(Dtl:ItemDtl:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item)  !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
```

View Commands

CLOSE (close a VIEW)

CLOSE(*view*)

CLOSE Closes a VIEW.

view The label of a VIEW.

The **CLOSE** statement closes a VIEW. A VIEW declared within a procedure is implicitly closed upon RETURN from the procedure, if it has not already been explicitly CLOSEd.

If the CLOSE(*view*) statement is not immediately preceded by a REGET statement, the primary and secondary related files in the VIEW are set to no current record. This means the contents of their record buffers are undefined and a SET or RESET statement must be issued before performing sequential processing on the file.

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
City            STRING(20)
State           STRING(20)
Zip             STRING(20)
            . .
ViewCust    VIEW(Customer)   !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
            END
  CODE
  OPEN(Customer,22h)
  SET(Cus:AcctKey)
  OPEN(ViewCust)          !Open the customer view
   !executable statements
  CLOSE(ViewCust)         !and close it again
```

OPEN (open a VIEW)

OPEN(*view*)

OPEN	Opens a VIEW structure for processing.
view	The label of a VIEW declaration.

The **OPEN** statement opens a VIEW structure for processing. A VIEW must be explicitly opened before it may be accessed. The files used in the VIEW must already be open.

Immediately before the OPEN(*view*) statement, you must issue a SET statement on the VIEW structure's primary file to setup sequential processing for the VIEW. You cannot issue a SET statement to the primary file while the VIEW is OPEN; you must CLOSE(*view*) then issue the SET before a subsequent OPEN(*view*).

Example:

```
Header      FILE,DRIVER('Clarion'),PRE(Hea)   !Declare header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey      KEY(Hea:OrderNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
ShipToCity      STRING(20)
ShipToState     STRING(20)
ShipToZip       STRING(20)
            . .
Detail      FILE,DRIVER('Clarion'),PRE(Dtl) !Declare detail file layout
OrderKey      KEY(Dtl:OrderNumber)
Record        RECORD
OrderNumber     LONG
Item            LONG
Quantity        SHORT
            . .
Product     FILE,DRIVER('Clarion'),PRE(Pro) !Declare product file layout
ItemKey       KEY(Pro:Item)
Record        RECORD
Item            LONG
Description     STRING(20)
Price           DECIMAL(9,2)
            . .
ViewOrder   VIEW(Header)                              !Declare VIEW structure
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber)      !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item)            !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
            . . .
  CODE
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder) !Open
```

DELETE (delete a view primary file record)

DELETE(*view*)

DELETE Removes a primary file record from a VIEW.

view The label of a VIEW declaration.

The **DELETE** statement removes the last VIEW primary file record that was accessed by a NEXTor PREVIOUS statement. The key entries for that record are also removed from the KEYs. DELETE does not remove records from any secondary JOIN files in the VIEW.

DELETE only deletes the primary file record in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to write only to the Primary file. Therefore, Clarion has adopted this same industry standard solution.

If no record was previously accessed, or the record is held by another workstation, DELETE posts the "Record Not Available" error and no record is deleted. The specific disk action DELETE performs in the file is file driver dependent.

Errors Posted: 05 Access Denied
33 Record Not Available

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus)   !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
Name            STRING(20)
Addr            STRING(20)
City            STRING(20)
State           STRING(20)
Zip             STRING(20)
              . .
CustView    VIEW(Customer)                    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
            END
  CODE
  OPEN(Customer)
  Cus:AcctNumber = 12345           !Initialize key field
  SET(Cus:AcctKey,Cus:AcctKey)
  OPEN(CustView)
  NEXT(CustView)                   !Get that record
    IF ERRORCODE() THEN STOP(ERROR()).
  DELETE(CustView)                 !Delete the customer record
```

See Also: HOLD, NEXT, PREVIOUS, PUT

HOLD (exclusive view record access)

HOLD(*view* [,*seconds*]**)**

HOLD	Arms record locking.
view	The label of a VIEW opened for shared access.
seconds	A numeric constant or variable which specifies the maximum wait time in seconds.

The **HOLD** statement arms record locking for the primary file in the VIEW in a multi-user environment. The following NEXT or PREVIOUS statement flags the primary file record as "held" when it successfully gets the record. Generally, this excludes other users from writing to the record, although it does not prevent them from reading the record. The specific action HOLD takes is file driver dependent.

HOLD(*view*)	Arms the process so that the following NEXT or PREVIOUS attempts to hold the record until it is successful. If it is held by another workstation, GET, NEXT, or PREVIOUS will wait until the other workstation releases it.
HOLD(*view*,*seconds*)	Arms the process so that the following NEXT or PREVIOUS statement posts the "Record Is Already Held" error after unsuccessfully trying to hold the record for *seconds*.

A user may only HOLD one record at a time in the VIEW. If a second record is to be accessed in the same file, the previously held record must be released (see RELEASE).

As with LOCK, a common problem to avoid when holding records is "deadly embrace." This condition occurs when two workstations attempt to hold the same set of records in two different orders and both are using the HOLD(*view*) form of HOLD. One workstation has already held a record that the other is trying to HOLD, and vice versa. This problem may be avoided by using the HOLD(*view*,*seconds*) form of HOLD, and trapping for the "Record Is Already Held" error.

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
                JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
  LOOP                          !Process records Loop
    LOOP                        !Loop to avoid "deadly embrace"
      HOLD(ViewOrder,1)         !Arm Hold on view, try for 1 second
      NEXT(ViewOrder)           !Get and hold the record
      IF ERRORCODE() = 43       !If someone else has it
        CYCLE                   ! try again
      ELSE
        BREAK                   !Break if not held
      END
    END
    IF ERRORCODE() THEN BREAK.      !Check for end of file
      !Process the records
    RELEASE(ViewOrder)            !release held records
  END
  CLOSE(ViewOrder)
```

See Also: RELEASE, NEXT, PREVIOUS

NEXT (read next view record in sequence)

NEXT(*view*)

NEXT Reads the next record(s) in sequence for a VIEW.

view The label of a VIEW declaration.

NEXT reads the next record(s) in sequence from a VIEW and places the appropriate fields in the VIEW structure component files' data buffer(s). If the VIEW contains JOIN structures, NEXT retrieves the appropriate next set of related records.

The SET statement issued on the VIEW's primary file before the OPEN(view) statement determines the sequence in which records are read. The first NEXT(*view*) reads the record at the position specified by the SET statement. Subsequent NEXT statements read subsequent records in that sequence. The sequence is not affected by PUT or DELETE statements.

Executing NEXT without a preceding SET, or attempting to read past the end of the primary file in the VIEW posts the "Record Not Available" error.

Errors Posted:
33 Record Not Available
37 File Not Open
43 Record Is Already Held

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
                JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
   LOOP                              !Read all records through end of primary file
    NEXT(ViewOrder)                  ! read a record sequentially
      IF ERRORCODE() THEN BREAK.     ! break on end of file
    DO PostTrans                     ! call transaction posting routine
  END                                !End loop
```

See Also: PREVIOUS, HOLD

NOMEMO (read view record without reading memos)

NOMEMO(*view***)**

NOMEMO Arms “memoless” record retrieval.

view The label of a VIEW.

The **NOMEMO** statement arms “memoless” record retrieval for the next NEXT or PREVIOUS statement encountered. The following NEXT or PREVIOUS gets the record but does not get any associated MEMO field(s) for the record. Generally, this speeds up access to the record when the contents of the MEMO field(s) are not needed by the procedure.

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Notes         MEMO(1024)
Record        RECORD
AcctNumber      LONG
Name            STRING(20)
Addr            STRING(20)
City            STRING(20)
State           STRING(20)
Zip             STRING(20)
              . .
CustView    VIEW(Customer)    !Declare VIEW structure
              END
  CODE
  OPEN(Customer)
  Cus:AcctNumber = 12345      !Initialize key field
  SET(Cus:AcctKey,Cus:AcctKey)
  OPEN(CustView)
  LOOP
    NOMEMO(CustView)
    NEXT(CustView)  !Get that record
      IF ERRORCODE() THEN BREAK.
    !Process the record
  END
  CLOSE(CustView)
```

See Also: GET, NEXT, PREVIOUS

PREVIOUS (read previous view record in sequence)

PREVIOUS(*view*)

PREVIOUS Reads the previous record in sequence from a VIEW.

view The label of a VIEW declaration.

PREVIOUS reads the previous record(s) in sequence from a VIEW and places the appropriate fields in the VIEW structure component files' data buffer(s). If the VIEW contains JOIN structures, PREVIOUS retrieves the appropriate previous set of related records.

The SET statement issued on the VIEW's primary file before the OPEN(view) statement determines the sequence in which records are read. The first PREVIOUS(*view*) reads the record at the position specified by the SET statement. Subsequent PREVIOUS statements read subsequent records in that sequence. The sequence is not affected by PUT or DELETE statements.

Executing PREVIOUS without a preceding SET, or attempting to read past the beginning of the primary file in the VIEW posts the "Record Not Available" error.

Errors Posted:
33 Record Not Available
37 VIEW Not Open
43 Record Is Already Held

Example:

```
ViewOrder   VIEW(Header)
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber)      !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item)            !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
  CODE
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
   LOOP                          !Read all records through beginning of primary file
    PREViOUS(ViewOrder)            ! read a record sequentially
      IF ERRORCODE() THEN BREAK.   ! break on end of file
    DO PostTrans                   ! call transaction posting routine
  END                              !End loop
```

See Also: NEXT, HOLD

PUT (write VIEW primary file record back)

PUT(*view*)

PUT Writes the VIEW's primary file record back to disk.

view The label of a VIEW declaration.

The **PUT** statement writes the current values in the VIEW structure's primary file's data buffer to a previously accessed primary file record in the *view*. If the record was held, it is automatically released. PUT writes to the last record accessed with the NEXT or PREVIOUS statements. If the values in the key variables were changed, then the KEYs are updated.

PUT only writes to the primary file in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to write only to the Primary file. Therefore, Clarion has adopted this same industry standard solution.

If a record was not accessed with NEXT or PREVIOUS statements, or was deleted, then the "Record Not Available" error is posted. PUT also posts the "Creates Duplicate Key" error. If any error is posted, then the record is not written to disk.

Errors Posted:

05 Access Denied
33 Record Not Available
40 Creates Duplicate Key

Example:

```
ViewOrder   VIEW(Header)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
              END
            END
  CODE
  OPEN((Header,22h)
  OPEN(Detail,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
  LOOP                                !Read all records in reverse order
    PREVIOUS(ViewOrder)               ! read a record sequentially
    IF ERRORCODE() THEN BREAK.        ! break at beginning of file
    DO LastInFirstOut                 !Call last in first out routine
    PUT(ViewOrder)                    !Write transaction record back to the file
    IF ERRORCODE() THEN STOP(ERROR()).
  END                                !End loop
```

See Also: NEXT, PREVIOUS

REGET (reget view record)

REGET(*view,string***)**

REGET	Re-gets a specific record in the VIEW.
view	The label of a VIEW declaration.
string	A string constant or variable containing the string returned by the POSITION function.

The **REGET** reads the VIEW record identified by the *string* returned by the POSITION(view) function. The value contained in the *string* returned by the POSITION function, and its length, are file driver dependent. If the VIEW contains JOIN structures, REGET retrieves the appropriate set of related records.

REGET re-loads all the VIEW component files' record buffers with complete records. It does not perform the relational "Project" operation. REGET(*view*) is explicitly designed to reset the record buffers to the appropriate records immediately prior to a CLOSE(*view*) statement. However, the processing sequence of the files must be reset with a SET or RESET statement.

Errors Posted: 33 Record Not Available

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
                JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
RecordQue   QUEUE,PRE(Que)
AcctNumber    LIKE(Cus:AcctNumber)
Name          LIKE(Cus:Name)
OrderNumber   LIKE(Hea:OrderNumber)
Item          LIKE(Det:Item)
Quantity      LIKE(Det:Quantity)
Description   LIKE(Pro:Description)
Price         LIKE(Pro:Price)
SavPosition   STRING(260)
            END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
  LOOP                                      !Read all records in file
    NEXT(ViewOrder)                         ! read a record sequentially
      IF ERRORCODE()
        DO DisplayQue
        BREAK
      END
    RecordQue :=: Cus:Record                !Move record into queue
    RecordQue :=: Hea:Record                !Move record into queue
    RecordQue :=: Dtl:Record                !Move record into queue
    RecordQue :=: Pro:Record                !Move record into queue
    SavPosition = POSITION(ViewOrder)       !Save record position
    ADD(RecordQue)                          ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
  END
  ACCEPT
    CASE ACCEPTED()
    OF ?ListBox
      GET(RecordQue,CHOICE())
      REGET(ViewOrder,Que:SavPosition)      !Reset the record buffers
      CLOSE(ViewOrder)                      ! and get the record again
      FREE(RecordQue)
      UpdateProc                            !Call Update Procedure
      BREAK
    END
  END
```

See Also: POSITION, RESET

RELEASE (release a held view record)

RELEASE(*view*)

RELEASE	Releases the held record(s).
view	The label of a VIEW declaration.

The **RELEASE** statement releases a previously held record in a VIEW. It will not release a record held by another user in a multi-user environment. If the record is not held, or is held by another user, RELEASE is ignored.

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
                JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
   LOOP                              !Process records Loop
    LOOP                             !Loop to avoid "deadly embrace"
      HOLD(ViewOrder,1)              !Arm Hold on view, try for 1 second
      NEXT(ViewOrder)                !Get and hold the record
      IF ERRORCODE() = 43            !If someone else has it
        CYCLE                        ! and try again
      ELSE
        BREAK                        !Break if not held
      END
    END
    IF ERRORCODE() THEN BREAK.       !Check for end of file
      !Process the records
    RELEASE(ViewOrder)               !release held records
  END
```

See Also: HOLD

RESET (reset view record sequence position)

RESET(*view,string***)**

RESET	Resets the sequential processing pointer to a specific record in the VIEW.
view	The label of a VIEW.
string	The string returned by the POSITION function.

RESET restores the record pointer to the record identified by the *string* that was returned by the POSITION function. Once RESET has restored the record pointer, either NEXT or PREVIOUS will read that record.

The value contained in the *string* (returned by the POSITION function) and its length, are file driver dependent. RESET is used in conjunction with POSITION to temporarily suspend and resume sequential VIEW processing.

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
                JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
RecordQue   QUEUE,PRE(Que)
AcctNumber    LIKE(Cus:AcctNumber)
Name          LIKE(Cus:Name)
OrderNumber   LIKE(Hea:OrderNumber)
Item          LIKE(Det:Item)
Quantity      LIKE(Det:Quantity)
Description   LIKE(Pro:Description)
Price         LIKE(Pro:Price)
            END
SavPosition STRING(260)
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                    !Top of file in keyed sequence
  LOOP                               !Read all records in file
    NEXT(ViewOrder)                  ! read a record sequentially
      IF ERRORCODE()
        DO DisplayQue
        BREAK
      END
    RecordQue :=: Cus:Record         !Move record into queue
    RecordQue :=: Hea:Record         !Move record into queue
    RecordQue :=: Dtl:Record         !Move record into queue
    RecordQue :=: Pro:Record         !Move record into queue
    ADD(RecordQue)                   ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20       !20 records in queue?
     DO DisplayQue                   !Display the queue
 . .                                 !End loop

DisplayQue  ROUTINE
      SavPosition = POSITION(ViewOrder)     !Save record position
      DO ProcessQue                         !Display the queue
      FREE(RecordQue)                       ! and free it
      RESET(ViewOrder,SavPosition)          !Reset the record pointer
      NEXT(ViewOrder)                       ! and get the record again
```

See Also: POSITION, NEXT, PREVIOUS

SKIP (bypass view records in sequence)

SKIP(*view*,*count***)**

SKIP	Bypasses records during sequential VIEW processing.
view	The label of a VIEW declaration.
count	A numeric constant or variable. The *count* specifies the number of records to bypass. If the value is positive, records are skipped in forward (NEXT) sequence; if *count* is negative, records are skipped in reverse (PREVIOUS) sequence.

The **SKIP** statement is used to bypass records during sequential VIEW processing. It bypasses records (in the sequence specified by the SET statement) by moving the file pointer *count* records. SKIP is more efficient than NEXT or PREVIOUS for skipping past records because it does not move records into the data buffer(s).

If SKIP reads past the end or beginning of VIEW, the EOF and BOF functions return true (if supported by the file system in use). If no SET has been issued, SKIP is ignored.

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
               JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
SavOrderNo  LONG
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                       !Top of file in keyed sequence
  LOOP                                  !Process all records
    NEXT(ViewOrder)                     ! Get a record
      IF ERRORCODE() THEN BREAK.
    IF Hea:OrderNumber <> SavOrderNo    ! Check for first item in order
      IF Hea:OrderStatus = 'Cancel'     !   Is it a canceled order?
        SKIP(Items,Vew:ItemCount-1)     !     SKIP rest of the items
        CYCLE                           !     and process next order
    . .                                 ! end ifs
    DO ItemProcess                      ! process the item
    SavInvNo = Hea:OrderNUmber          ! save the invoice number
  END                                   !End loop
```

WATCH (automatic view concurrency check)

WATCH(*view*)

WATCH Arms automatic optimistic concurrency checking.

view The label of a VIEW declaration.

The **WATCH** statement arms automatic optimistic concurrency checking by the file driver for a following NEXT or PREVIOUS statement in a multi-user environment. Generally, the file driver retains a copy of the retrieved fields from each file on the NEXT or PREVIOUS when it successfully gets the *view*. When the fields are PUT to the *view*, the fields on disk are compared to the original data retrieved. An error is returned by the PUT statement if the data has been changed by another user. The specific action WATCH performs is file driver dependent.

Example:

```
Customer    FILE,DRIVER('Clarion'),PRE(Cus)   !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Record        RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
City            STRING(20)
State           STRING(20)
Zip             STRING(20)
            . .
CustView    VIEW(Customer)                  !Declare VIEW structure
            END
  CODE
  OPEN(Customer,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                           !Top of file in keyed sequence
  LOOP                                      !Process all records
    WATCH(ViewOrder)                        !Arm concurrency check
    NEXT(ViewOrder)                         ! Get a record
      IF ERRORCODE() THEN BREAK.
    DO ItemProcess                          ! process the item
    PUT(ViewOrder)                          ! and put it back
      IF ERRORCODE() THEN STOP(ERROR()).    !Stop on any error, including
                                            ! record changed by another user
  END
```

View Functions

POSITION (return view record sequence position)

POSITION(*sequence***)**

POSITION Identifies a record's unique position in the VIEW.

sequence The label of a VIEW declaration.

POSITION returns a STRING which identifies a record's unique position within the *sequence*. POSITION returns the position of the last record accessed in the VIEW. The POSITION function is used with RESET to temporarily suspend and resume sequential processing.

The return string for POSITION(view) contains the sequence set by the SET statement on the primary file issued immediately before the OPEN(view) statement. It also contains the file system's POSITION return value for the primary file key and all secondary file linking keys. This allows POSITION(view) to accurately define a position for all related records in the VIEW.

As a general rule, for file systems that have record numbers, the size of the STRING returned by POSITION(file) is 4 bytes. The return string from POSITION(key) is 4 bytes plus the sum of the sizes of the fields in the key. For file systems that do not have record numbers, the size of the STRING returned by POSITION(file) is usually the sum of the sizes of the fields in the Primary Key (the first KEY on the FILE that does not have the DUP or OPT attribute). The return string from POSITION(key) is the sum of the sizes of the fields in the Primary Key plus the sum of the sizes of the fields in the key.

Return Data Type: STRING

Example:

```
ViewOrder   VIEW(Customer)    !Declare VIEW structure
              PROJECT(Cus:AcctNumber,Cus:Name)
              JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
                PROJECT(Hea:OrderNumber)
                JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                  PROJECT(Det:Item,Det:Quantity)
                  JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                    PROJECT(Pro:Description,Pro:Price)
                  END
                END
              END
            END
RecordQue   QUEUE,PRE(Que)
AcctNumber    LIKE(Cus:AcctNumber)
Name          LIKE(Cus:Name)
OrderNumber   LIKE(Hea:OrderNumber)
Item          LIKE(Det:Item)
Quantity      LIKE(Det:Quantity)
Description   LIKE(Pro:Description)
Price         LIKE(Pro:Price)
            END
SavPosition STRING(260)
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                          !Top of file in keyed sequence
  LOOP                                     !Read all records in file
    NEXT(ViewOrder)                        ! read a record sequentially
      IF ERRORCODE()
        DO DisplayQue                      !Display the queue
        BREAK
      END
    RecordQue :=: Cus:Record               !Move record into queue
    RecordQue :=: Hea:Record               !Move record into queue
    RecordQue :=: Dtl:Record               !Move record into queue
    RecordQue :=: Pro:Record               !Move record into queue
    ADD(RecordQue)                         ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20             !20 records in queue?
      DO DisplayQue                        !Display the queue
   . .

DisplayQue  ROUTINE
      SavPosition = POSITION(ViewOrder)    !Save record position
      DO ProcessQue                        !Display the queue
      FREE(RecordQue)                      ! and free it
      RESET(ViewOrder,SavPosition)         !Reset the record pointer
      NEXT(ViewOrder)                      ! and get the record again
```

See Also: RESET

Queue Structur e

Contents

QUEUE (declare a memory QUEUE structure)

```
label      QUEUE( [ group ] ) [,PRE] [,STATIC] [,THREAD] [,TYPE] [,BINDABLE] [,EXTERNAL] [,DLL]
fieldlabel          variable [,NAME( )]
           END
```

QUEUE	Declares a memory queue structure.
label	The name of the QUEUE.
group	The label of a previously declared GROUP, QUEUE, or RECORD structure from which it will inherit its structure. This may be a GROUP or QUEUE with the TYPE attribute.
PRE	Declare a *fieldlabel* prefix for the structure.
STATIC	Declares a QUEUE, local to a PROCEDURE or FUNCTION, whose buffer is allocated in static memory.
THREAD	Specify memory for the queue is allocated once for each execution thread. Must be used with the STATIC attribute on Procedure Local data.
TYPE	Specify the QUEUE is a type definition for QUEUEs passed as parameters.
BINDABLE	Specify all variables in the queue may be used in dynamic expressions.
EXTERNAL	Specify the QUEUE is defined, and its memory is allocated, in an external library.
DLL	Specify the QUEUE is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
fieldlabel	The name of the *variables* in the queue.
variable	Data declaration. The sum of the memory required for all declared *variables* in the QUEUE must not be greater than 65,000 bytes in 16-bit applications and 4MB in 32-bit applications.

QUEUE declares a memory QUEUE structure. The *label* of the QUEUE structure is used in queue processing statements and functions. When used in assignment statements, expressions, or parameter lists, a QUEUE is treated like a GROUP data type.

The structure of a QUEUE declared with the *group* parameter begins with the same structure as the named *group*; the QUEUE inherits the fields of the named *group*. The QUEUE may also contain its own *declarations* that follow the inherited fields.

A QUEUE may be thought of as a "memory file" which is internally implemented as a dynamic array of the QUEUE entries. When a QUEUE is declared, a data buffer is allocated (just as with a file). Each entry in the QUEUE occupies exactly the same amount of memory as the data buffer with no per-entry overhead (also no data compression or clipping).

The data buffer for a Procedure local QUEUE (declared in the data section of a PROCEDURE or FUNCTION) is allocated on the stack (unless it has the STATIC attribute or is too large). The memory allocated to the entries in a procedure-local QUEUE without the STATIC attribute is allocated only until you FREE the QUEUE, or you RETURN from the PROCEDURE or FUNCTION—the QUEUE is automatically FREEd upon RETURN.

For a Global data, Module data, or Local data QUEUE with the STATIC attribute, the data buffer is allocated static memory and the data in the buffer is persistent between procedure calls. The memory allocated to the entries in the QUEUE remains allocated until you FREE the QUEUE.

The *variables* in the QUEUE's data buffer are not automatically initialized to any value, they must be explicitly assigned values. Do not assume that they contain blanks or zero before your program's first assignment to them.

As entries are added to the QUEUE, memory for the entry is dynamically allocated and the data is copied from the buffer to the entry. As entries are deleted from the QUEUE, the memory used by the deleted entry is freed. The maximum number of entries in a QUEUE is 1,000,000. The memory used by each entry in the QUEUE is equal to the total of the field sizes.

A QUEUE with the BINDABLE attribute makes all the variables within the QUEUE available for use in a dynamic expression, without requiring a separate BIND statement for each (allowing BIND(queue) to enable all the fields in the queue). The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A QUEUE with the TYPE attribute is not allocated any memory; it is only a type definition for QUEUEs that are passed as parameters to PROCEDUREs or FUNCTIONs. This allows the receiving procedure to directly address component fields in the passed QUEUE. The parameter declaration on the PROCEDURE or FUNCTION statement instantiates a local prefix for the passed QUEUE as it names the passed QUEUE for the procedure. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type declaration) to directly address component fields of the QUEUE actually passed as the parameter.

Example:

```
NameQue   QUEUE,PRE(Nam)                        !Declare a queue
Name        STRING(20)
Zip         DECIMAL(5,0),NAME('SortField')
          END                                   !End queue structure

NameQue2  QUEUE(NameQue),PRE(Nam2)    !Queue that inherits Name and Zip fields
Phone       STRING(10)                ! and adds a Phone field
          END                                   !End queue structure
```

See Also: PRE, STATIC, NAME, FREE, THREAD

PRE (set label prefix)

PRE([*prefix*] **)**

PRE Provides a label prefix for variables declared in the QUEUE.

prefix Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alphabet character and must not be a reserved word.

The **PRE** attribute provides a label prefix for variables declared in the QUEUE. It is used to distinguish between identical variable names that occur in different structures. When referenced in executable statements, assignments, and parameter lists, a *prefix* is attached to a label by a colon (Pre:Label).

Another method to distinguish between identical variable names that occur in different structures does not use the PRE attribute, but instead uses the Field Qualification syntax. When referenced in executable statements, assignments, and parameter lists, the label of the structure containing the field is attached to the field label by a colon (QueueName:Label).

Example:

```
SaveQueue   QUEUE,PRE(Sav)
Field1        LONG                !Referenced as Sav:Field1 or SaveQueue:Field1
Field2        STRING              !Referenced as Sav:Field2 or SaveQueue:Field2
            END
```

See Also: Reserved Words, Field Qualification

STATIC (set local queue static)

STATIC

The **STATIC** attribute allows the data buffer of a QUEUE declared within a PROCEDURE or FUNCTION to be allocated static memory instead of stack memory. This makes any value contained in the data buffer "persistent" from one instance of the procedure to the next.

Example:

```
SomeProc    PROCEDURE
SaveQueue   QUEUE,STATIC        !Static QUEUE data buffer
Field1        LONG              !Value retained between
Field2        STRING            ! procedure calls
            END
```

See Also: Data Declarations and Memory Allocation

THREAD (set thread-specific static queue)

THREAD

The **THREAD** attribute declares a static QUEUE which is allocated memory separately for each execution thread in the program. This makes the values contained in the QUEUE dependent upon which thread is executing. Whenever a new execution thread is begun, a new instance of the QUEUE, specific to that thread, is created.

The THREAD attribute implies a static QUEUE, so the STATIC attribute is unnecessary on a Procedure Local QUEUE. This attribute creates a lot of runtime "overhead," particularly on Global or Module data. Therefore, it should be used only when absolutely necessary.

Example:

```
SomeProc    PROCEDURE
SaveQueue   QUEUE,THREAD        !Static QUEUE data buffer
Field1        LONG              !Thread-specific QUEUE
Field2        STRING
            END
```

See Also: Data Declarations and Memory Allocation

NAME (set queue variable external name)

NAME([*name*] **)**

NAME Specifies an "external" name for queue processing.

name A string constant containing the "external" name of the variable.

The **NAME** attribute on a variable declared in a QUEUE structure specifies an "external" *name* for queue processing. The *name* is an alternate method of addressing the variables in the QUEUE used by the SORT, GET, PUT, and ADD statements.

Example:

```
SortQue   QUEUE,PRE(Que)
Field1      STRING(10),NAME('FirstField')       !QUEUE SORT NAME
Field2      LONG,NAME('SecondField')            !QUEUE SORT NAME
          END
```

See Also: QUEUE, SORT, GET, PUT, ADD

TYPE (QUEUE type definition)

TYPE

The **TYPE** attribute creates a QUEUE that is not allocated any memory; it is only a type definition for QUEUEs that are passed as parameters to PROCEDUREs or FUNCTIONs. This allows the receiving procedure to directly address component fields in the passed QUEUE. The parameter declaration on the PROCEDURE or FUNCTION statement instantiates a local prefix for the passed QUEUE as it names the passed QUEUE for ther procedure. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type definition) to directly address component fields of the QUEUE passed as the parameter.

Example:

```
PassQue   QUEUE,TYPE                !Type-definition for passed QUEUE parameters
F1          STRING(20)              ! first field
F2          STRING(1)               ! middle field
F3          STRING(20)              ! last field
          END

    MAP
      MyProc1(PassQue)              !Passes a QUEUE defined the same as PassGroup
    END

NameQue   QUEUE,PRE(Nme)            !Name queue
First       STRING(20)              ! first name
Middle      STRING(1)               ! middle initial
Last        STRING(20)              ! last name
          END                       !End queue declaration

  CODE
  MyProc1(NameQue)                  !Call proc passing NameQue as parameter

MyProc1     PROCEDURE(LOC:PassedGroup)   !Proc to receive QUEUE parameter
LocalVar    STRING(20)
  CODE
  LocalVar = LOC:F1                 !Assign value in Nme:First to LocalVar
                                    !  from passed parameter
```

BINDABLE (set runtime expression string QUEUE variables)

BINDABLE

The **BINDABLE** attribute on a QUEUE statement declares a structure whose constituent variables are all available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

The BIND(group) form of the BIND statement must still be used in the executable code before the individual fields in the QUEUE structure may be used.

Example:

```
Names     QUEUE,BINDABLE    !Bindable Record structure
Name        STRING(20)
FileName    STRING(8),NAME('FName')       !Dynamic name: FName
Dot         STRING(1)                     !Dynamic name: Dot
Extension   STRING(3),NAME('EXT')         !Dynamic name: EXT
          END
  CODE
  OPEN(Names)
  BIND(Names)
```

See Also: BIND, UNBIND, EVALUATE

EXTERNAL (set queue defined externally)

EXTERNAL

The **EXTERNAL** attribute specifies that the QUEUE on which it is placed is defined in an external library. Therefore, a QUEUE with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory. The memory for the QUEUE is allocated by the external library. This allows the Clarion program access to QUEUEs declared as public in external libraries.

When using EXTERNAL to declare a QUEUE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the QUEUE without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the QUEUE with the EXTERNAL attribute. This ensures that there is only one memory allocation for the QUEUE and all the libraries and the .EXE will reference the same memory when referring to that QUEUE.

The QUEUE declarations in all libraries (or .EXEs) that reference common QUEUEs must be EXACTLY the same (with the appropriate addition of the EXTERNAL attribute). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same variables would have one .DLL containing the actual data definition that only contains FILE and global variable and QUEUE definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common FILEs, QUEUEs, and variables with the EXTERNAL attribute.

Example:

```
Names     QUEUE,EXTERNAL                !A queue declared in an external library
Name        STRING(20)
FileName    STRING(8),NAME('FName')     !Dynamic name: FName
Dot         STRING(1)                   !Dynamic name: Dot
Extension   STRING(3),NAME('EXT')       !Dynamic name: EXT
          END
```

DLL (set queue defined externally in .DLL)

DLL([*flag*] **)**

DLL	Declares a QUEUE defined externally in a .DLL.
flag	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the QUEUE on which it is placed is defined in a .DLL. A QUEUE with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the QUEUE.

The QUEUE declarations in all libraries (or .EXEs) that reference common QUEUEs must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL to declare a QUEUE shared by .DLLs and .EXE, only one .DLL should define the QUEUE without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the QUEUE with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the QUEUE and all the .DLLs and the .EXE will reference the same memory when referring to that QUEUE.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same QUEUEs would have one .DLL containing the actual data definition that only contains FILE and global QUEUE and variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
DLLQueue      QUEUE,PRE(Que),EXTERNAL,DLL     !A queue declared in an external .DLL
  TotalCount    LONG
              END
```

See Also: EXTERNAL

Queue Procedures

ADD (add an entry)

```
                | pointer          |
ADD(queue [,    | [+]key,...,[-]key] | ] )
                | name             |
```

ADD Writes a new entry to the QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

pointer A numeric constant, variable, or numeric expression. The *pointer* must be in the range from 1 to the number of entries in the memory queue.

+ - The leading plus or minus sign specifies the *key* is sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

ADD writes a new entry from the QUEUE structure data buffer to the QUEUE. If there is not enough memory to ADD a new entry, the "Insufficient Memory" error is posted.

ADD(*queue*) Appends a new entry to the end of the QUEUE.

ADD(*queue,pointer*) Places a new entry at the relative position specified by the *pointer* parameter. If there is an entry already at the relative *pointer* position, it is "pushed down" to make room for the new entry. All following pointers are readjusted to account for the new entry. For example, an entry added at position 10 pushes entry 10 to position 11, entry 11 to position 12, etc. If *pointer* is zero or greater than the number of entries in the QUEUE, the entry is added at the end.

ADD(*queue,key*) Inserts a new entry in a sorted memory queue. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching *key* values. If there are no entries, ADD(*queue,key*) may be used to build the QUEUE in sorted order.

ADD(*queue,name*)

Inserts a new queue entry in a sorted memory queue. The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching field values. If there are no entries, ADD(*queue,name*) may be used to build the QUEUE in sorted order.

Errors Posted:

08 Insufficient Memory
75 Invalid Field Type Descriptor

Example:

```
NameQue   QUEUE,PRE(Que)
Name        STRING(20),NAME('FirstField')
Zip         DECIMAL(5,0),NAME('SecondField')
          END
  CODE
  ADD(NameQue)                      !Add an entry at the end of the QUEUE
  ADD(NameQue,1)                    !Add an entry at position 1
  Que:Name = 'Jones'                !Initialize fields
  Que:Zip = 12345
  ADD(NameQue,+Que:Name,-Que:Zip)   !Ascending name, descending zip order
  Que:Name = 'Smith'                !Initialize fields
  Que:Zip = 12345
  ADD(NameQue,'+FirstField,-SecondField')
                                    !Add in name, descending zip order
```

DELETE (delete an entry)

DELETE(*queue*)

DELETE Removes a QUEUE entry.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

DELETE removes the QUEUE entry at the position of the last successful GET or ADD and de-allocates its memory. If no previous GET or ADD was executed, the "Entry Not Found" error is posted. DELETE does not affect the curent POINTER function return value.

Errors Posted: 08 Insufficient Memory
30 Entry Not Found

Example:

```
Que:Name = 'Jones'          !Initialize the key
GET(NameQue,Que:Name)       !Get the matching record
DELETE(NameQue)             !Delete the entry
```

FREE (delete all entries)

FREE(*queue*)

FREE Deletes all entries from a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

FREE deletes all entries from a QUEUE and de-allocates the memory they occupied. It also de-allocates the memory used by the QUEUE's "overhead." FREE does not clear the QUEUE's data buffer.

Errors Posted: 08 Insufficient Memory

Example:

```
FREE(Location)          !Free the location queue
FREE(NameQue)           !Free the name queue
```

GET (read an entry)

```
                | pointer          |
GET(queue ,     | [+]key,...,[-]key] | )
                | name             |
```

GET Retrieves a specific QUEUE entry.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

pointer A numeric constant, variable, or numeric expression. The *pointer* must be in the range from 1 to the number of entries in the memory queue.

+ - The leading plus or minus sign specifies the *key* is sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

GET reads an entry into the QUEUE structure data buffer for processing. If GET does not find a match, the "Entry Not Found" error is posted.

GET(*queue,pointer*) Retrieves the entry at the relative entry position specified by the *pointer* value. If *pointer* is zero, the value returned by the POINTER function is set to zero.

GET(*queue,key*) Searches for a QUEUE entry that matches the value in the *key* field(s). Multiple *key* parameters may be used (up to 16), separated by commas. The QUEUE must already be sorted on the field(s) used as the *key* parameter(s).

GET(*queue,name*) Searches for a QUEUE entry that matches the value in the *name* field(s). The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The QUEUE must already be sorted on the field(s) listed in the *name* parameter.

Errors Posted:

08 Insufficient Memory
30 Entry Not Found
75 Invalid Field Type Descriptor

Example:

```
NameQue   QUEUE,PRE(Que)
Name        STRING(20),NAME('FirstField')
Zip         DECIMAL(5,0),NAME('SecondField')
          END
  CODE
  DO BuildQue                              !Call routine to build the queue
  GET(NameQue,1)                           !Get the first entry
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = 'Jones'                       !Initialize key field
  GET(NameQue,Que:Name)                    !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = Fil:Name                      !Initialize to value in Fil:Name
  GET(NameQue,Que:Name)                    !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = 'Smith'                       !Initialize the key fields
  Que:Zip = 12345
  GET(NameQue,'FirstField,SecondField')    !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).
```

See Also: SORT

PUT (write an entry)

PUT(*queue* , | [**+**]*key*,...,[**-**]*key*] | **)**
| *name* |

PUT	Writes an entry back to the QUEUE.
queue	The label of a QUEUE structure, or the label of a passed QUEUE parameter.
+ -	The leading plus or minus sign specifies the *key* is sorted in ascending or descending sequence.
key	The label of a field declared within the QUEUE structure. If a the QUEUE has a PRE attribute, the *key* must include the prefix.
name	A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

PUT writes the contents of the data buffer back to the QUEUE (after a successful GET or ADD) to the position returned by the POINTER function. If no previous GET or ADD was executed, the "Entry Not Found" error is posted.

PUT(*queue*)	Writes the data buffer back to the same relative position within the QUEUE of the last successful GET or ADD.
PUT(*queue,key*)	Returns an entry to a sorted memory queue after a successful GET or ADD, maintaining the sort order if any *key* fields have changed value. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching *key* values.
PUT(*queue,name*)	Returns an entry to a sorted memory queue after a successful GET or ADD, maintaining the sort order if any key fields have changed value. The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching field values.

Errors Posted:

08 Insufficient Memory
30 Entry Not Found
75 Invalid Field Type Descriptor

Example:

```
NameQue   QUEUE,PRE(Que)
Name        STRING(20),NAME('FirstField')
Zip         DECIMAL(5,0),NAME('SecondField')
          EBD
  CODE
  DO BuildQue                                  !Call routine to build the queue

  Que:Name = 'Jones'                    !Initialize key field
  GET(NameQue,Que:Name)                 !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).
  Que:Zip = 12345                       !Change the zip
  PUT(NameQue)                          !Write the changes to the queue
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = 'Jones'                    !Initialize key field
  GET(NameQue,Que:Name)                 !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).
  Que:Name = 'Smith'                    !Change key field
  PUT(NameQue,Que:Name)                 !Write changes to the queue
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = 'Smith'                    !Initialize key field
  GET(NameQue,'FirstField')             !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).
  Que:Name = 'Jones'                    !Change key field
  PUT(NameQue,'FirstField')             !Write changes to the queue
    IF ERRORCODE() THEN STOP(ERROR()).
```

SORT (sort entries)

SORT(*queue*, | [**+**]*key*,...,[**-**]*key*] | **)**
| *name* |

SORT Reorders entries in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

+ - The leading plus or minus sign specifies the *key* will be sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

SORT reorders the entries in a QUEUE. QUEUE entries with identical key values maintain their relative position.

SORT(*queue,key*) Reorders the QUEUE in the sequence specified by the *key*. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence.

SORT(*queue,name*)
Reorders the QUEUE in the sequence specified by the *name* string. The *name* string must contain the NAME attributes of the fields, separated by commas, with leading plus or minus signs to indicate ascending or descending sequence.

Errors Posted:
08 Insufficient Memory
75 Invalid Field Type Descriptor

Example:

```
Location    QUEUE,PRE(Loc)
Name          STRING(20),NAME('FirstField')
City          STRING(10),NAME('SecondField')
State         STRING(2)
Zip           DECIMAL(5,0)
            END
  CODE
  SORT(Location,Loc:State,Loc:City,Loc:Zip)     !Sort by zip in city in state
  SORT(Location,+Loc:State,-Loc:Zip)            !Sort descending by zip in state
  SORT(Location,'FirstField,-SecondField')      !Sort descending by city in name
```

Queue Functions

POINTER (return last entry position)

POINTER(*queue*)

POINTER Returns the entry number of the last entry accessed in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

The **POINTER** function returns a LONG integer specifying the entry number of the last QUEUE entry accessed by ADD, GET, or PUT.

Return Data Type: LONG

Example:

```
Que:Name = 'Jones'                      !Initialize key field
GET(NameQue,Que:Name)                   !Get the entry
  IF ERRORCODE() THEN STOP(ERROR()).    ! and check for errors
SavPoint = POINTER(NameQue)             !Save the pointer
```

RECORDS (return number of entries)

RECORDS(*queue*)

RECORDS Returns the number of entries in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

The **RECORDS** function returns a LONG integer containing the number of entries in the *queue.*

Return Data Type: LONG

Example:

```
Entries# = RECORDS(Location)        !Determine number of entries
LOOP I# = 1 TO Entries#             !Loop through QUEUE
  GET(Location,I#)                  ! getting each entry
    IF ERRORCODE() THEN STOP(ERROR()).
  DO SomeProcess                  ! process the entry
END
```

Mathematical Functions

Contents

ABS (return absolute value)

ABS(*expression***)**

ABS Returns absolute value.

expression A constant, variable, or expression.

The **ABS** function returns the absolute value of an *expression*. The absolute value of a number is always positive (or zero).

Return Data Type: REAL or DECIMAL

Example:

```
C = ABS(A - B)                !C is absolute value of the difference
IF B < 0 THEN B = ABS(B).     !If b is negative make it positive
```

See Also: BCD Operations and Functions

INRANGE (check number within range)

INRANGE(*expression*,*low*,*high***)**

INRANGE Return number in valid range.

expression A numeric constant, variable, or expression.

low A numeric constant, variable, or expression of the lower boundary of the range.

high A numeric constant, variable, or expression of the upper boundary of the range.

The **INRANGE** function compares a numeric *expression* to an inclusive range of numbers. If the value of the *expression* is within the range, the function returns the value 1 for "true." If the *expression* is greater than the *high* parameter, or less than the *low* parameter, the function returns a zero for "false."

Return Data Type: LONG

Example:

```
IF INRANGE(Date % 7,1,5)    !If this is a week day
  DO WeekdayRate            ! use the weekday rate
ELSE                        !Otherwise
  DO WeekendRate            ! use the weekend rate
END
```

INT (truncate fraction)

INT(*expression***)**

INT Return integer.

expression A numeric constant, variable, or expression.

The **INT** function returns the integer portion of a numeric expression. No rounding is performed, and the sign remains unchanged.

Return Data Type: REAL or DECIMAL

Example:

```
    !INT(8.5)     returns 8
    !INT(-5.9)    returns -5

x = INT(y)            !Return integer portion of y variable contents
```

See Also: BCD Operations and Functions

LOGE (return natural logarithm)

LOGE(*expression***)**

LOGE Returns the natural logarithm.

expression A numeric constant, variable, or expression. If the value of the *expression* is less than zero, the return value is zero. The natural logarithm is undefined for values less than zero.

The **LOGE** (pronounced “log-e”) function returns the natural logarithm of a numeric *expression*. The natural logarithm of a value is the power to which **e** must be raised to equal that value. The value of **e** is 2.71828182846.

Return Data Type: REAL

Example:

```
  !LOGE(2.71828182846)  returns 1
  !LOGE(1)              returns 0

LogVal = LOGE(Val)     !Get the natural log of Val
```

LOG10 (return base 10 logarithm)

LOG10(*expression***)**

LOG10 Returns base 10 logarithm.

expression A numeric constant, variable, or expression. If the value of the *expression* is zero or less, the return value will be zero. The base 10 logarithm is undefined for values less than or equal to zero.

The **LOG10** (pronounced "log ten") function returns the base 10 logarithm of a numeric *expression*. The base 10 logarithm of a value is the power to which 10 must be raised to equal that value.

Return Data Type: REAL

Example:

```
  !LOG10(10)    returns 1
  !LOG10(1)     returns 0
LogStore = LOG10(Var)        !Store the log 10 of var
```

RANDOM (return random number)

RANDOM(*low*,*high***)**

RANDOM Returns random integer.

low A numeric constant, variable, or expression for the lower boundary of the range.

high A numeric constant, variable, or expression for the upper boundary of the range.

The **RANDOM** function returns a random integer between the *low* and *high* parameter values, inclusively. The *low* and *high* parameters may be any numeric expression, but only their integer portion is used for the inclusive range.

Return Data Type: LONG

Example:

```
Num                BYTE,DIM(49)
LottoNbr           BYTE,DIM(6)
  CODE
  CLEAR(Num)
  CLEAR(LottoNbr)
  LOOP X# = 1 TO 6
    LottoNbr[X#] = RANDOM(1,49)      !Pick numbers for Lotto
    IF NOT Num[LottoNbr[X#]]
      Num[LottoNbr[X#]] = 1
    ELSE
      X# -= 1
  . .
```

ROUND (return rounded number)

ROUND(*expression*,*order*)

ROUND Returns rounded value.

expression A numeric constant, variable, or expression.

order A numeric expression with a value equal to a power of ten, such as 1, 10, 100, 0.1, 0.001, etc. If the value is not an even power of ten, the next lowest power is used; 0.55 will use 0.1 and 155 will use 100.

The **ROUND** function returns the value of an *expression* rounded to a power of ten. If the *order* is a LONG or DECIMAL Base Type, then rounding is performed as a BCD operation. Note that if you want to round a real number larger than 1e30, you should use ROUND(num,1.0e0), and not ROUND(num,1). The ROUND function is very efficient ("cheap") as a BCD operation and should be used to compare REALs to DECIMALs at decimal width.

Return Data Type: DECIMAL or REAL

Example:

```
  !ROUND(5163,100)        returns 5200
  !ROUND(657.50,1)        returns 658
  !ROUND(51.63594,.01)    returns 51.64

Commission = ROUND(Price / Rate,.01)   !Round the commission to the nearest cent
```

See Also: BCD Operations and Functions

SQRT (return square root)

SQRT(*expression*)

SQRT Returns square root.

expression A numeric constant, variable, or expression. If the value of the expression is less than zero, the return value is zero.

The **SQRT** function returns the square root of the *expression*. If X represents any positive real number, the square root of X is a number that, when multiplied by itself, produces a product equal to X.

Return Data Type: REAL

Example:

```
Length = SQRT(X^2 + Y^2)  !Find the distance from 0,0 to x,y (pythagorean theorem)
```

Trigonometric Functions

Trigonometric functions return values representing angles and ratios of the sides of a right triangle (a triangle containing a 90-degree angle). The hypotenuse is the side of the triangle opposite the right (90-degree) angle. For either of the other two angles, the adjacent side forms the angle with the hypotenuse, and the opposite side is opposite the angle. (See any good Trigonometry text for further explanation of these terms.)

Angles are expressed in radians. PI is a constant which represents the ratio of the circumference and radius of a circle. There are 2*PI radians (or 360 degrees) in a circle.

The following equates provide high precision constants for PI and the conversion factors between degrees and radians.

```
PI  EQUATE(3.1415926535898)        !The value of PI
Rad2Deg  EQUATE(57.295779513082)   !Number of degrees in a radian
Deg2Rad  EQUATE(.0174532925199)    !Number of radians in a degree
```

SIN (return sine)

SIN(*radians***)**

SIN	Returns sine.
radians	A numeric constant, variable or expression for the angle expressed in radians.

The **SIN** function returns the trigonometric sine of an angle measured in *radians*. The sine is the ratio of the length of the angle's opposite side divided by the length of the hypotenuse.

Return Data Type: REAL

Example:

```
Angle = 45 * Deg2Rad        !Translate 45 degrees to Radians
SineAngle = SIN(Angle)      !Get the sine of 45 degree angle
```

COS (return cosine)

COS(*radians***)**

COS Returns cosine.

radians A numeric constant, variable or expression for the angle in radians.

The **COS** function returns the trigonometric cosine of an angle measured in *radians*. The cosine is the ratio of the length of the angle's adjacent side divided by the length of the hypotenuse.

Return Data Type: REAL

Example:

```
Angle = 45 * Deg2Rad          !Translate 45 degrees to Radians
CosineAngle = COS(Angle)      !Get the cosine of 45 degree angle
```

TAN (return tangent)

TAN(*radians***)**

TAN Returns tangent.

radians A numeric constant, variable or expression for the angle in radians.

The **TAN** function returns the trigonometric tangent of an angle measured in *radians*. The tangent is the ratio of the angle's opposite side divided by its adjacent side.

Return Data Type: REAL

Example:

```
Angle = 45 * Deg2Rad          !Translate 45 degrees to Radians
TangentAngle = TAN(Angle)     !Get the tangent of 45 degree angle
```

ASIN (return arcsine)

ASIN(*expression***)**

ASIN Returns inverse sine.

expression A numeric constant, variable, or expression for the value of the sine.

The **ASIN** function returns the inverse sine. The inverse of a sine is the angle that produces the sine. The return value is the angle in radians.

Return Data Type: REAL

Example:

```
InvSine = ASIN(SineAngle)    !Get the Arcsine
```

See Also: SIN

ACOS (return arccosine)

ACOS(*expression***)**

ACOS Returns inverse cosine.

expression A numeric constant, variable, or expression for the value of the cosine.

The **ACOS** function returns the inverse cosine. The inverse of a cosine is the angle that produces the cosine. The return value is the angle in radians.

Return Data Type: REAL

Example:

```
InvCosine = ACOS(CosineAngle)    !Get the Arccosine
```

See Also: COS

ATAN (return arctangent)

ATAN(*expression***)**

ATAN	Returns inverse tangent.
expression	A numeric constant, variable, or expression for the value of the tangent.

The **ATAN** function returns the inverse tangent. The inverse of a tangent is the angle that produces the tangent. The return value is the angle in radians.

Return Data Type REAL

Example:

```
InvTangent = ATAN(TangentAngle)    !Get the Arctangent
```

See Also: TAN

String Functions

ALL (return repeated characters)

ALL(*string* [,*length*]**)**

ALL	Returns repeated characters.
string	A string expression containing the character sequence to be repeated.
length	The length of the return string. If omitted the *length* of the return string is 255 characters.

The **ALL** function returns a string containing repetitions of the character sequence *string*.

Return Data Type: STRING

Example:

```
Starline = ALL('*',25)        !Get 25 asterisks
Dotline = ALL('.')            !Get 255 dots
```

CENTER (return centered string)

CENTER(*string* [,*length*]**)**

CENTER	Returns centered string.
string	A string constant, variable or expression.
length	The length of the return string. If omitted, the length of the *string* parameter is used.

The **CENTER** function first removes leading and trailing spaces from a *string*, then pads it with leading and trailing spaces to center it within the *length*, and returns a centered string.

Return Data Type: STRING

Example:

```
!CENTER('ABC',5)    returns ' ABC '
!CENTER('ABC  ')    returns ' ABC '
!CENTER(' ABC')     returns ' ABC '

Message = CENTER(Message)       !Center the message
Rpt:Title = CENTER(Name,60)     !Center the name
```

CHR (return character from ASCII)

CHR(*code*)

CHR Returns the display character.

code A numeric expression containing a numeric ASCII character code.

The **CHR** function returns the character represented by the ASCII character *code* parameter.

Return Data Type: STRING

Example:

```
Stringvar = CHR(122)        !Get lower case z
Stringvar = CHR(65)         !Get upper case A
```

CLIP (return string without trailing spaces)

CLIP(*string*)

CLIP Removes trailing spaces.

string A string expression.

The **CLIP** function removes trailing spaces from a *string*. The return string is a substring with no trailing spaces. CLIP is frequently used with the concatenation operator in string expressions.

Return Data Type: STRING

Example:

```
Name = CLIP(Last) & ', ' & CLIP(First) & Init & '.' !Full name in military order
```

DEFORMAT (remove formatting from numeric string)

DEFORMAT(*string* [,*picture*]**)**

DEFORMAT Removes formatting characters from a numeric string.

string A string expression containing a numeric string.

picture A picture token or the label of a CSTRING, STRING, or PSTRING variable containing a picture token (CSTRING is more efficient than STRING or PSTRING). If omitted, the picture for the *string* parameter is used. If the *string* parameter was not declared with a picture token, the return value will contain only characters that are valid for a numeric constant.

The **DEFORMAT** function removes formatting characters from a numeric string, returning only the numbers contained in the string. When used with a dateor time *picture* (except those containing alphabetic characters), it returns a STRING containing the Clarion Standard Date or Time.

Return Data Type: STRING

Example:

```
DialString = 'ATDT1' & DEFORMAT(Phone,@P(###)###-####P) & '<13,10>'
                                        !Get phone number for modem to dial
ClarionDate = DEFORMAT(dBaseDate,@D1)   !Clarion Standard date from mm/dd/yy string
```

FORMAT (format numbers into a picture)

FORMAT(*value*,*picture***)**

FORMAT Returns a formatted numeric string.

value A numeric expression for the *value* to be formatted.

picture A picture token or the label of a STRING, CSTRING, or PSTRING variable containing a picture token (CSTRING is more efficient than STRING or PSTRING).

The **FORMAT** function returns a numeric string formatted according to the *picture* parameter.

Return Data Type: STRING

Example:

```
Rpt:SocSecNbr = FORMAT(Emp:SSN,@P###-##-####P)           !Format the soc-sec-nbr
Phone = FORMAT(DEFORMAT(Phone,@P###-###-####P),@P(###)###-####P)
                                        !Change phone format from dashes to parens
DateString = FORMAT(DateLong,@D1)       !Format a date as a string
```

INLIST (search for entry in list)

INLIST(*searchstring*,*liststring*,*liststring* [,*liststring...*]**)**

INLIST	Returns item in a list.
searchstring	A constant, variable, or expression that contains the value for which to search. If the value is numeric, it is converted to a string before comparisons are made.
liststring	The label of a variable or constant value to compare against the *searchstring*. If the value is numeric, it is converted to a string before comparisons are made. There may be any number of *liststring* parameters, but there must be at least two.

The **INLIST** function compares the contents of the *searchstring* against the values contained in each *liststring* parameter. If a matching value is found, the function returns the number of the *liststring* parameter containing the matching value (relative to the first *liststring* parameter). If the *searchstring* is not found in any *liststring* parameter, **INLIST** returns zero.

Return Data Type: LONG

Example:

```
!INLIST('D','A','B','C','D','E') returns 4
!INLIST('B','A','B','C','D','E') returns 2

 EXECUTE  INLIST(Emp:Status,'Fulltime','Parttime','Retired','Consultant')
   Scr:Message = 'All Benefits'           !Full timer
   Scr:Message = 'Holidays Only'          !Part timer
   Scr:Message = 'Medical/Dental Only'    !Retired
   Scr:Message = 'No Benefits'            !Consultant
 END
```

INSTRING (search for substring)

INSTRING(*substring*,*string* [,*step*] [,*start*]**)**

INSTRING	Searches for a substring in a string.
substring	A string constant, variable, or expression that contains the string for which to search. You should CLIP a variable *substring* so INSTRING will not look for a match that contains the trailing spaces in the variable.
string	A string constant, or the label of the STRING, CSTRING, or PSTRING variable to be searched.
step	A numeric constant, variable, or expression which specifies the step length of the search. A *step* of 1 searches for the *substring* beginning at every character in the *string*, a *step* of 2 starts at every other character, and so on. If *step* is omitted, the step length defaults to the length of the *substring*.
start	A numeric constant, variable, or expression which specifies where to begin the search of the *string*. If omitted, the search starts at the first character position.

The **INSTRING** function *steps* through a *string*, searching for the occurrence of a *substring*. If the *substring* is found, the function returns the *step* number on which the *substring* was found. If the *substring* is not found in the *string*, **INSTRING** returns zero.

Return Data Type: LONG

Example:

```
!INSTRING('DEF','ABCDEFGHIJ',1,1) returns 4
!INSTRING('DEF','ABCDEFGHIJ',2,1) returns 0
!INSTRING('DEF','ABCDEFGHIJ',2,2) returns 2
!INSTRING('DEF','ABCDEFGHIJ',3,1) returns 2

 Extension = SUB(FileSpec,INSTRING('.',FileSpec) + 1,3)
                                    !Extract extension from file spec

 IF  INSTRING(CLIP(Search),Cus:Notes,1,1)   !If search variable found
   Scr:Message = 'Found'              ! display message
 END
```

LEFT (return left justified string)

LEFT(*string* [,*length*]**)**

LEFT	Left justifies a string.
string	A string constant, variable, or expression.
length	A numeric constant, variable, or expression for the length of the return string. If omitted, *length* defaults to the length of the *string*.

The **LEFT** function returns a left justified string. Leading spaces are removed from the *string*.

Return Data Type: STRING

Example:

```
CompanyName = LEFT(CompanyName)     !Left justify the company name
```

LEN (return length of string)

LEN(*string***)**

LEN	Returns length of a string.
string	A string constant, variable, or expression.

The **LEN** function returns the length of a *string*. If the *string* parameter is the label of a variable, the function will return the declared length of the variable. Numeric variables are automatically converted to STRING intermediate values.

Return Data Type: LONG

Example:

```
IF LEN(CLIP(Title) & ' ' & CLIP(First) & ' ' & CLIP(Last)) > 30
                                                    !If full name won't fit
  Rpt:Name = CLIP(Title) & ' ' & SUB(First,1,1) & '. ' & Last
                                                    ! use first initial
ELSE
  Rpt:Name = CLIP(Title) & ' ' & CLIP(First) & ' ' & CLIP(Last)
                                                    ! else use full name
END
Rpt:Title = CENTER(Cus:Name,LEN(Rpt:Title))         !Center the name in the title
```

LOWER (return lower case)

LOWER(*string*)

LOWER Converts a string to all lower case.

string A string constant, variable, or expression for the *string* to be converted.

The **LOWER** function returns a string with all letters converted to lower case.

Return Data Type: STRING

Example:

```
Name = SUB(Name,1,1) & LOWER(SUB(Name,2,19))
                                        !Make the rest of the name lower case
```

NUMERIC (check numeric string)

NUMERIC(*string*)

NUMERIC Validates all numeric string.

string A string constant, variable, or expression.

The **NUMERIC** function returns the value 1 (true) if the *string* contains a valid numeric value. It returns zero (false) if the *string* contains non-numeric characters. Valid numeric characters are the digits 0 through 9, a leading minus sign, and a decimal point.

Return Data Type: LONG

Example:

```
IF NOT NUMERIC(PartNumber)        !If part number is not numeric
  DO ChkValidPart                 ! check for valid part number
END                               !End if
```

RIGHT (return right justified string)

RIGHT(*string* [,*length*]**)**

RIGHT	Right justifies a string.
string	A string constant, variable, or expression.
length	A numeric constant, variable, or expression for the length of the return string. If omitted, the *length* is set to the length of the *string*.

The **RIGHT** function returns a right justified string. Trailing spaces are removed, then the string is right justified and returned with leading spaces.

Return Data Type: STRING

Example:

```
Message = RIGHT(Message)        !Right justify the message
```

SUB (return substring of string)

SUB(*string*,*position*,*length***)**

SUB	Returns a portion of a string.
string	A string constant, variable or expression.
position	A integer constant, variable, or expression. If positive, it points to a character position relative to the beginning of the *string*. If negative, it points to the character position relative to the end of the *string* (i.e., a *position* value of -3 points to a position 3 characters from the end of the *string*).
length	A numeric constant, variable, or expression of number of characters to return.

The **SUB** function parses out a sub-string from a *string* by returning *length* characters from the *string*, starting at *position*.

The SUB function is similar to the "string slicing" operation on STRING, CSTRING, and PSTRING variables, but is less flexible and efficient. "String slicing" is more flexible because it may be used on both the destination and source sides of an assignment statement, while the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of a string, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the string. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Return Data Type: STRING

Example:

```
!SUB('ABCDEFGHI',1,1)  returns 'A'
!SUB('ABCDEFGHI',-1,1) returns 'I'
!SUB('ABCDEFGHI',4,3)  returns 'DEF'
 Extension = SUB(FileName,INSTRING('.',FileName,1,1)+1,3)
                                !Get the file extension using SUB function
Extension = FileName[(INSTRING('.',FileName,1,1)+1):(INSTRING('.',FileName,1,1)+3)]
                                !The same operation using string slicing
```

See Also: INSTRING, STRING, CSTRING, PSTRING, String Slicing

UPPER (return upper case)

UPPER(*string***)**

UPPER	Returns all upper case string.
string	A string constant, variable, or expression for the *string* to be converted.

The **UPPER** function returns a string with all letters converted to upper case.

Return Data Type: STRING

Example:

```
Name = UPPER(Name)      !Make the name upper case
```

VAL (return ASCII value)

VAL(*character***)**

VAL	Returns ASCII code.
character	A one-byte string containing a character.

The **VAL** function returns the ASCII code of a *character*.

Return Data Type: LONG

Example:

```
  !VAL('A')   returns 65
  !VAT('z')   returns 122

CharVal = VAL(StrChar)   !Get the ASCII value of the string character
```

Bit Manipulation Functions

BAND (return bitwise AND)

BAND(*value*,*mask***)**

BAND Performs bitwise AND operation.

value A numeric constant, variable, or expression for the bit *value* to be compared to the bit *mask*. The *value* is converted to a LONG data type prior to the operation, if necessary.

mask A numeric constant, variable, or expression for the bit *mask*. The *mask* is converted to a LONG data type prior to the operation, if necessary.

The **BAND** function compares the *value* to the *mask*, performing a Boolean AND operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where the *value* and the *mask* both contain one (1), and zeroes in all other bit positions.

BAND is usually used to determine whether an individual bit, or multiple bits, are on (1) or off (0) within a variable.

Return Data Type: LONG

Example:

```
!BAND(0110b,0010b) returns 0010b  !0110b = 6, 0010b = 2

RateType  BYTE              !Type of rate
Female    EQUATE(0001b)     !Female mask
Male      EQUATE(0010b)     !Male mask
Over25    EQUATE(0100b)     !Over age 25 mask
  CODE
  IF BAND(RateType,Female)    |     !If female
      AND BAND(RateType,Over25)     ! and over 25
    DO BaseRate                     ! use base premium
  ELSIF BAND(RateType,Male)         !If male
    DO AdjBase                      ! adjust base premium
  END
```

BOR (return bitwise OR)

BOR(*value*,*mask***)**

BOR	Performs bitwise OR operation.
value	A numeric constant, variable, or expression for the bit *value* to be compared to the bit *mask*. The *value* is converted to a LONG data type prior to the operation, if necessary.
mask	A numeric constant, variable, or expression for the bit *mask*. The *mask* is converted to a LONG data type prior to the operation, if necessary.

The **BOR** function compares the *value* to the *mask*, performing a Boolean OR operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where the *value*, or the *mask*, or both, contain a one (1), and zeroes in all other bit positions.

BOR is usually used to unconditionally turn on (set to one), an individual bit, or multiple bits, within a variable.

Return Data Type: LONG

Example:

```
!BOR(0110b,0010b) returns 0110b  !0110b = 6, 0010b = 2

RateType  BYTE              !Type of rate
Female    EQUATE(0001b)     !Female mask
Male      EQUATE(0010b)     !Male mask
Over25    EQUATE(0100b)     !Over age 25 mask
  CODE
  RateType = BOR(RateType,Over25)   !Turn on over 25 bit
  RateType = BOR(RateType,Male)     !Set rate to male
```

BXOR (return bitwise exclusive OR)

BXOR(*value*,*mask***)**

BXOR	Performs bitwise exclusive OR operation.
value	A numeric constant, variable, or expression for the bit *value* to be compared to the bit *mask*. The *value* is converted to a LONG data type prior to the operation, if necessary.
mask	A numeric constant, variable, or expression for the bit *mask*. The *mask* is converted to a LONG data type prior to the operation, if necessary.

The **BXOR** function compares the *value* to the *mask*, performing a Boolean XOR operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where either the *value* or the *mask* contain a one (1), but not both. Zeroes are returned in all bit positions where the bits in the *value* and *mask* are alike.

BXOR is usually used to toggle on (1) or off (0) an individual bit, or multiple bits, within a variable.

Return Data Type: LONG

Example:

```
!BXOR(0110b,0010b) returns 0100b   !0110b = 6, 0100b = 4, 0010b = 2

RateType  BYTE              !Type of rate
Female    EQUATE(0001b)     !Female mask
Male      EQUATE(0010b)     !Male mask
Over25    EQUATE(0100b)     !Over age 25 mask
Over65    EQUATE(1100b)     !Over age 65 mask
  CODE
  RateType = BXOR(RateType,Over65)    !Toggle over 65 bits
```

BSHIFT (return shifted bits)

BSHIFT(*value*,*count***)**

BSHIFT	Performs the bit shift operation.
value	A numeric constant, variable, or expression. The *value* is converted to a LONG data type prior to the operation, if necessary.
count	A numeric constant, variable, or expression for the number of bit positions to be shifted. If *count* is positive, *value* is shifted left. If *count* is negative, *value* is shifted right.

The **BSHIFT** function shifts a bit *value* by a bit *count*. The bit value may be shifted left (toward the high order), or right (toward the low order). Zero bits are supplied to fill vacated bit positions when shifting.

Return Data Type: LONG

Example:

```
  !BSHIFT(0110b,1)    returns 1100b
  !BSHIFT(0110b,-1)   returns 0011b

Varswitch = BSHIFT(20,3)              !Multiply by eight
Varswitch = BSHIFT(Varswitch,-2)      !Divide by four
```

Date / Time Procedures and Functions

Standard Date

A Clarion standard date is the number of days that have elapsed since December 28, 1800. The range of accessible dates is from January 1, 1801 (standard date 4) to December 31, 2099 (standard date 109,211). Date functions will not return correct values outside the limits of this range. The standard date calendar also adjusts for each leap year within the range of accessible dates. Dividing a standard date by modulo 7 gives you the day of the week: zero = Sunday, one = Monday, etc.

The LONG data type with a date format (@D) display picture is normally used for a standard date. The DATE data type is a data format used in the Btrieve Record Manager. A DATE field is internally converted to LONG containing the Clarion standard date before any mathematical or date function operation is performed. Therefore, DATE should be used for external Btrieve file compatibility, and LONG should normally be used for other dates.

Standard Time

A Clarion standard time is the number of hundredths of a second that have elapsed since midnight, plus one (1). The valid range is from 1 (defined as midnight) to 8,640,000 (defined as 11:59:59:99). A standard time of one is exactly equal to midnight (which allows a zero value to be used to detect no time entered). Although time is expressed to the nearest hundredth of a second, the system clock is only updated 18.2 times a second (approximately every 5.5 hundredths of a second).

The LONG data type with a time format (@T) display picture is normally used for a standard time. The TIME data type is a data format used in the Btrieve Record Manager. A TIME field is internally converted to LONG containing the Clarion standard time before any mathematical or time function operation is performed. Therefore, TIME should be used for external Btrieve file compatibility, and LONG should normally be used for other times.

TODAY (return system date)

TODAY( )

The **TODAY** function returns the DOS system date as a standard date. The range of possible dates is from January 1, 1801 (standard date 4) to December 31, 2099 (standard date 109,211).

Return Data Type: LONG

Example:

```
OrderDate = TODAY()     !Set the order date to system date
```

SETTODAY (set system date)

SETTODAY(*date*)

SETTODAY Sets the DOS system date.

date A numeric constant, variable, or expression for a standard date.

The **SETTODAY** statement sets the DOS system date.

Example:

```
SETTODAY(TODAY() + 1)     !Set the date ahead one day
```

CLOCK (return system time)

CLOCK( )

The **CLOCK** function returns the time of day from the DOS system time in standard time (expressed as hundredths of a second since midnight). Although the time is expressed to the nearest hundredth of a second, the system clock is only updated 18.2 times a second (approximately every 5.5 hundredths of a second).

Return Data Type: LONG

Example:

```
Time = CLOCK()        !Save the system time
```

SETCLOCK (set system time)

SETCLOCK(*time***)**

SETCLOCK Sets the DOS system clock.

time A numeric constant, variable, or expression for a standard time (expressed as hundredths of a second since midnight plus one).

The **SETCLOCK** statement sets the DOS system time of day.

Example:

```
SETCLOCK(1)         !Set clock to midnight
```

DATE (return standard date)

DATE(*month,day,year***)**

DATE Return standard date.

month A numeric constant, variable, or expression for the *month*.

day A numeric constant, variable, or expression for the *day* of the month.

year A numeric constant, variable or expression for the *year*. The valid range for a *year* value is 00 through 99 (which assumes the range 1900 - 1999), or 1801 through 2099.

The **DATE** function returns a standard date for a given *month*, *day*, and *year*. The *month* and *day* parameters allow out-of-range values. A *month* value of 13 is interpreted as January of the next year. A *day* value of 32 in January is interpreted as the first of February. Consequently, DATE(12,32,87), DATE(13,1,87), and DATE(1,1,88) all produce the same result.

Return Data Type: LONG

Example:

```
HireDate = DATE(Hir:Month,Hir:Day,Hir:Year) !Compute hire date
```

See Also: Standard Date

DAY (return day of month)

DAY(*date*)

DAY Returns day of month.

date A numeric constant, variable, expression, or the label of a STRING, CSTRING, or PSTRING variable declared with a date picture token. The *date* must be a standard date. A variable declared with a date picture token is automatically converted to a standard date intermediate value.

The **DAY** function computes the day of the month (1 to 31) for a given standard date.

Return Data Type: LONG

Example:

```
OutDay = DAY(TODAY())     !Get the day from today's date
DueDay = DAY(TODAY()+2)   !Calculate the return day
```

See Also: Standard Date

MONTH (return month of date)

MONTH(*date*)

MONTH Returns month in year.

date A numeric constant, variable, expression, or the label of a STRING, CSTRING, or PSTRING variable declared with a date picture token. The *date* must be a standard date. A variable declared with a date picture token is automatically converted to a standard date intermediate value.

The **MONTH** function returns the month of the year (1 to 12) for a given standard date.

Return Data Type: LONG

Example:

```
PayMonth = MONTH(DueDate)    !Get the month from the date
```

See Also: Standard Date

YEAR (return year of date)

YEAR(*date***)**

YEAR	Returns the year.
date	A numeric constant, variable, expression, or the label of a string variable declared with a date picture, containing a standard date. A variable declared with a date picture is automatically converted to a standard date intermediate value.

The **YEAR** function returns a four digit number for the year of a standard *date* (1801 to 2099).

Return Data Type: LONG

Example:

```
IF YEAR(LastOrd) < YEAR(TODAY())   !If last order date not from this year
  DO StartNewYear      !  start new year to date totals
END
```

See Also: Standard Date

AGE (return age from base date)

AGE(*birthdate* [,*base date*]**)**

AGE	Returns elapsed time.
birthdate	A numeric expression for a standard date.
base date	A numeric expression for a standard date. If this parameter is omitted, the system date from DOS is used for the computation.

The **AGE** function returns a string containing the time elapsed between two dates. The age return string is in the following format:

```
1 to 60 days   - 'nn DAYS'
61 days to 24 months - 'nn MOS'
2 years to 999 years - 'nnn YRS'
```

Return Data Type: STRING

Example:

```
Message = Emp:Name & 'is ' & AGE(Emp:DOB,TODAY()) & ' old today.'
```

Operating System Procedures and Functions

COMMAND (return command line)

COMMAND(*flag* **)**

COMMAND Returns command line parameters.

flag A string constant or variable containing the parameter for which to search, or the number of the command line parameter to return.

The **COMMAND** function returns the value of the *flag* parameter from the command line. If the *flag* is not found, COMMAND returns an empty string. If the *flag* is multiply defined, only the first occurrence encountered is returned.

COMMAND searches the command line for *flag=value* and returns *value*. There must be no blanks between *flag*, the equal sign, and *value*. The returned *value* terminates at the first comma or blank space. If a blank or comma is desired in a command line parameter, everything to the right of the equal sign must be enclosed in double quotes (*flag=*"*value*").

COMMAND will also search the command line for a *flag* containing a leading slash (/). If found, COMMAND returns the value of *flag* without the slash. If the *flag* only contains a number, COMMAND returns the parameter at that numbered position on the command line. A *flag* of '0' returns the minimum path DOS used to find the command. This minimum path always includes the command (without command line parameters) but may not include the path (if DOS found it in the current directory). A *flag* containing '1' returns the first command line parameter. If *flag* is an empty string (''), all command parameters are returned as entered on the command line, appended to a leading space.

Return Data Type: STRING

Example:

```
IF COMMAND('/N')                !Was /N on the command line?
  DO SomeProcess
END
IF COMMAND('Option') = '1'      !Was Option=1 on the command line?
  DO OneProcess
END
CommandString = COMMAND('')     !Get all command parameters
CommandItself = COMMAND('0')    !Get the command itself
SecondParm = COMMAND('2')       !Get second parameter from command line
```

See Also: SETCOMMAND

DIRECTORY (get file directory)

DIRECTORY(*queue, path, attributes* **)**

DIRECTORY	Gets a file directory listing (just like the DIR command in DOS).
queue	The label of the QUEUE structure that will receive the directory listing. This must be exactly the same structure as the ff_:queue structure in the EQUATES.CLW file.
path	A string constant, variable, or expression that specifies the path and filenames directory listing to get. This may include the wildcard characters (* and ?).
attributes	An integer constant, variable, or expression that specifies the attributes of the files to place in the *queue.*

The **DIRECTORY** procedure returns a directory listing of all files in the *path* with the specified *attributes* into the specified *queue*.

The *queue* parameter must name a QUEUE with a structure that begins the same as the following structure contained in EQUATES.CLW:

```
ff_:queue   QUEUE,PRE(ff_),TYPE
name          STRING(13)
date          LONG
time          LONG
size          LONG
attrib        BYTE
            END
```

Your QUEUE may contain more fields, but must begin with these five fields. It will receive the returned information about each file in the *path* that has the *attributes* you specify. The date and time fields will contain standard Clarion date and time information (the conversion from the operating system's storage format to Clarion standard format is automatic).

The *attributes* parameter is a bitmap which specifies what filenames to place in the *queue*. The following equates are contained in EQUATES.CLW:

```
ff_:NORMAL        EQUATE(0)
ff_:READONLY      EQUATE(1)
ff_:HIDDEN        EQUATE(2)
ff_:SYSTEM        EQUATE(4)
ff_:DIRECTORY     EQUATE(10H)
ff_:ARCHIVE       EQUATE(20H)      ! NOT Win95 compatible
```

The *attributes* bitmap is a non-exclusive OR filter: if you add the equates, you get files with any of the attributes you specify. This means that, when you just set the ff_:NORMAL attribute, you only get files (no sub-directories) without the read-only, hidden, system, or archive attributes set. If you add ff_:DIRECTORY to ff_:NORMAL, you will get files AND sub-directories from the *path*.

Example:

```
DirectoryList PROCEDURE

AllFiles      QUEUE,PRE(FIL)
name            STRING(13)
date            LONG
time            LONG
size            LONG
attrib          BYTE
              END
LP            LONG
Recs          LONG

  CODE
  DIRECTORY(AllFiles,'*.*',ff_:DIRECTORY)   !Get all files and directories
  Recs = RECORDS(AllFiles)
  LOOP LP = 1 to Recs
    GET(AllFiles,LP)
    IF BAND(FIL:Attrib,ff_:DIRECTORY) AND FIL:Name <> '..' AND FIL:Name <> '.'
      CYCLE                             !Let sub-directory entries stay
    ELSE
      DELETE(AllFiles)                  !Get rid of all other entries
    END
  END
```

PATH (return current directory)

PATH( )

PATH returns a string containing the current drive and directory.

Return Data Type: STRING

Example:

```
IF PATH() = 'C:\'                              !If in the root
  MESSAGE('You are in the Root Directory')     ! display message
END
```

See Also: SETPATH

RUNCODE (return program exit code)

RUNCODE( )

The **RUNCODE** function returns the exit code passed to the operating system from the command executed by the RUN statement. This is the exit code passed by the HALT statement in Clarion programs and is the same as the DOS ERRORLEVEL. RUNCODE returns a LONG integer which may be any value that is returned to DOS as an exit code by the child program.

The child program may only supply a BYTE value as an exit code, therefore negative values are not possible as exit codes. This fact allows RUNCODE to reserve these values to handle situations in which an exit code is not available:

```
 0 normal termination
-1 program terminated with Ctrl-C
-2 program terminated with Critical error
-3 TSR exit
-4 program did not run (check ERROR())
```

Return Data Type: LONG

Example:

```
RUN('Nextprog.exe')                    !Run next program
IF RUNCODE() = -4
  IF ERROR() = 'Not Enough Memory'     !If program didn't run for lack of memory
    MESSAGE('Insufficient memory')     ! display a message
    RETURN                             ! and terminate the procedure
  ELSE
    STOP(ERROR())                      ! terminate program
. .
```

See Also: RUN, HALT

SETCOMMAND (set command line parameters)

SETCOMMAND(*commandline***)**

SETCOMMAND Internally sets command line parameters.

commandline A string constant, variable, or expression containing the new command line parameters.

SETCOMMAND allows the program to internally specify command line parameters that may be read by the COMMAND function. SETCOMMAND overwrites any previous command line flag of the same value. To turn off a leading slash flag, append an equal sign (=) to it in the *commandline*.

SETCOMMAND may not be used to set system level switches which must be specified when the program is loaded. The temporary files directory switch (CLATMP=) may be set with SETCOMMAND.

Example:

```
SETCOMMAND(' /N')       !Add /N parameter
SETCOMMAND(' /N=')      !Turn off /N parameter
```

See Also: COMMAND

SETPATH (change current drive and directory)

SETPATH(*path***)**

SETPATH Changes the current drive and directory.

path A string constant or the label of a STRING, CSTRING, or PSTRING variable containing a new drive and/or directory specification.

SETPATH changes the current drive and directory. If the *drive and path* entry is invalid, the "Path Not Found" error is posted, and the current directory is not changed.

If the drive letter and colon are omitted from the *path*, the current drive is assumed. If only a drive letter and colon are in the *path*, SETPATH changes to the current directory of that drive.

Errors Posted: 03 Path Not Found

Example:

```
SETPATH('C:\LEDGER')        !Change to the ledger directory
SETPATH(UserPath)           !Change to the user's directory
```

Error Reporting Functions

ERROR (return error message)

ERROR( )

The **ERROR** function returns a string containing a description of any error that was posted. If no error was posted, ERROR returns an empty string.

Return Data Type: STRING

Example:

```
PUT(NameQueue)                          !Write the record
IF ERROR() = 'Queue Entry Not Found'    !If not found
  ADD(NameQueue)                        ! add new entry
    IF ERRORCODE() THEN STOP(ERROR()).  !Check for unexpected error
END
```

ERRORCODE (return error code number)

ERRORCODE( )

The **ERRORCODE** function returns the code number for any error that was posted. If no error was posted, ERRORCODE returns zero.

Return Data Type: LONG

Example:

```
ADD(Location)                   !Add new entry
IF ERRORCODE() = 8              !If not enough memory
  MESSAGE('Out of Memory')      ! display message
END
```

ERRORFILE (return error filename)

ERRORFILE( )

The **ERRORFILE** function returns the name of the file for which an error was posted. If the file is open, the full DOS file specification is returned. If the file is not open, the contents of the FILE statement's NAME attribute is returned. If the file is not open and the file has no NAME attribute, the label of the FILE statement is returned. If no error was posted, or the posted error did not involve a file, ERRORFILE returns an empty string.

Return Data Type: STRING

Example:

```
ADD(Location)      !Add new entry
IF ERRORCODE()
  MESSAGE('Error with ' & ERRORFILE())  !Display error filename
END
```

FILEERROR (return file driver error message)

FILEERROR(*file* **)**

The **FILEERROR** function returns a string containing the "native" error message from the file system (file driver) being used to access a data file. If no error was posted, FILEERROR returns an empty string.

Return Data Type: STRING

Example:

```
PUT(NameFile)      !Write the record
IF FILEERRORCODE()
  MESSAGE(FILERROR())
  RETURN
END
```

See Also: FILEERRORCODE

FILEERRORCODE (return file driver error code number)

FILEERRORCODE( )

The **FILEERRORCODE** function returns a string containing the code number for the "native" error message from the file system (file driver) being used to access a data file. If no error was posted, FILEERRORCODE returns an empty string.

Return Data Type: STRING

Example:

```
PUT(NameFile)      !Write the record
IF FILEERRORCODE()
  MESSAGE(FILERROR())
  RETURN
END
```

See Also: FILEERROR

REJECTCODE (return reject code number)

REJECTCODE( )

The **REJECTCODE** function returns the code number for the reason any EVENT:Rejected that was posted. If no EVENT:Rejected was posted, REJECTCODE returns zero. The EQUATES.CLW file contains equates for the values returned by REJECTCODE:

```
REJECT:RangeHigh    ! Above the top range on a SPIN
REJECT:RangeLow     ! Below the bottom range on a SPIN
REJECT:Range        ! Other range error
REJECT:Invalid      ! Invalid input
```

Return Data Type: LONG

Example:

```
CASE EVENT()
OF EVENT:Rejected
  EXECUTE REJECTCODE()
    MESSAGE('Input invalid -- out of range -- too high')
    MESSAGE('Input invalid -- out of range -- too low')
    MESSAGE('Input invalid -- out of range')
    MESSAGE('Input invalid')
  END
END
```

Miscellaneous Procedures and Functions

ADDRESS (return a memory address)

```
ADDRESS(    | segment,offset | )
            | variable     |
            | procedure      |
```

ADDRESS Returns memory address of a variable.

segment The label of a data element, or an integer variable or constant containing the segment portion of a segment:offset real-mode absolute memory address.

offset An integer variable or constant containing the offset portion of a segment:offset real-mode absolute memory address.

variable The label of a data element.

procedure The label of a PROCEDURE or FUNCTION.

The **ADDRESS** function returns a LONG integer containing a memory address in selector:offset format, where the selector is a reference into the protected mode lookup table.

ADDRESS(*segment,offset*)
Returns the protected mode selector:offset for the real mode address specified by the *segment* and *offset* parameters. This allows protected mode direct memory access without incurring a protection violation.

ADDRESS(*variable*)
Returns the protected mode address of the data element specified by the *variable* parameter.

ADDRESS(*procedure*)
Returns the protected mode address of the PROCEDURE or FUNCTION specified by the *procedure* parameter.

Tha ADDRESS function allows you to pass the address of a *variable* or *procedure* to external libraries written in other languages.

Return Data Type: LONG

Example:

```
 MAP
   ClarionProc                    !A Clarion language procedure
   MODULE('External.Obj')         !An external library
     ExternVarProc(LONG)          !C function receiving variable address
     ExternProc(LONG)             !C function receiving procedure address
 . .

Var1 CSTRING(10)   !Define a null-terminated string

  CODE
  ExternVarProc(ADDRESS(Var1))        !Pass address of Var1 to external procedure
  ExternProc(ADDRESS(ClarionProc))    !Pass address of ClarionProc

ClarionProc PROCEDURE                 !A Clarion language procedure
  CODE
  RETURN
```

BEEP (sound tone on speaker)

BEEP([*sound*] **)**

BEEP	Generates a sound through the system speaker.
sound	A numeric constant, variable, expression, or EQUATE for the Windows sound to issue.

The **BEEP** statement generates a sound through the system speaker. These are standard Windows sounds available through the [sounds] section of the WIN.INI file. Standard EQUATE values similar to these are listed in the EQUATES.CLW file:

BEEP:SystemDefault
BEEP:SystemHand
BEEP:SystemQuestion
BEEP:SystemExclamation
BEEP:SystemAsterisk

Example:

```
IF ERRORCODE()                          !If unexpected error
  BEEP(BEEP:SystemDefault)              ! sound a standard beep
  STOP(ERROR())                         ! stop for the error
END
```

CALL (call procedure from a DLL)

CALL(*file, procedure* **)**

CALL	Calls a procedure that has not been prototyped in the application's MAP structure from a Windows standard .DLL.
file	A string constant, variable, or expression containing the name (including extension) of the .DLL to open. This may include a full path.
procedure	A string constant, variable, or expression containing the name of the *procedure* to call (which may not receive parameters or return a value). This can also be the ordinal number indicating the *procedure's* position within the .DLL.

The **CALL** function calls a *procedure* from a Windows-standard .DLL. The *procedure* does not need to be prototyped in the application's MAP structure. If it is not already loaded by Windows, the .DLL *file* is loaded into memory.

CALL returns zero (0) for a successful *procedure* call, or one of the following error values:

```
2   File not found
3   Path not found
5   Attempted to load a task, not a .DLL
6   Library requires separate data segments for each task
10  Wrong Windows version
11  Invalid .EXE file (DOS file or error in program header)
12  OS/2 application
13  DOS 4.0 application
14  Unknown .EXE type
15  Attempt to load an .EXE created for an earlier version of Windows.
    This error will not occur if Windows is run in Real mode.
16  Attempt to load a second instance of an .EXE file containing
    multiple, writeable data segments.
17  EMS memory error on the second loading of a .DLL
18  Attempt to load a protected-mode-only application while Windows is
    running in Real mode
```

Return Data Type: LONG

Example:

```
X# = CALL('CUSTOM.DLL','1')     !Call first procedure in CUSTOM.DLL
IF X# THEN STOP(X#).            !Check for successful execution
```

MAXIMUM (return maximum subscript value)

MAXIMUM(*variable*,*subscript***)**

MAXIMUM	Returns maximum subscript value.
variable	The label of a variable declared with a DIM attribute.
subscript	A numeric constant, variable, or expression for the subscript number. The *subscript* identifies which array dimension is passed to the function.

The **MAXIMUM** function returns the maximum subscript value for an explicitly dimensioned variable. MAXIMUM does not operate on the implicit array dimension of STRING, CSTRING, or PSTRING variables. This is usually used to determine the size of an array passed as a parameter to a procedure or function.

Return Data Type: LONG

Example:

```
Array BYTE,DIM(10,12)         !Define a two-dimensional array

    !For the above Array:     MAXIMUM(Array,1) returns 10
    !                         MAXIMUM(Array,2) returns 12

  CODE
  LOOP X# = 1 TO MAXIMUM(Array,1)       !Loop until end of 1st dimension
    LOOP Y# = 1 TO MAXIMUM(Array,2)     ! Loop until end of 2nd dimension
      Array[X#,Y#] = 27                 !   Initialize each element to default
  . .
```

See Also: DIM, Arrays as Parameters of PROCEDUREs and FUNCTIONs

NAME (return DOS file or device name)

NAME(*label***)**

NAME	Returns name of a file.
label	The label of a FILE declaration.

The **NAME** function returns a string containing the DOS device name for the structure identified by the *label*. For FILE structures, if the file is OPEN, the complete DOS file specification (drive, path, name, and extension) is returned. If the FILE is closed, the contents of the NAME attribute on the FILE are returned.

Return Data Type: STRING

Example:

```
OpenFile = NAME(Customer)  !Save the name of the open file
```

OMITTED (check omitted parameters)

OMITTED(*position*)

OMITTED Tests for unpassed parameters.

position An integer constant or variable which specifies the parameter to test.

The **OMITTED** function tests whether a parameter of a PROCEDURE or FUNCTION was actually passed. The return value is 1 (true) if the parameter in the specified *position* was omitted. The return value is zero (false) if the parameter was passed. Any *position* past the last parameter passed is considered omitted.

A parameter may only be omitted if its data type is enclosed in angle brackets (< >) in the PROCEDURE or FUNCTION prototype in the MAP structure.

Return Data Type: LONG

Example:

```
    PROGRAM
    MAP
     SomeProc(STRING,<LONG>,STRING)         !Procedure prototype
     SomeFunction(STRING,<LONG>),STRING     !Function prototype
    END
    CODE
    SomeProc(Field1,,Field3)
      !For this statement:
      ! OMITTED(1) returns 0
      ! OMITTED(2) returns 1
      ! OMITTED(3) returns 0
      ! OMITTED(4) returns 1

SomeProc  PROCEDURE(Field1,Date,Field3)
  CODE
  IF OMITTED(2)         !If date parameter was omitted
    Date = TODAY()      ! substitute the system date
  END
```

See Also: FUNCTION and PROCEDURE Prototypes

PEEK (read memory address)

PEEK(*segment:offset*,*destination*)

PEEK	Reads data from a memory address.
segment:offset	A numeric constant, variable, or expression which specifies a memory address. The segment must be in the high order two bytes, and the offset in the low order two bytes. The integer portion of a REAL is the data type used for the intermediate value, to assure 32 bit precision. This parameter should always use the ADDRESS function, to ensure the correct protected mode selector:offset address is used.
destination	The label of a variable to receive the contents of the memory location.

The **PEEK** statement reads the memory address at *segment:offset* and copies the data found there into the *destination* variable. PEEK reads as many bytes as are required to fill the *destination* variable.

It is easily possible to create a General Protection Fault (GPF) if you PEEK at an address that belongs to another program, so great care should be taken when using PEEK. There are usually Windows API functions that will do whatever you require of PEEK and these should be used in preference to PEEK.

Example:

```
Segment       USHORT
Offset        USHORT
Dest1         BYTE
Dest2         SHORT
Dest3         REAL
KeyboardFlag  BYTE

  CODE
  PEEK(ADDRESS(Segment,Offset),Dest1)        !Read 1 byte

  PEEK(ADDRESS(Segment,Offset),Dest2)        !Read 2 bytes

  PEEK(ADDRESS(Segment,Offset),Dest3)        !Read 8 bytes

  PEEK(ADDRESS(0040h,0017h),KeyboardFlag)    !Read keyboard status byte
```

See Also: POKE, ADDRESS

POKE (write to memory address)

POKE(*segment:offset*,*source***)**

POKE Writes data to a memory address.

segment:offset A numeric constant, variable, or expression which specifies a memory address. The segment must be in the high order two bytes, and the offset in the low order two bytes. The integer portion of a REAL is the data type used for the intermediate value, to assure 32 bit precision. This parameter should always use the ADDRESS function, to ensure the correct protected mode selector:offset address is used.

source The label of a variable.

The **POKE** statement writes the contents of the *source* variable to the memory address at *segment:offset*. POKE writes as many bytes as are in the *source* variable.

It is easily possible to create a General Protection Fault (GPF) if you POKE to an address that belongs to another program, so great care should be taken when using POKE. There are usually Windows API functions that will do whatever you require of POKE and these should be used in preference to POKE.

Example:

```
Segment       USHORT
Offset        USHORT
Source1       BYTE
Source2       SHORT
Source3       REAL
KeyboardFlag  BYTE

  CODE
  POKE(ADDRESS(Segment,Offset),Source1)      !Write 1 byte to the memory location

  POKE(ADDRESS(Segment,Offset),Source2)      !Write 2 bytes to the memory location

  POKE(ADDRESS(Segment,Offset),Source3)      !Write 8 bytes to the memory location

  PEEK(ADDRESS(0040h,0017h),KeyboardFlag)    !Read keyboard status byte
  KeyboardFlag = BOR(KeyboardFlag,40h)       !  turn on caps lock
  POKE(ADDRESS(0040h,0017h),KeyboardFlag)    !  and put it back
```

See Also: PEEK, ADDRESS

Dynamic Data Exchange

Contents

Overview

Dynamic Data Exchange (DDE) is a very powerful Windows tool that allows a user to access data from another separately executing Windows application. This allows the user to work with the data in its native format (in the originating application), while ensuring that the application in which the data is used always has the most current values.

DDE is based upon establishing "conversations" (links) between two concurrently executing Windows applications. One of the applications acts as the DDE server to provide the data, and the other is the DDE client that receives the data. A single application may be both a DDE client and server, getting data from other applications and providing data to other applications. Multiple DDE "conversations" can occur concurrently between any given DDE server and client.

To be a DDE server, a Clarion application must:

- Open at least one window, since all DDE servers must be associated with a window.
- Register with Windows as a DDE server, using the DDESERVER function.
- Provide the requested data to the client, using the DDEWRITE statement.
- When DDE is no longer required, terminate the link by using the DDECLOSE statement. You can also allow it to terminate automatically when the user closes the server application or the window that started the link.

To be a DDE client, a Clarion application must:

- Open a link to a DDE server as its client, using the DDECLIENT function.
- Ask the server for data, using the DDEREAD statement, or ask the server for a service using the DDEEXECUTE statement.
- When DDE is no longer required, terminate the link by using the DDECLOSE statement. You can also allow it to terminate automatically when the user closes the client window or application.

The DDE functions are prototyped in the DDE.CLW file, which you must INCLUDE in your program's MAP structure. The DDE process posts DDE-specific field-independent events to the ACCEPT loop of the window that opened the link between applications as a server or client.

DDE Events

The DDE process is governed by several field-independent events specific to DDE. These events are posted to the ACCEPT loop of the window that opened the link between applications, either as a server or client.

The following events are posted only to a Clarion server application:

EVENT:DDErequest
: A client has requested a data item.

EVENT:DDEadvise
: A client has requested continuous updates of a data item.

EVENT:DDEexecute
: A client has executed a DDEEXECUTE statement.

EVENT:DDEpoke
: A client has sent unsolicited data

The following events are posted only to a Clarion client application:

EVENT:DDEdata
: A server has supplied an updated data item.

EVENT:DDEclose
: A server has terminated the DDE link.

When one of these DDE events occur there are several functions that tell you what posted the event:

- DDECHANNEL() returns the handle of the DDE server or client.
- DDEITEM() returns the item or command string passed to the server by the DDEREAD or DDEEXECUTE statements.
- DDEAPP() returns the name of the application.
- DDETOPIC() returns the name of the topic.

When a Clarion program creates a DDE server, external clients can link to this server and request data. Each data request is accompanied by a string (in some specific format which the server program knows) indicating the required data item. If the Clarion server already knows the value for a given item, it supplies it to the client automatically without generating any events. If it doesn't, an EVENT:DDErequest or EVENT:DDEadvise event is posted to the server window's ACCEPT loop.

When a Clarion program creates a DDE client, it can link to external servers which can provide data. When the server first provides the value for a given item, it supplies it to the client automatically without generating any events. If the client has established a "hot" link with the server, an EVENT:DDEdata event is posted to the client window's ACCEPT loop whenever the server posts a new value for the data item.

DDE Functions

DDESERVER (return DDE server channel)

DDESERVER([*application*] [, *topic*] **)**

DDESERVER Returns a new DDE server channel number.

application A string constant or variable containing the name of the application. Usually, this is the name of the application. If omitted, the filename of the application (without extension) is used.

topic A string constant or variable containing the name of the application-specific topic. If omitted, the *application* will respond to any data request.

The **DDESERVER** function returns a new DDE server channel number for the *application* and *topic*. The channel number specifies a *topic* for which the *application* will provide data. This allows a single Clarion *application* to register as a DDE server for multiple *topics*.

Return Data Type: LONG

Example:

```
DDERetVal   STRING(20)
WinOne      WINDOW,AT(0,0,160,400)
              ENTRY(@s20),USE(DDERetVal)
            END
MyServer    LONG
  CODE
  OPEN(WinOne)
  MyServer = DDESERVER('MyApp','DataEntered')  !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest                    !As server for data requested once
      DDEWRITE(MyServer,DDE:manual,'DataEntered',DDERetVal)!Provide data once
    OF EVENT:DDEadvise                     !As server for constant update request
      DDEWRITE(MyServer,15,'DataEntered',DDERetVal)
                                           !Check for change every 15 seconds
                                           ! and provide data whenever changed
    END
  END
```

See Also: DDECLIENT, DDEWRITE

DDECLIENT (return DDE client channel)

DDECLIENT([*application*] [, *topic*] **)**

DDECLIENT Returns a new DDE client channel number.

application A string constant or variable containing the name of the server application to link to. Usually, this is the name of the application. If omitted, the first DDE server application available is used.

topic A string constant or variable containing the name of the application-specific topic. If omitted, the first topic available in the *application* is used.

The **DDECLIENT** function returns a new DDE client channel number for the *application* and *topic*. If the *application* is not currently executing, DDECLIENT returns zero (0).

Typically, when opening a DDE channel as the client, the *application* is the name of the server application. The *topic* is a string that the *application* has either registered with Windows as a valid *topic* for the *application*, or represents some value that tells the *application* what data to provide. You can use the DDEQUERY function to determine the *applications* and *topics* currently registered with Windows.

Return Data Type: LONG

Example:

```
DDEReadVal  REAL
WinOne      WINDOW,AT(0,0,160,400)
              ENTRY(@s20),USE(DDEReadVal)
            END
ExcelServer LONG
  CODE
  OPEN(WinOne)
  ExcelServer = DDECLIENT('Excel','MySheet.XLS')
                                          !Open as client to Excel spreadsheet
  IF NOT ExcelServer                      !If the server is not running
    MESSAGE('Please start Excel')         !alert the user to start it
    RETURN                                ! and try again
  END
  DDEREAD(ExcelServer,DDE:auto,'R5C5',DDEReadVal)
  ACCEPT
    CASE EVENT()
    OF EVENT:DDEdata                        !As changed data comes from Excel
      PassedData(DDEReadVal)                ! process the new data
    END
  END
```

See Also: DDEQUERY, DDEWRITE, DDESERVER

DDEQUERY (return registered DDE servers)

DDEQUERY([*application*] [, *topic*] **)**

DDEQUERY	Returns currently executing DDE servers.
application	A string constant or variable containing the name of the application to query. For most applications, this is the name of the application. If omitted, all registered *applications* registered with the specified *topic* are returned.
topic	A string constant or variable containing the name of the application-specific topic to query. If omitted, all *topics* for the *application* are returned.

The **DDEQUERY** function returns a string containing the names of the currently available DDE server *applications* and their *topics*.

If the *topic* parameter is omitted, all *topics* for the specified *application* are returned. If the *application* parameter is omitted, all registered *applications* registered with the specified *topic* are returned. If both parameters are omitted, DDEQUERY returns all currently available DDE servers.

The format of the data in the return string is *application:topic* and it can contain multiple *application* and *topic* pairs delimited by commas (for example, 'Excel:MySheet.XLS,ClarionApp:DataFile.DAT').

Return Data Type: STRING

Example:

```
  !This example code does not handle DDEADVISE
WinOne  WINDOW,AT(0,0,160,400)
        END
SomeServer    LONG
ServerString  STRING(200)
  CODE
  OPEN(WinOne)
  LOOP
    ServerString = DDEQUERY()                    !Return all registered servers
    IF NOT INSTRING('SomeApp:MyTopic',ServerString,1,1)
      MESSAGE('Open SomeApp, Please')
    ELSE
      BREAK
    END
  END
  SomeServer = DDECLIENT('SomeApp','MyTopic')      !Open as client
  ACCEPT
  END
  DDECLOSE(SomeServer)
```

DDECHANNEL (return DDE channel number)

DDECHANNEL( )

The **DDECHANNEL** function returns a LONG integer containing the channel number of the DDE client or server application that has just posted a DDE event. This is the same value returned by the DDESERVER or DDECLIENT function when the DDE channel is established.

Return Data Type: LONG

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
        END
TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)
  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER('SomeApp','Time')      !Open as server
  DateServer = DDESERVER('SomeApp','Date')      !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDECHANNEL()                         !Check which channel
      OF TimeServer
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE:manual,'Time',FormatTime)
      OF DateServer
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE:manual,'Date',FormatDate)
      END
    END
  END
```

DDEAPP (return server application)

DDEAPP( )

The **DDEAPP** function returns a string containing the application name in the DDE channel that has just posted a DDE event. This is usually the same as the first parameter to the DDESERVER or DDECLIENT function when the DDE channel is established.

Return Data Type: STRING

Example:

```
ClientApp STRING(20)
WinOne    WINDOW,AT(0,0,160,400)
            STRING(@S20),AT(5,5,90,20),USE(ClientApp)
          END

TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)

  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER('SomeApp','Time')      !Open as server
  DateServer = DDESERVER('SomeApp','Date')      !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDECHANNEL()
      OF TimeServer
        ClientApp = DDEAPP()                    !Get client's name
        DISPLAY                                 ! and display on screen
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE:manual,'Time',FormatTime)
      OF DateServer
        ClientApp = DDEAPP() !Get client's name
        DISPLAY                                 ! and display on screen
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE:manual,'Date',FormatDate)
      END
    END
  END
```

DDEITEM (return server item)

DDEITEM( )

The **DDEITEM** function returns a string containing the name of the item for the current DDE event. This is the item requested by a DDEREAD, the data item supplied by DDEPOKE, or the command to execute from a DDEEXECUTE statement.

Return Data Type: STRING

Example:

```
WinOne    WINDOW,AT(0,0,160,400)
          END

Server      LONG
FormatTime  STRING(5)
FormatDate  STRING(8)

  CODE
  OPEN(WinOne)
  Server = DDESERVER('SomeApp','Clock')  !Open as server for my topic
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDEITEM()
      OF 'Time'
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(Server,DDE:manual,'Time',FormatTime)
      OF 'Date'
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(Server,DDE:manual,'Date',FormatDate)
      END
    OF EVENT:DDEadvise
      CASE DDEITEM()
      OF 'Time'
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(Server,1,'Time',FormatTime)
      OF 'Date'
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(Server,60,'Date',FormatDate)
      END
    END
  END
```

See Also: DDEREAD, DDEEXECUTE

DDETOPIC (return server topic)

DDETOPIC( )

The **DDETOPIC** function returns a string containing the topic name for the DDE channel that has just posted a DDE event.

Return Data Type: STRING

Example:

```
WinOne  WINDOW,AT(0,0,160,400)
        END

TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)

  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER('SomeApp')       !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDETOPIC()                     !Get requested topic
      OF 'Time'
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE:manual,'Time',FormatTime)
      OF 'Date'
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE:manual,'Date',FormatDate)
      END
    END
  END
```

See Also: DDEREAD

DDEVALUE (return data value sent to server)

DDEVALUE( )

The **DDEVALUE** function returns a string containing the data sent to a Clarion DDE server by the DDEPOKE statement.

Return Data Type: STRING

Example:

```
WinOne      WINDOW,AT(0,0,160,400)
            END
TimeServer  LONG

TimeStamp   FILE,DRIVER(ASCII),PRE(Tim)
Record        RECORD
FormatTime      STRING(5)
FormatDate      STRING(8)
Message         STRING(50)
            . .

  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER('TimeStamp')            !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDEpoke
      OPEN(TimeStamp)
      Tim:FormatTime = FORMAT(CLOCK(),@T1)
      Tim:FormatDate = FORMAT(TODAY(),@D1)
      Tim:Message   = DDEVALUE()                 !Get data
      ADD(TimeStamp)
      CLOSE(TimeStamp)
      CYCLE                                      !Ensure acknowledgement
    END
  END
```

See Also: DDEPOKE

DDE Procedures

DDEREAD (get data from DDE server)

DDEREAD(*channel, mode, item* [, *variable*] **)**

DDEREAD	Gets data from a previously opened DDE client channel.
channel	A LONG integer constant or variable containing the client channel—the value returned by the DDECLIENT function.
mode	An EQUATE defining the type of data link: DDE:auto, DDE:manual, or DDE:remove (defined in EQUATES.CLW).
item	A string constant or variable containing the application-specific name of the data item to retrieve.
variable	The name of the variable to receive the retrieved data. If omitted and *mode* is DDE:remove, all links to the *item* are canceled.

The **DDEREAD** procedure allows a DDE client program to read data from the *channel* into the *variable*. The type of update is determined by the *mode* parameter. The *item* parameter supplies some string value to the server application that tells it what specific data item is being requested. The format and structure of the *item* string is dependent upon the server application.

If the *mode* is DDE:auto, the *variable* is continually updated by the server (a "hot" link). If the *mode* is DDE:manual, the *variable* is updated once and another DDEREAD request must be sent to the server to check for any changed value (a "cold" link). If the *mode* is DDE:remove, a previous "hot" link to the *variable* is terminated. If the *mode* is DDE:remove and *variable* is omitted, all previous "hot" links to the *item* are terminated, no matter what *variables* were linked. This means the client must send another DDEREAD request to the server to check for any changed value.

Errors Posted:

601 Invalid DDE Channel
602 DDE Channel Not Open
605 Time Out

Events Generated:

These events are posted to the client application:

EVENT:DDEdata	A server has supplied an updated data item for a hot link.
EVENT:DDEclose	A server has terminated the DDE link.

Example:

```
WinOne    WINDOW,AT(0,0,160,400)
          END

ExcelServer LONG(0)
DDEReadVal  REAL

  CODE
  OPEN(WinOne)
  ExcelServer = DDECLIENT('Excel','MySheet.XLS')
                                    !Open as client to Excel spreadsheet
  IF NOT ExcelServer                !If the server is not running
    MESSAGE('Please start Excel')   ! alert the user to start it
    CLOSE(WinOne)
    RETURN
  END
  DDEREAD(ExcelServer,DDE:auto,'R5C5',DDEReadVal)
                                   !Request continual update from server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDEdata                !As changed data comes from Excel
      PassedData(DDEReadVal)        ! call proc to process the new data
    END
  END
```

See Also: DDEQUERY, DDEWRITE

DDEWRITE (provide data to DDE client)

DDEWRITE(*channel, mode, item* [, *variable*] **)**

DDEWRITE	Provide data to an open DDE server channel.
channel	A LONG integer constant or variable containing the server channel—the value returned by the DDESERVER function.
mode	An integer constant or variable containing the interval (in seconds) to poll for changes to the *variable*, or an EQUATE defining the type of data link: DDE:auto, DDE:manual, or DDE:remove (defined in EQUATES.CLW).
item	A string constant or variable containing the application-specific name of the data item to provide.
variable	The name of the variable providing the data. If omitted and *mode* is DDE:remove, all links to the *item* are canceled.

The **DDEWRITE** procedure allows a DDE server program to provide the *variable's* data to the client. The *item* parameter supplies a string value that identifies the specific data item being provided. The format and structure of the *item* string is dependent upon the *server* application. The type of update performed is determined by the *mode* parameter.

If the *mode* is DDE:auto, the client program receives the current value of the variable and the internal libraries continue to provide that value whenever the client (or any other client) asks for it again. If the client requested a "hot" link, any changes to the *variable* should be tracked by the Clarion program so it can issue a new DDEWRITE statement to update the client with the new value.

If the *mode* is DDE:manual, the *variable* is updated only once. If the client requested a "hot" link, any changes to the *variable* should be tracked by the Clarion program so it can issue a new DDEWRITE statement to update the client with the new value. PROP:DDETimeOut can be used to set or get the time out value for the DDE connection (default is five seconds).

If the *mode* is a positive integer, the internal libraries check the value of the *variable* whenever the specified number of seconds has passed. If the value has changed, the client is automatically updated with the new value by the internal libraries (without the need for any further Clarion code). This can incur significant overhead, depending upon the data, and so should be used only when necessary.

If the *mode* is DDE:remove, any previous "hot" link to the *variable* is terminated. If the *mode* is DDE:remove and *variable* is omitted, all previous "hot" links to the *item* are terminated, no matter what *variables* were linked.

This means the client must send another DDEREAD request to the server to check for any changed value.

Errors Posted:

601 Invalid DDE Channel
602 DDE Channel Not Open
605 Time Out

Events Generated:

EVENT:DDErequest
A client has requested a data item (a “cold” link).

EVENT:DDEadvise
A client has requested continuous updates of a data item (a “hot” link).

Example:

```
DDERetVal   STRING(20)
WinOne      WINDOW,AT(0,0,160,400)
              ENTRY(@s20),USE(DDERetVal)
            END
MyServer    LONG
  CODE
  OPEN(WinOne)
  MyServer = DDESERVER('MyApp','DataEntered')    !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest                     !As server for data requested once
      DDEWRITE(MyServer,DDE:manual,'DataEntered',DDERetVal)
                                            !Provide data once
    OF EVENT:DDEadvise                      !As server for constant update request
      DDEWRITE(MyServer,15,'DataEntered',DDERetVal)
                                            !Check for change every 15 seconds
                                            ! and provide data whenever changed
    END
  END
```

See Also: DDEQUERY, DDEREAD

DDEEXECUTE (send command to DDE server)

DDEEXECUTE(*channel, command* **)**

DDEEXECUTE Sends a command string to an open DDE client channel.

channel A LONG integer constant or variable containing the client channel—the value returned by the DDECLIENT function.

command A string constant or variable containing the application-specific command for the server to execute.

The **DDEEXECUTE** procedure allows a DDE client program to communicate a *command* to the server. The *command* must be in a format the server application can recognize and act on. The server does not need to be a Clarion program. By convention, the entire *command* string is normally contained within square brackets ([ ]).

A Clarion DDE server can use the DDEITEM() function to determine what *command* the client has sent.The CYCLE statement after an EVENT:DDEexecute signals positive acknowledgement to the client that sent the *command*.

Errors Posted:

601 Invalid DDE Channel
602 DDE Channel Not Open
603 DDEEXECUTE Failed
605 Time Out

Events Generated:

EVENT:DDEexecute
A client has sent a command.

Example:

```
      !The client application's code contains:
WinOne    WINDOW,AT(0,0,160,400)
          END
SomeServer LONG
DDEChannel LONG
  CODE
  OPEN(WinOne)
  DDEChannel = DDECLIENT('PROGMAN','PROGMAN')
                                      !Open a channel to Windows Program Manager
  DDEEXECUTE(DDEChannel,'[CreateGroup(Clarion Applications)]')
                                      !Create a new program group
  DDEEXECUTE(DDEChannel,'[ShowGroup(1)]')   !Display it
  DDEEXECUTE(DDEChannel,'[AddItem(MYAPP.EXE,My Program,PROGMAN.EXE,2)]')
                                      !Create new item in the group
                                      ! using second icon in progman.exe
```

DDEPOKE (send unsolicited data to DDE server)

DDEPOKE(*channel, item, value* **)**

DDEPOKE	Sends unsolicited data through an open DDE client channel to a DDE server.
channel	A LONG integer constant or variable containing the client channel—the value returned by the DDECLIENT function.
item	A string constant or variable containing the application-specific item to receive the unsolicited data.
value	A string constant or variable containing the data to place in the *item*.

The **DDEPOKE** procedure allows a DDE client program to communicate unsolicited data to the server. The *item* and *value* parameters must be in a format the server application can recognize and act on. The server does not need to be a Clarion program.

A Clarion DDE server can use the DDEITEM() and DDEVALUE() functions to determine what the client has sent. The CYCLE statement after an EVENT:DDEpoke signals positive acknowledgement to the client that sent the unsolicited data.

Errors Posted:

601 Invalid DDE Channel
602 DDE Channel Not Open
604 DDEPOKE Failed
605 Time Out

Events Generated:

EVENT:DDEpoke	A client has sent unsolicited data

Example:

```
WinOne      WINDOW,AT(0,0,160,400)
            END
DDEChannel  LONG
  CODE
  OPEN(WinOne)
  DDEChannel = DDECLIENT('Excel','System')              !Open channel to Excel
  DDEEXECUTE(DDEChannel,'[NEW(1)]')                     !Create a new spreadsheet
  DDEEXECUTE(DDEChannel,'[Save.As("DDE_CHART.XLS")]')   !Save it as DDE_CHART.XLS
  DDECLOSE(DDEChannel)                                  !Close conversation
  DDEChannel = DDECLIENT('Excel','DDE_CHART.XLA')       !Open channel to new chart
  DDEPOKE(DDEChannel,'R1C2','Widgets')                  !Send it data
  DDEPOKE(DDEChannel,'R1C3','Gadgets')
  DDEPOKE(DDEChannel,'R2C1','East')
  DDEPOKE(DDEChannel,'R3C1','West')
  DDEPOKE(DDEChannel,'R2C2','450')
  DDEPOKE(DDEChannel,'R3C2','275')
  DDEPOKE(DDEChannel,'R2C3','340')
  DDEPOKE(DDEChannel,'R3C3','390')
  DDEEXECUTE(DDEChannel,'[SELECT("R1C1:R3C2")]')        !Highlight the data
  DDEEXECUTE(DDEChannel,'[NEW(2,2)]')                   ! and create a new chart
     !Send some more commands to format the chart and work with it
  DDECLOSE(DDEChannel)                                  !Close channel when done
```

DDECLOSE (terminate DDE server link)

DDECLOSE(*channel* **)**

DDECLOSE Closes an open DDE channel.

channel The label of the LONG integer variable containing the channel number—the value returned by the DDESERVER or DDECLIENT function.

The **DDECLOSE** procedure allows a DDE client program to terminate the specified *channel*. A *channel* is automatically terminated when the window which opened the *channel* is closed.

Errors Posted: 601 Invalid DDE Channel
602 DDE Channel Not Open
605 Time Out

Example:

```
WinOne      WINDOW,AT(0,0,160,400)
            END
SomeServer  LONG
  CODE
  OPEN(WinOne)
  SomeServer = DDECLIENT('SomeApp','MyTopic') !Open as client
  ACCEPT
  END
  DDECLOSE(SomeServer)
```

Clarion Keycodes

Contents

Windows Keycode Mapping Format

Each key on the keyboard is assigned a keycode. Keycodes are 16-bit values where the low-order 8 bits (values from 0 to 255) represent the key that was pressed, and the high-order 8 bits indicate the state of the Shift, Ctrl, and Alt keys. Keycodes are returned by the KEYCODE() and KEYBOARD() functions, and use the following format:

```
          | A | C | S |   CODE  |

 Bits:  10   9   8   7         0

CODE    - The Key pressed
A       - Alt key bit
C       - Ctrl key bit
S       - Shift key bit
```

Calculating a keycode's numeric value is generally unnecessary, since most of the possible key combinations are listed as EQUATES in KEYCODES.CLW (INCLUDE this file and use the equates instead of the numbers). The contents of KEYCODES.CLW are listed in Appendix A.

KEYCODES.CLW

Keycode equate labels assign mnemonic labels to Clarion keycodes. The keycode equates file (KEYCODES.CLW) is a Clarion source file which contains an EQUATE statement for each keycode. This file is located in the directory in which you installed Clarion Database Developer. It may be merged into a source PROGRAM with the statement: INCLUDE('KEYCODES.CLW').

This file contains EQUATE statements for all the keycodes:

```
Key0            EQUATE(0030H)               !0 Key
Key1            EQUATE(0031H)               !1 Key
Key2            EQUATE(0032H)               !2 Key
Key3            EQUATE(0033H)               !3 Key
Key4            EQUATE(0034H)               !4 Key
Key5            EQUATE(0035H)               !5 Key
Key6            EQUATE(0036H)               !6 Key
Key7            EQUATE(0037H)               !7 Key
Key8            EQUATE(0038H)               !8 Key
Key9            EQUATE(0039H)               !9 Key
AKey            EQUATE(0041H)               !A Key
BKey            EQUATE(0042H)               !B Key
CKey            EQUATE(0043H)               !C Key
DKey            EQUATE(0044H)               !D Key
EKey            EQUATE(0045H)               !E Key
FKey            EQUATE(0046H)               !F Key
GKey            EQUATE(0047H)               !G Key
HKey            EQUATE(0048H)               !H Key
IKey            EQUATE(0049H)               !I Key
JKey            EQUATE(004AH)               !J Key
KKey            EQUATE(004BH)               !K Key
LKey            EQUATE(004CH)               !L Key
MKey            EQUATE(004DH)               !M Key
NKey            EQUATE(004EH)               !N Key
OKey            EQUATE(004FH)               !O Key
PKey            EQUATE(0050H)               !P Key
QKey            EQUATE(0051H)               !Q Key
RKey            EQUATE(0052H)               !R Key
SKey            EQUATE(0053H)               !S Key
TKey            EQUATE(0054H)               !T Key
UKey            EQUATE(0055H)               !U Key
VKey            EQUATE(0056H)               !V Key
WKey            EQUATE(0057H)               !W Key
XKey            EQUATE(0058H)               !X Key
YKey            EQUATE(0059H)               !Y Key
ZKey            EQUATE(005AH)               !Z Key
F1Key           EQUATE(0070H)               !F1 Key
F2Key           EQUATE(0071H)               !F2 Key
F3Key           EQUATE(0072H)               !F3 Key
F4Key           EQUATE(0073H)               !F4 Key
F5Key           EQUATE(0074H)               !F5 Key
F6Key           EQUATE(0075H)               !F6 Key
F7Key           EQUATE(0076H)               !F7 Key
F8Key           EQUATE(0077H)               !F8 Key
F9Key           EQUATE(0078H)               !F9 Key
F10Key          EQUATE(0079H)               !F10 Key
F11Key          EQUATE(007AH)               !F11 Key
F12Key          EQUATE(007BH)               !F12 Key
```

```
AstKey          EQUATE(006AH)       !Asterisk Key
BSKey           EQUATE(0008H)       !Backspace Key
CapsLockKey     EQUATE(0014H)       !CapsLock Key
DecimalKey      EQUATE(006EH)       !Decimal Key
DeleteKey       EQUATE(002EH)       !Delete Key
DivideKey       EQUATE(006FH)       !Divide Key
DownKey         EQUATE(0028H)       !Cursor Down Key
EndKey          EQUATE(0023H)       !End Key
EnterKey        EQUATE(000DH)       !Enter Key
EscKey          EQUATE(001BH)       !Esc Key
HomeKey         EQUATE(0024H)       !Home Key
InsertKey       EQUATE(002DH)       !Insert Key
LeftKey         EQUATE(0025H)       !Cursor Left Key
MinusKey        EQUATE(006DH)       !Minus Key
PauseKey        EQUATE(0013H)       !Pause Key
PgDnKey         EQUATE(0022H)       !PgDn Key
PgUpKey         EQUATE(0021H)       !PgUp Key
PlusKey         EQUATE(006BH)       !Plus Key
PrintKey        EQUATE(002CH)       !PrintScreen Key
RightKey        EQUATE(0027H)       !Cursor Right Key
SlashKey        EQUATE(006FH)       !Slash Key
SpaceKey        EQUATE(0020H)       !Spacebar
TabKey          EQUATE(0009H)       !Tab Key
UpKey           EQUATE(0026H)       !Cursor Up Key
KeyPad0         EQUATE(0060H)       !0 on numeric keypad
KeyPad1         EQUATE(0061H)       !1 on numeric keypad
KeyPad2         EQUATE(0062H)       !2 on numeric keypad
KeyPad3         EQUATE(0063H)       !3 on numeric keypad
KeyPad4         EQUATE(0064H)       !4 on numeric keypad
KeyPad5         EQUATE(0065H)       !5 on numeric keypad
KeyPad6         EQUATE(0066H)       !6 on numeric keypad
KeyPad7         EQUATE(0067H)       !7 on numeric keypad
KeyPad8         EQUATE(0068H)       !8 on numeric keypad
KeyPad9         EQUATE(0069H)       !9 on numeric keypad
MouseLeft       EQUATE(0001H)       !Left mouse button
MouseRight      EQUATE(0002H)       !Right mouse button
MouseCenter     EQUATE(0004H)       !Middle mouse button
Alt0            EQUATE(0430H)       !Alt-0 Key
Alt1            EQUATE(0431H)       !Alt-1 Key
Alt2            EQUATE(0432H)       !Alt-2 Key
Alt3            EQUATE(0433H)       !Alt-3 Key
Alt4            EQUATE(0434H)       !Alt-4 Key
Alt5            EQUATE(0435H)       !Alt-5 Key
Alt6            EQUATE(0436H)       !Alt-6 Key
Alt7            EQUATE(0437H)       !Alt-7 Key
Alt8            EQUATE(0438H)       !Alt-8 Key
Alt9            EQUATE(0439H)       !Alt-9 Key
AltA            EQUATE(0441H)       !Alt-A Key
AltB            EQUATE(0442H)       !Alt-B Key
AltC            EQUATE(0443H)       !Alt-C Key
AltD            EQUATE(0444H)       !Alt-D Key
AltE            EQUATE(0445H)       !Alt-E Key
AltF            EQUATE(0446H)       !Alt-F Key
AltG            EQUATE(0447H)       !Alt-G Key
AltH            EQUATE(0448H)       !Alt-H Key
AltI            EQUATE(0449H)       !Alt-I Key
AltJ            EQUATE(044AH)       !Alt-J Key
AltK            EQUATE(044BH)       !Alt-K Key
AltL            EQUATE(044CH)       !Alt-L Key
AltM            EQUATE(044DH)       !Alt-M Key
AltN            EQUATE(044EH)       !Alt-N Key
AltO            EQUATE(044FH)       !Alt-O Key
```

```
AltP              EQUATE(0450H)             !Alt-P Key
AltQ              EQUATE(0451H)             !Alt-Q Key
AltR              EQUATE(0452H)             !Alt-R Key
AltS              EQUATE(0453H)             !Alt-S Key
AltT              EQUATE(0454H)             !Alt-T Key
AltU              EQUATE(0455H)             !Alt-U Key
AltV              EQUATE(0456H)             !Alt-V Key
AltW              EQUATE(0457H)             !Alt-W Key
AltX              EQUATE(0458H)             !Alt-X Key
AltY              EQUATE(0459H)             !Alt-Y Key
AltZ              EQUATE(045AH)             !Alt-Z Key
AltF1             EQUATE(0470H)             !Alt-F1 Key
AltF2             EQUATE(0471H)             !Alt-F2 Key
AltF3             EQUATE(0472H)             !Alt-F3 Key
AltF4             EQUATE(0473H)             !Alt-F4 Key
AltF5             EQUATE(0474H)             !Alt-F5 Key
AltF6             EQUATE(0475H)             !Alt-F6 Key
AltF7             EQUATE(0476H)             !Alt-F7 Key
AltF8             EQUATE(0477H)             !Alt-F8 Key
AltF9             EQUATE(0478H)             !Alt-F9 Key
AltF10            EQUATE(0479H)             !Alt-F10 Key
AltF11            EQUATE(047AH)             !Alt-F11 Key
AltF12            EQUATE(047BH)             !Alt-F12 Key
AltAst            EQUATE(046AH)             !Alt-Asterisk Key
AltBS             EQUATE(0408H)             !Alt-Backspace Key
AltDecimal        EQUATE(046EH)             !Alt-Decimal Key
AltDelete         EQUATE(042EH)             !Alt-Delete Key
AltDivide         EQUATE(046FH)             !Alt-Divide Key
AltDown           EQUATE(0428H)             !Alt-Cursor Down Key
AltEnd            EQUATE(0423H)             !Alt-End Key
AltEnter          EQUATE(040DH)             !Alt-Enter Key
AltEsc            EQUATE(041BH)             !Alt-Esc Key
AltHome           EQUATE(0424H)             !Alt-Home Key
AltInsert         EQUATE(042DH)             !Alt-Insert Key
AltLeft           EQUATE(0425H)             !Alt-Cursor Left Key
AltMinus          EQUATE(046DH)             !Alt-Minus Key
AltPause          EQUATE(0413H)             !Alt-Pause Key
AltPgDn           EQUATE(0422H)             !Alt-PgDn Key
AltPgUp           EQUATE(0421H)             !Alt-PgUp Key
AltPlus           EQUATE(046BH)             !Alt-Plus Key
AltPrint          EQUATE(042CH)             !Alt-PrintScreen Key
AltRight          EQUATE(0427H)             !Alt-Cursor Right Key
AltSlash          EQUATE(046FH)             !Alt-Slash Key
AltSpace          EQUATE(0420H)             !Alt-Spacebar
AltTab            EQUATE(0409H)             !Alt-Tab Key
AltUp             EQUATE(0426H)             !Alt-Cursor Up Key
AltPad0           EQUATE(0460H)           !Alt-0 on numeric keypad
AltPad1           EQUATE(0461H)           !Alt-1 on numeric keypad
AltPad2           EQUATE(0462H)           !Alt-2 on numeric keypad
AltPad3           EQUATE(0463H)           !Alt-3 on numeric keypad
AltPad4           EQUATE(0464H)           !Alt-4 on numeric keypad
AltPad5           EQUATE(0465H)           !Alt-5 on numeric keypad
AltPad6           EQUATE(0466H)           !Alt-6 on numeric keypad
AltPad7           EQUATE(0467H)           !Alt-7 on numeric keypad
AltPad8           EQUATE(0468H)           !Alt-8 on numeric keypad
AltPad9           EQUATE(0469H)           !Alt-9 on numeric keypad
AltMouseLeft      EQUATE(0401H)           !Alt-Left mouse button
AltMouseRight     EQUATE(0402H)           !Alt-Right mouse button
AltMouseCenter    EQUATE(0404H)           !Alt-Middle mouse button
Ctrl0             EQUATE(0230H)             !Ctrl-0 Key
Ctrl1             EQUATE(0231H)             !Ctrl-1 Key
Ctrl2             EQUATE(0232H)             !Ctrl-2 Key
```

```
Ctrl3           EQUATE(0233H)           !Ctrl-3 Key
Ctrl4           EQUATE(0234H)           !Ctrl-4 Key
Ctrl5           EQUATE(0235H)           !Ctrl-5 Key
Ctrl6           EQUATE(0236H)           !Ctrl-6 Key
Ctrl7           EQUATE(0237H)           !Ctrl-7 Key
Ctrl8           EQUATE(0238H)           !Ctrl-8 Key
Ctrl9           EQUATE(0239H)           !Ctrl-9 Key
CtrlA           EQUATE(0241H)           !Ctrl-A Key
CtrlB           EQUATE(0242H)           !Ctrl-B Key
CtrlC           EQUATE(0243H)           !Ctrl-C Key
CtrlD           EQUATE(0244H)           !Ctrl-D Key
CtrlE           EQUATE(0245H)           !Ctrl-E Key
CtrlF           EQUATE(0246H)           !Ctrl-F Key
CtrlG           EQUATE(0247H)           !Ctrl-G Key
CtrlH           EQUATE(0248H)           !Ctrl-H Key
CtrlI           EQUATE(0249H)           !Ctrl-I Key
CtrlJ           EQUATE(024AH)           !Ctrl-J Key
CtrlK           EQUATE(024BH)           !Ctrl-K Key
CtrlL           EQUATE(024CH)           !Ctrl-L Key
CtrlM           EQUATE(024DH)           !Ctrl-M Key
CtrlN           EQUATE(024EH)           !Ctrl-N Key
CtrlO           EQUATE(024FH)           !Ctrl-O Key
CtrlP           EQUATE(0250H)           !Ctrl-P Key
CtrlQ           EQUATE(0251H)           !Ctrl-Q Key
CtrlR           EQUATE(0252H)           !Ctrl-R Key
CtrlS           EQUATE(0253H)           !Ctrl-S Key
CtrlT           EQUATE(0254H)           !Ctrl-T Key
CtrlU           EQUATE(0255H)           !Ctrl-U Key
CtrlV           EQUATE(0256H)           !Ctrl-V Key
CtrlW           EQUATE(0257H)           !Ctrl-W Key
CtrlX           EQUATE(0258H)           !Ctrl-X Key
CtrlY           EQUATE(0259H)           !Ctrl-Y Key
CtrlZ           EQUATE(025AH)           !Ctrl-Z Key
CtrlF1          EQUATE(0270H)           !Ctrl-F1 Key
CtrlF2          EQUATE(0271H)           !Ctrl-F2 Key
CtrlF3          EQUATE(0272H)           !Ctrl-F3 Key
CtrlF4          EQUATE(0273H)           !Ctrl-F4 Key
CtrlF5          EQUATE(0274H)           !Ctrl-F5 Key
CtrlF6          EQUATE(0275H)           !Ctrl-F6 Key
CtrlF7          EQUATE(0276H)           !Ctrl-F7 Key
CtrlF8          EQUATE(0277H)           !Ctrl-F8 Key
CtrlF9          EQUATE(0278H)           !Ctrl-F9 Key
CtrlF10         EQUATE(0279H)           !Ctrl-F10 Key
CtrlF11         EQUATE(027AH)           !Ctrl-F11 Key
CtrlF12         EQUATE(027BH)           !Ctrl-F12 Key
CtrlAst         EQUATE(026AH)           !Ctrl-Asterisk Key
CtrlBS          EQUATE(0208H)           !Ctrl-Backspace Key
CtrlDecimal     EQUATE(026EH)           !Ctrl-Decimal Key
CtrlDelete      EQUATE(022EH)           !Ctrl-Delete Key
CtrlDivide      EQUATE(026FH)           !Ctrl-Divide Key
CtrlDown        EQUATE(0228H)           !Ctrl-Cursor Down Key
CtrlEnd         EQUATE(0223H)           !Ctrl-End Key
CtrlEnter       EQUATE(020DH)           !Ctrl-Enter Key
CtrlEsc         EQUATE(021BH)           !Ctrl-Esc Key
CtrlHome        EQUATE(0224H)           !Ctrl-Home Key
CtrlInsert      EQUATE(022DH)           !Ctrl-Insert Key
CtrlLeft        EQUATE(0225H)           !Ctrl-Cursor Left Key
CtrlMinus       EQUATE(026DH)           !Ctrl-Minus Key
CtrlPause       EQUATE(0213H)           !Ctrl-Pause Key
CtrlPgDn        EQUATE(0222H)           !Ctrl-PgDn Key
CtrlPgUp        EQUATE(0221H)           !Ctrl-PgUp Key
CtrlPlus        EQUATE(026BH)           !Ctrl-Plus Key
```

```
CtrlPrint       EQUATE(022CH)               !Ctrl-PrintScreen Key
CtrlRight       EQUATE(0227H)               !Ctrl-Cursor Right Key
CtrlSlash       EQUATE(026FH)               !Ctrl-Slash Key
CtrlSpace       EQUATE(0220H)               !Ctrl-Spacebar
CtrlTab         EQUATE(0209H)               !Ctrl-Tab Key
CtrlUp          EQUATE(0226H)               !Ctrl-Cursor Up Key
CtrlPad0        EQUATE(0260H)              !Ctrl-0 on numeric keypad
CtrlPad1        EQUATE(0261H)              !Ctrl-1 on numeric keypad
CtrlPad2        EQUATE(0262H)              !Ctrl-2 on numeric keypad
CtrlPad3        EQUATE(0263H)              !Ctrl-3 on numeric keypad
CtrlPad4        EQUATE(0264H)              !Ctrl-4 on numeric keypad
CtrlPad5        EQUATE(0265H)              !Ctrl-5 on numeric keypad
CtrlPad6        EQUATE(0266H)              !Ctrl-6 on numeric keypad
CtrlPad7        EQUATE(0267H)              !Ctrl-7 on numeric keypad
CtrlPad8        EQUATE(0268H)              !Ctrl-8 on numeric keypad
CtrlPad9        EQUATE(0269H)              !Ctrl-9 on numeric keypad
CtrlMouseLeft   EQUATE(0201H)              !Ctrl-Left mouse button
CtrlMouseRight  EQUATE(0202H)              !Ctrl-Right mouse button
CtrlMouseCenter EQUATE(0204H)              !Ctrl-Middle mouse button
Shift0          EQUATE(0130H)               !Shift-0 Key
Shift1          EQUATE(0131H)               !Shift-1 Key
Shift2          EQUATE(0132H)               !Shift-2 Key
Shift3          EQUATE(0133H)               !Shift-3 Key
Shift4          EQUATE(0134H)               !Shift-4 Key
Shift5          EQUATE(0135H)               !Shift-5 Key
Shift6          EQUATE(0136H)               !Shift-6 Key
Shift7          EQUATE(0137H)               !Shift-7 Key
Shift8          EQUATE(0138H)               !Shift-8 Key
Shift9          EQUATE(0139H)               !Shift-9 Key
ShiftA          EQUATE(0141H)               !Shift-A Key
ShiftB          EQUATE(0142H)               !Shift-B Key
ShiftC          EQUATE(0143H)               !Shift-C Key
ShiftD          EQUATE(0144H)               !Shift-D Key
ShiftE          EQUATE(0145H)               !Shift-E Key
ShiftF          EQUATE(0146H)               !Shift-F Key
ShiftG          EQUATE(0147H)               !Shift-G Key
ShiftH          EQUATE(0148H)               !Shift-H Key
ShiftI          EQUATE(0149H)               !Shift-I Key
ShiftJ          EQUATE(014AH)               !Shift-J Key
ShiftK          EQUATE(014BH)               !Shift-K Key
ShiftL          EQUATE(014CH)               !Shift-L Key
ShiftM          EQUATE(014DH)               !Shift-M Key
ShiftN          EQUATE(014EH)               !Shift-N Key
ShiftO          EQUATE(014FH)               !Shift-O Key
ShiftP          EQUATE(0150H)               !Shift-P Key
ShiftQ          EQUATE(0151H)               !Shift-Q Key
ShiftR          EQUATE(0152H)               !Shift-R Key
ShiftS          EQUATE(0153H)               !Shift-S Key
ShiftT          EQUATE(0154H)               !Shift-T Key
ShiftU          EQUATE(0155H)               !Shift-U Key
ShiftV          EQUATE(0156H)               !Shift-V Key
ShiftW          EQUATE(0157H)               !Shift-W Key
ShiftX          EQUATE(0158H)               !Shift-X Key
ShiftY          EQUATE(0159H)               !Shift-Y Key
ShiftZ          EQUATE(015AH)               !Shift-Z Key
ShiftF1         EQUATE(0170H)               !Shift-F1 Key
ShiftF2         EQUATE(0171H)               !Shift-F2 Key
ShiftF3         EQUATE(0172H)               !Shift-F3 Key
ShiftF4         EQUATE(0173H)               !Shift-F4 Key
ShiftF5         EQUATE(0174H)               !Shift-F5 Key
ShiftF6         EQUATE(0175H)               !Shift-F6 Key
ShiftF7         EQUATE(0176H)               !Shift-F7 Key
```

```
ShiftF8         EQUATE(0177H)               !Shift-F8 Key
ShiftF9         EQUATE(0178H)               !Shift-F9 Key
ShiftF10        EQUATE(0179H)               !Shift-F10 Key
ShiftF11        EQUATE(017AH)               !Shift-F11 Key
ShiftF12        EQUATE(017BH)               !Shift-F12 Key
ShiftAst        EQUATE(016AH)               !Shift-Asterisk Key
ShiftBS         EQUATE(0108H)               !Shift-Backspace Key
ShiftDecimal    EQUATE(016EH)               !Shift-Decimal Key
ShiftDelete     EQUATE(012EH)               !Shift-Delete Key
ShiftDivide     EQUATE(016FH)               !Shift-Divide Key
ShiftDown       EQUATE(0128H)               !Shift-Cursor Down Key
ShiftEnd        EQUATE(0123H)               !Shift-End Key
ShiftEnter      EQUATE(010DH)               !Shift-Enter Key
ShiftEsc        EQUATE(011BH)               !Shift-Esc Key
ShiftHome       EQUATE(0124H)               !Shift-Home Key
ShiftInsert     EQUATE(012DH)               !Shift-Insert Key
ShiftLeft       EQUATE(0125H)               !Shift-Cursor Left Key
ShiftMinus      EQUATE(016DH)               !Shift-Minus Key
ShiftPause      EQUATE(0113H)               !Shift-Pause Key
ShiftPgDn       EQUATE(0122H)               !Shift-PgDn Key
ShiftPgUp       EQUATE(0121H)               !Shift-PgUp Key
ShiftPlus       EQUATE(016BH)               !Shift-Plus Key
ShiftPrint      EQUATE(012CH)               !Shift-PrintScreen Key
ShiftRight      EQUATE(0127H)               !Shift-Cursor Right Key
ShiftSlash      EQUATE(016FH)               !Shift-Slash Key
ShiftSpace      EQUATE(0120H)               !Shift-Spacebar
ShiftTab        EQUATE(0109H)               !Shift-Tab Key
ShiftUp         EQUATE(0126H)               !Shift-Cursor Up Key
ShiftPad0       EQUATE(0160H)             !Shift-0 on numeric keypad
ShiftPad1       EQUATE(0161H)             !Shift-1 on numeric keypad
ShiftPad2       EQUATE(0162H)             !Shift-2 on numeric keypad
ShiftPad3       EQUATE(0163H)             !Shift-3 on numeric keypad
ShiftPad4       EQUATE(0164H)             !Shift-4 on numeric keypad
ShiftPad5       EQUATE(0165H)             !Shift-5 on numeric keypad
ShiftPad6       EQUATE(0166H)             !Shift-6 on numeric keypad
ShiftPad7       EQUATE(0167H)             !Shift-7 on numeric keypad
ShiftPad8       EQUATE(0168H)             !Shift-8 on numeric keypad
ShiftPad9       EQUATE(0169H)             !Shift-9 on numeric keypad
ShiftMouseLeft   EQUATE(0101H)            !Shift-Left mouse button
ShiftMouseRight  EQUATE(0102H)            !Shift-Right mouse button
ShiftMouseCenter EQUATE(0104H)            !Shift-Middle mouse button
AltShift0       EQUATE(0530H)               !Alt-Shift-0 Key
AltShift1       EQUATE(0531H)               !Alt-Shift-1 Key
AltShift2       EQUATE(0532H)               !Alt-Shift-2 Key
AltShift3       EQUATE(0533H)               !Alt-Shift-3 Key
AltShift4       EQUATE(0534H)               !Alt-Shift-4 Key
AltShift5       EQUATE(0535H)               !Alt-Shift-5 Key
AltShift6       EQUATE(0536H)               !Alt-Shift-6 Key
AltShift7       EQUATE(0537H)               !Alt-Shift-7 Key
AltShift8       EQUATE(0538H)               !Alt-Shift-8 Key
AltShift9       EQUATE(0539H)               !Alt-Shift-9 Key
AltShiftA       EQUATE(0541H)               !Alt-Shift-A Key
AltShiftB       EQUATE(0542H)               !Alt-Shift-B Key
AltShiftC       EQUATE(0543H)               !Alt-Shift-C Key
AltShiftD       EQUATE(0544H)               !Alt-Shift-D Key
AltShiftE       EQUATE(0545H)               !Alt-Shift-E Key
AltShiftF       EQUATE(0546H)               !Alt-Shift-F Key
AltShiftG       EQUATE(0547H)               !Alt-Shift-G Key
AltShiftH       EQUATE(0548H)               !Alt-Shift-H Key
AltShiftI       EQUATE(0549H)               !Alt-Shift-I Key
AltShiftJ       EQUATE(054AH)               !Alt-Shift-J Key
AltShiftK       EQUATE(054BH)               !Alt-Shift-K Key
```

```
AltShiftL       EQUATE(054CH)                   !Alt-Shift-L Key
AltShiftM       EQUATE(054DH)                   !Alt-Shift-M Key
AltShiftN       EQUATE(054EH)                   !Alt-Shift-N Key
AltShiftO       EQUATE(054FH)                   !Alt-Shift-O Key
AltShiftP       EQUATE(0550H)                   !Alt-Shift-P Key
AltShiftQ       EQUATE(0551H)                   !Alt-Shift-Q Key
AltShiftR       EQUATE(0552H)                   !Alt-Shift-R Key
AltShiftS       EQUATE(0553H)                   !Alt-Shift-S Key
AltShiftT       EQUATE(0554H)                   !Alt-Shift-T Key
AltShiftU       EQUATE(0555H)                   !Alt-Shift-U Key
AltShiftV       EQUATE(0556H)                   !Alt-Shift-V Key
AltShiftW       EQUATE(0557H)                   !Alt-Shift-W Key
AltShiftX       EQUATE(0558H)                   !Alt-Shift-X Key
AltShiftY       EQUATE(0559H)                   !Alt-Shift-Y Key
AltShiftZ       EQUATE(055AH)                   !Alt-Shift-Z Key
AltShiftF1      EQUATE(0570H)                   !Alt-Shift-F1 Key
AltShiftF2      EQUATE(0571H)                   !Alt-Shift-F2 Key
AltShiftF3      EQUATE(0572H)                   !Alt-Shift-F3 Key
AltShiftF4      EQUATE(0573H)                   !Alt-Shift-F4 Key
AltShiftF5      EQUATE(0574H)                   !Alt-Shift-F5 Key
AltShiftF6      EQUATE(0575H)                   !Alt-Shift-F6 Key
AltShiftF7      EQUATE(0576H)                   !Alt-Shift-F7 Key
AltShiftF8      EQUATE(0577H)                   !Alt-Shift-F8 Key
AltShiftF9      EQUATE(0578H)                   !Alt-Shift-F9 Key
AltShiftF10     EQUATE(0579H)                   !Alt-Shift-F10 Key
AltShiftF11     EQUATE(057AH)                   !Alt-Shift-F11 Key
AltShiftF12     EQUATE(057BH)                   !Alt-Shift-F12 Key
AltShiftAst     EQUATE(056AH)                   !Alt-Shift-Asterisk Key
AltShiftBS      EQUATE(0508H)                   !Alt-Shift-Backspace
AltShiftDecimal EQUATE(056EH)                   !Alt-Shift-Decimal Key
AltShiftDelete  EQUATE(052EH)                   !Alt-Shift-Delete Key
AltShiftDivide  EQUATE(056FH)                   !Alt-Shift-Divide Key
AltShiftDown    EQUATE(0528H)                   !Alt-Shift-Cursor Down
AltShiftEnd     EQUATE(0523H)                   !Alt-Shift-End Key
AltShiftEnter   EQUATE(050DH)                   !Alt-Shift-Enter Key
AltShiftEsc     EQUATE(051BH)                   !Alt-Shift-Esc Key
AltShiftHome    EQUATE(0524H)                   !Alt-Shift-Home Key
AltShiftInsert  EQUATE(052DH)                   !Alt-Shift-Insert Key
AltShiftLeft    EQUATE(0525H)                   !Alt-Shift-Cursor Left
Key
AltShiftMinus   EQUATE(056DH)                   !Alt-Shift-Minus Key
AltShiftPause   EQUATE(0513H)                   !Alt-Shift-Pause Key
AltShiftPgDn    EQUATE(0522H)                   !Alt-Shift-PgDn Key
AltShiftPgUp    EQUATE(0521H)                   !Alt-Shift-PgUp Key
AltShiftPlus    EQUATE(056BH)                   !Alt-Shift-Plus Key
AltShiftPrint   EQUATE(052CH)                   !Alt-Shift-PrintScreen
AltShiftRight   EQUATE(0527H)                   !Alt-Shift-Cursor Right
AltShiftSlash   EQUATE(056FH)                   !Alt-Shift-Slash Key
AltShiftSpace   EQUATE(0520H)                   !Alt-Shift-Spacebar
AltShiftTab     EQUATE(0509H)                   !Alt-Shift-Tab Key
AltShiftUp      EQUATE(0526H)                   !Alt-Shift-Cursor Up
AltShiftPad0    EQUATE(0560H)           !Alt-Shift-0 on numeric keypad
AltShiftPad1    EQUATE(0561H)           !Alt-Shift-1 on numeric keypad
AltShiftPad2    EQUATE(0562H)          Alt-Shift-2 on numeric keypad
AltShiftPad3    EQUATE(0563H)           !Alt-Shift-3 on numeric keypad
AltShiftPad4    EQUATE(0564H)           !Alt-Shift-4 on numeric keypad
AltShiftPad5    EQUATE(0565H)           !Alt-Shift-5 on numeric keypad
AltShiftPad6    EQUATE(0566H)           !Alt-Shift-6 on numeric keypad
AltShiftPad7    EQUATE(0567H)           !Alt-Shift-7 on numeric keypad
AltShiftPad8    EQUATE(0568H)           !Alt-Shift-8 on numeric keypad
AltShiftPad9    EQUATE(0569H)           !Alt-Shift-9 on numeric keypad
AltShiftMouseLeft EQUATE(0501H)         !Alt-Shift-Left mouse button
```

```
AltShiftMouseRight  EQUATE(0502H)       !Alt-Shift-Right mouse button
AltShiftMouseCenter EQUATE(0504H)       !Alt-Shift-Middle mouse button
CtrlShift0      EQUATE(0330H)                   !Ctrl-Shift-0 Key
CtrlShift1      EQUATE(0331H)                   !Ctrl-Shift-1 Key
CtrlShift2      EQUATE(0332H)                   !Ctrl-Shift-2 Key
CtrlShift3      EQUATE(0333H)                   !Ctrl-Shift-3 Key
CtrlShift4      EQUATE(0334H)                   !Ctrl-Shift-4 Key
CtrlShift5      EQUATE(0335H)                   !Ctrl-Shift-5 Key
CtrlShift6      EQUATE(0336H)                   !Ctrl-Shift-6 Key
CtrlShift7      EQUATE(0337H)                   !Ctrl-Shift-7 Key
CtrlShift8      EQUATE(0338H)                   !Ctrl-Shift-8 Key
CtrlShift9      EQUATE(0339H)                   !Ctrl-Shift-9 Key
CtrlShiftA      EQUATE(0341H)                   !Ctrl-Shift-A Key
CtrlShiftB      EQUATE(0342H)                   !Ctrl-Shift-B Key
CtrlShiftC      EQUATE(0343H)                   !Ctrl-Shift-C Key
CtrlShiftD      EQUATE(0344H)                   !Ctrl-Shift-D Key
CtrlShiftE      EQUATE(0345H)                   !Ctrl-Shift-E Key
CtrlShiftF      EQUATE(0346H)                   !Ctrl-Shift-F Key
CtrlShiftG      EQUATE(0347H)                   !Ctrl-Shift-G Key
CtrlShiftH      EQUATE(0348H)                   !Ctrl-Shift-H Key
CtrlShiftI      EQUATE(0349H)                   !Ctrl-Shift-I Key
CtrlShiftJ      EQUATE(034AH)                   !Ctrl-Shift-J Key
CtrlShiftK      EQUATE(034BH)                   !Ctrl-Shift-K Key
CtrlShiftL      EQUATE(034CH)                   !Ctrl-Shift-L Key
CtrlShiftM      EQUATE(034DH)                   !Ctrl-Shift-M Key
CtrlShiftN      EQUATE(034EH)                   !Ctrl-Shift-N Key
CtrlShiftO      EQUATE(034FH)                   !Ctrl-Shift-O Key
CtrlShiftP      EQUATE(0350H)                   !Ctrl-Shift-P Key
CtrlShiftQ      EQUATE(0351H)                   !Ctrl-Shift-Q Key
CtrlShiftR      EQUATE(0352H)                   !Ctrl-Shift-R Key
CtrlShiftS      EQUATE(0353H)                   !Ctrl-Shift-S Key
CtrlShiftT      EQUATE(0354H)                   !Ctrl-Shift-T Key
CtrlShiftU      EQUATE(0355H)                   !Ctrl-Shift-U Key
CtrlShiftV      EQUATE(0356H)                   !Ctrl-Shift-V Key
CtrlShiftW      EQUATE(0357H)                   !Ctrl-Shift-W Key
CtrlShiftX      EQUATE(0358H)                   !Ctrl-Shift-X Key
CtrlShiftY      EQUATE(0359H)                   !Ctrl-Shift-Y Key
CtrlShiftZ      EQUATE(035AH)                   !Ctrl-Shift-Z Key
CtrlShiftF1     EQUATE(0370H)                   !Ctrl-Shift-F1 Key
CtrlShiftF2     EQUATE(0371H)                   !Ctrl-Shift-F2 Key
CtrlShiftF3     EQUATE(0372H)                   !Ctrl-Shift-F3 Key
CtrlShiftF4     EQUATE(0373H)                   !Ctrl-Shift-F4 Key
CtrlShiftF5     EQUATE(0374H)                   !Ctrl-Shift-F5 Key
CtrlShiftF6     EQUATE(0375H)                   !Ctrl-Shift-F6 Key
CtrlShiftF7     EQUATE(0376H)                   !Ctrl-Shift-F7 Key
CtrlShiftF8     EQUATE(0377H)                   !Ctrl-Shift-F8 Key
CtrlShiftF9     EQUATE(0378H)                   !Ctrl-Shift-F9 Key
CtrlShiftF10    EQUATE(0379H)                   !Ctrl-Shift-F10 Key
CtrlShiftF11    EQUATE(037AH)                   !Ctrl-Shift-F11 Key
CtrlShiftF12    EQUATE(037BH)                   !Ctrl-Shift-F12 Key
CtrlShiftAst    EQUATE(036AH)                   !Ctrl-Shift-Asterisk
CtrlShiftBS     EQUATE(0308H)                   !Ctrl-Shift-Backspace
CtrlShiftDecimal  EQUATE(036EH)                 !Ctrl-Shift-Decimal
CtrlShiftDelete   EQUATE(032EH)                 !Ctrl-Shift-Delete
CtrlShiftDivide EQUATE(036FH)                   !Ctrl-Shift-Divide Key
CtrlShiftDown   EQUATE(0328H)                   !Ctrl-Shift-Cursor Down
CtrlShiftEnd    EQUATE(0323H)                   !Ctrl-Shift-End Key
CtrlShiftEnter  EQUATE(030DH)                   !Ctrl-Shift-Enter Key
CtrlShiftEsc    EQUATE(031BH)                   !Ctrl-Shift-Esc Key
CtrlShiftHome   EQUATE(0324H)                   !Ctrl-Shift-Home Key
CtrlShiftInsert EQUATE(032DH)                   !Ctrl-Shift-Insert Key
CtrlShiftLeft   EQUATE(0325H)                   !Ctrl-Shift-Cursor Left
```

```
CtrlShiftMinus  EQUATE(036DH)                   !Ctrl-Shift-Minus Key
CtrlShiftPause  EQUATE(0313H)                   !Ctrl-Shift-Pause Key
CtrlShiftPgDn   EQUATE(0322H)                   !Ctrl-Shift-PgDn Key
CtrlShiftPgUp   EQUATE(0321H)                   !Ctrl-Shift-PgUp Key
CtrlShiftPlus   EQUATE(036BH)                   !Ctrl-Shift-Plus Key
CtrlShiftPrint  EQUATE(032CH)                   !Ctrl-Shift-PrintScreen
CtrlShiftRight  EQUATE(0327H)                   !Ctrl-Shift-Cursor
Right
CtrlShiftSlash  EQUATE(036FH)                   !Ctrl-Shift-Slash Key
CtrlShiftSpace  EQUATE(0320H)                   !Ctrl-Shift-Spacebar
CtrlShiftTab    EQUATE(0309H)                   !Ctrl-Shift-Tab Key
CtrlShiftUp     EQUATE(0326H)                   !Ctrl-Shift-Cursor Up
CtrlShiftPad0   EQUATE(0360H)           !Ctrl-Shift-0 on numeric keypad
CtrlShiftPad1   EQUATE(0361H)           !Ctrl-Shift-1 on numeric keypad
CtrlShiftPad2   EQUATE(0362H)           !Ctrl-Shift-2 on numeric keypad
CtrlShiftPad3   EQUATE(0363H)           !Ctrl-Shift-3 on numeric keypad
CtrlShiftPad4   EQUATE(0364H)           !Ctrl-Shift-4 on numeric keypad
CtrlShiftPad5   EQUATE(0365H)           !Ctrl-Shift-5 on numeric keypad
CtrlShiftPad6   EQUATE(0366H)           !Ctrl-Shift-6 on numeric keypad
CtrlShiftPad7   EQUATE(0367H)           !Ctrl-Shift-7 on numeric keypad
CtrlShiftPad8   EQUATE(0368H)           !Ctrl-Shift-8 on numeric keypad
CtrlShiftPad9   EQUATE(0369H)           !Ctrl-Shift-9 on numeric keypad
CtrlShiftMouseLeft    EQUATE(0301H)     !Ctrl-Shift-Left mouse button
CtrlShiftMouseRight  EQUATE(0302H)      !Ctrl-Shift-Right mouse button
CtrlShiftMouseCenter  EQUATE(0304H)     !Ctrl-Shift-Middle mouse button
CtrlAltO         EQUATE(0630H)                  !Ctrl-Alt-O Key
CtrlAlt1         EQUATE(0631H)                  !Ctrl-Alt-1 Key
CtrlAlt2         EQUATE(0632H)                  !Ctrl-Alt-2 Key
CtrlAlt3         EQUATE(0633H)                  !Ctrl-Alt-3 Key
CtrlAlt4         EQUATE(0634H)                  !Ctrl-Alt-4 Key
CtrlAlt5         EQUATE(0635H)                  !Ctrl-Alt-5 Key
CtrlAlt6         EQUATE(0636H)                  !Ctrl-Alt-6 Key
CtrlAlt7         EQUATE(0637H)                  !Ctrl-Alt-7 Key
CtrlAlt8         EQUATE(0638H)                  !Ctrl-Alt-8 Key
CtrlAlt9         EQUATE(0639H)                  !Ctrl-Alt-9 Key
CtrlAltA         EQUATE(0641H)                  !Ctrl-Alt-A Key
CtrlAltB         EQUATE(0642H)                  !Ctrl-Alt-B Key
CtrlAltC         EQUATE(0643H)                  !Ctrl-Alt-C Key
CtrlAltD         EQUATE(0644H)                  !Ctrl-Alt-D Key
CtrlAltE         EQUATE(0645H)                  !Ctrl-Alt-E Key
CtrlAltF         EQUATE(0646H)                  !Ctrl-Alt-F Key
CtrlAltG         EQUATE(0647H)                  !Ctrl-Alt-G Key
CtrlAltH         EQUATE(0648H)                  !Ctrl-Alt-H Key
CtrlAltI         EQUATE(0649H)                  !Ctrl-Alt-I Key
CtrlAltJ         EQUATE(064AH)                  !Ctrl-Alt-J Key
CtrlAltK         EQUATE(064BH)                  !Ctrl-Alt-K Key
CtrlAltL         EQUATE(064CH)                  !Ctrl-Alt-L Key
CtrlAltM         EQUATE(064DH)                  !Ctrl-Alt-M Key
CtrlAltN         EQUATE(064EH)                  !Ctrl-Alt-N Key
CtrlAltO         EQUATE(064FH)                  !Ctrl-Alt-O Key
CtrlAltP         EQUATE(0650H)                  !Ctrl-Alt-P Key
CtrlAltQ         EQUATE(0651H)                  !Ctrl-Alt-Q Key
CtrlAltR         EQUATE(0652H)                  !Ctrl-Alt-R Key
CtrlAltS         EQUATE(0653H)                  !Ctrl-Alt-S Key
CtrlAltT         EQUATE(0654H)                  !Ctrl-Alt-T Key
CtrlAltU         EQUATE(0655H)                  !Ctrl-Alt-U Key
CtrlAltV         EQUATE(0656H)                  !Ctrl-Alt-V Key
CtrlAltW         EQUATE(0657H)                  !Ctrl-Alt-W Key
CtrlAltX         EQUATE(0658H)                  !Ctrl-Alt-X Key
CtrlAltY         EQUATE(0659H)                  !Ctrl-Alt-Y Key
CtrlAltZ         EQUATE(065AH)                  !Ctrl-Alt-Z Key
CtrlAltF1        EQUATE(0670H)                  !Ctrl-Alt-F1 Key
```

```
CtrlAltF2       EQUATE(0671H)               !Ctrl-Alt-F2 Key
CtrlAltF3       EQUATE(0672H)               !Ctrl-Alt-F3 Key
CtrlAltF4       EQUATE(0673H)               !Ctrl-Alt-F4 Key
CtrlAltF5       EQUATE(0674H)               !Ctrl-Alt-F5 Key
CtrlAltF6       EQUATE(0675H)               !Ctrl-Alt-F6 Key
CtrlAltF7       EQUATE(0676H)               !Ctrl-Alt-F7 Key
CtrlAltF8       EQUATE(0677H)               !Ctrl-Alt-F8 Key
CtrlAltF9       EQUATE(0678H)               !Ctrl-Alt-F9 Key
CtrlAltF10      EQUATE(0679H)               !Ctrl-Alt-F10 Key
CtrlAltF11      EQUATE(067AH)               !Ctrl-Alt-F11 Key
CtrlAltF12      EQUATE(067BH)               !Ctrl-Alt-F12 Key
CtrlAltAst      EQUATE(066AH)               !Ctrl-Alt-Asterisk Key
CtrlAltBS       EQUATE(0608H)               !Ctrl-Alt-Backspace Key
CtrlAltDecimal  EQUATE(066EH)               !Ctrl-Alt-Decimal Key
CtrlAltDelete   EQUATE(062EH)               !Ctrl-Alt-Delete Key
CtrlAltDivide   EQUATE(066FH)               !Ctrl-Alt-Divide Key
CtrlAltDown     EQUATE(0628H)               !Ctrl-Alt-Cursor Down
CtrlAltEnd      EQUATE(0623H)               !Ctrl-Alt-End Key
CtrlAltEnter    EQUATE(060DH)               !Ctrl-Alt-Enter Key
CtrlAltEsc      EQUATE(061BH)               !Ctrl-Alt-Esc Key
CtrlAltHome     EQUATE(0624H)               !Ctrl-Alt-Home Key
CtrlAltInsert   EQUATE(062DH)               !Ctrl-Alt-Insert Key
CtrlAltLeft     EQUATE(0625H)               !Ctrl-Alt-Cursor Left
CtrlAltMinus    EQUATE(066DH)               !Ctrl-Alt-Minus Key
CtrlAltPause    EQUATE(0613H)               !Ctrl-Alt-Pause Key
CtrlAltPgDn     EQUATE(0622H)               !Ctrl-Alt-PgDn Key
CtrlAltPgUp     EQUATE(0621H)               !Ctrl-Alt-PgUp Key
CtrlAltPlus     EQUATE(066BH)               !Ctrl-Alt-Plus Key
CtrlAltPrint    EQUATE(062CH)               !Ctrl-Alt-PrintScreen
CtrlAltRight    EQUATE(0627H)               !Ctrl-Alt-Cursor Right
CtrlAltSlash    EQUATE(066FH)               !Ctrl-Alt-Slash Key
CtrlAltSpace    EQUATE(0620H)               !Ctrl-Alt-Spacebar
CtrlAltTab      EQUATE(0609H)               !Ctrl-Alt-Tab Key
CtrlAltUp       EQUATE(0626H)               !Ctrl-Alt-Cursor Up Key
CtrlAltPad0     EQUATE(0660H)           !Ctrl-Alt-0 on numeric keypad
CtrlAltPad1     EQUATE(0661H)           !Ctrl-Alt-1 on numeric keypad
CtrlAltPad2     EQUATE(0662H)           !Ctrl-Alt-2 on numeric keypad
CtrlAltPad3     EQUATE(0663H)           !Ctrl-Alt-3 on numeric keypad
CtrlAltPad4     EQUATE(0664H)           !Ctrl-Alt-4 on numeric keypad
CtrlAltPad5     EQUATE(0665H)           !Ctrl-Alt-5 on numeric keypad
CtrlAltPad6     EQUATE(0666H)           !Ctrl-Alt-6 on numeric keypad
CtrlAltPad7     EQUATE(0667H)           !Ctrl-Alt-7 on numeric keypad
CtrlAltPad8     EQUATE(0668H)           !Ctrl-Alt-8 on numeric keypad
CtrlAltPad9     EQUATE(0669H)           !Ctrl-Alt-9 on numeric keypad
CtrlAltMouseLeft    EQUATE(0601H)       !Ctrl-Alt-Left mouse button
CtrlAltMouseRight   EQUATE(0602H)       !Ctrl-Alt-Right mouse button
CtrlAltMouseCenter  EQUATE(0604H)       !Ctrl-Alt-Middle mouse button
```

Data Structure Properties

Contents

The attributes (properties) of many of the APPLICATION, WINDOW, and REPORT data structures, and their component controls, are designed take constant values (not variables) as their parameters in the data structure declaration. The same is true of FILE, VIEW, and QUEUE data structures. This may seem to be a restriction, however, the values of these constant properties may be easily changed or determined using simple assignment statements containing property expressions.

Property expressions represent the attributes (properties) and the parameters of attributes declared in APPLICATION, WINDOW, REPORT, FILE, VIEW, and QUEUE structures, and their components. Most attributes have corresponding property expressions. However, some attributes (such as PRE, OVER, and THREAD) are actually compiler directives which have no associated property expression. In addition, there are some property expressions which are not associated with declared attributes (undeclared properties).

A property expression can be used as the destination of an assignment statement. This changes the value of the attribute (or attribute parameter) associated with the property. A property expression can also be used in any string expression to determine the current value of the attribute (or attribute parameter).

Built-inVariables

There are three built-in variables in the Clarion for Windows runtime library: TARGET, PRINTER, and SYSTEM. These are only used with the property assignment syntax to identify the target of a property assignment.

TARGET normally references the window that currently has focus. It can also be set to reference a window in another execution thread or the currently printing REPORT, enabling you to affect the properties of controls and windows in other execution threads and dynamically change report control properties while printing. The SETTARGET procedure is used to change the TARGET variable's reference.

PRINTER references the printer properties used by the next REPORT opened (and all subsequest reports). This is used only with the Printer Properties.

SYSTEM is a built-in variable that specifies global properties used by the the entire application. There are several specific undeclared properties that may use the SYSTEM variable to set or query global application-wide properties.

Property Expressions

[*target*] [**$**] [*control*] { *property* [,*element*] }

target	The label of an APPLICATION, WINDOW, REPORT, VIEW, or FILE structure, the label of a BLOB, or one of the built-in variables: TARGET, PRINTER, or SYSTEM. If omitted, TARGET is assumed.
$	Required separator when both *target* and *control* are specified. May be omitted if either *target* or *control* is omitted.
control	A field number or field equate label for the control in the *target* structure (APPLICATION, WINDOW, or REPORT) to affect. If omitted, the *target* must be specified. The *control* must be omitted if the *target* is a FILE structure, the label of a BLOB, or the PRINTER or SYSTEM built-in variables.
property	An integer constant, EQUATE, or variable that specifies the property (attribute) to change. It can also be a string when referencing a .VBX property.
element	An integer constant or variable that specifies which element to change (for those attributes that are arrays with multiple values).

This property expression syntax allows you access to all the attributes (properties) of APPLICATION, WINDOW, or REPORT structures, or any control within these structures. To specify an attribute of an APPLICATION, WINDOW, REPORT, VIEW, or FILE structure (not a component control), omit the *control* portion of the property expression. To specify a control in the current window, omit the *target* portion of the property expression.

REPORT data structures are never the default *target*. Therefore, either SETTARGET must be used to change the default *target*, or the structure's label must be explicitly specified as the *target* before you can change any property of the structure, or any control it contains.

Property expressions may be used in Clarion language statements anywhere a string expression is allowed, or as the destination of a simple assignment statement. Therefore, assigning a new value to a property is an assignment with the property as the destination and the new value as the source. Determining the current value of a property is an assignment where the property is the source and the variable to recieve its value is the destination.

All properties are treated as string data at runtime; the compiler automatically performs any necessary data type conversion. Any property without parameters is binary. Binary properties are either "present" or "missing" and returns a '1' if it is present, and '' (null) if it is missing. Changing the value of a binary property to '' (null), '0' (zero), or any non-

numeric string sets it to missing. Changing it to any other value sets it to "present."

Most properties can be both examined (read) and changed (written). However, some properties are "read-only" and cannot be changed. Assigning a value to a "read-only" property has no effect at all. Other properties are "write-only" properties that are meaningless if read.

Some properties are arrays that contain multiple values. The syntax for addresssing a particular property array *element* uses a comma (not square brackets) as the delimiter between the *property* and the *element* number.

Example:

```
MainWin APPLICATION('My  Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,RESIZE
          MENUBAR
            MENU('File'),USE(?FileMenu)
              ITEM('Open...'),USE(?OpenFile)
              ITEM('Close'),USE(?CloseFile),DISABLE
              ITEM('E&xit'),USE(?MainExit)
            END
            MENU('Help'),USE(?HelpMenu)
              ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
              ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
              ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
              ITEM('About MyApp...'),USE(?HelpAbout)
            END
          END
          TOOLBAR
            BUTTON('Open'),USE(?OpenButton),ICON(ICON:Open)
          END
        END
  CODE
  OPEN(MainWin)
  MainWin{PROP:text} = 'A New Title'        !Change window title
  ?OpenButton{PROP:icon} = ICON:Asterisk    !Change button icon
  ?OpenButton{PROP:at,1} = 5                !Change button x position
  ?OpenButton{PROP:at,2} = 5                !Change button y position
  IF MainWin$?HelpContents{PROP:std} <> STD:HelpIndex
    MainWin$?HelpContents{PROP:std} = STD:HelpIndex
  END
  MainWin{PROP:maximize} = 1                !Expand to full screen
  ACCEPT
    CASE ACCEPTED()                         !Which control was chosen?
    OF ?OpenFile                            !Open... menu selection
    OROF ?OpenButton                        !Open button on toolbar
      START(OpenFileProc)                   !Start new execution thread
    OF ?MainExit                            !Exit menu selection
    OROF ?MainExitButton                    !Exit button on toolbar
      BREAK                                 !Break ACCEPT loop
    OF ?HelpAbout                           !About... menu selection
      HelpAboutProc                         !Call application information procedure
    END
  END
  CLOSE(MainWin)                            !Close APPLICATION
  RETURN
```

See Also: SETTARGET

Attribute Property Equates

Equates for all properties are contained in the PROPERTY.CLW file. This file also contains equates for the standard values used by some of these properties. Some properties are “read-only” and their value may not be changed, and others are “write-only” properties whose value cannot be determined. These restrictions are noted for each control affected.

Each of the following properties references an attribute (or one of its parameters) of a window, report, or control. The referenced attribute is listed in the explanation and you should look up the attribute itself for further explanation of its effect on the window or control it modifies.

Some property descriptions state: (‘’ if missing, else present), which means the attribute is either active for the window,report, or control, or it is not. Querying the property returns a blank string when the attribute is not active for the window, report, or control. Assigning a blank string (‘’) to such an attribute turns it off, and assigning any other value turns it on.

PROP:Text	The *text* parameter of an APPLICATION(*text*), WINDOW(*text*), or control(*text*). This could contain any value that is valid as the parameter to a control declaration. For example, ?Image{PROP:Text} = ‘My.BMP’ displays a new bitmap in the referenced IMAGE control.
PROP:Type	Contains the type of control. Values are the CREATE:xxxx equates (listed in EQUATES.CLW). (READ-ONLY)

AT attribute properties:

PROP:At	AT attribute. An array (4 values).
PROP:Xpos	AT(x) parameter, equivalent to {PROP:At,1}
PROP:Ypos	AT(,y) parameter, equivalent to {PROP:At,2}
PROP:Width	AT(,,width) parameter, equivalent to {PROP:At,3}
PROP:Height	AT(,,,height) parameter, equivalent to {PROP:At,4}

FONT attribute properties:

PROP:Font	FONT attribute. An array (4 values).
PROP:FontName	FONT(fontname) parameter, equivalent to {PROP:Font,1}.
PROP:FontSize	FONT(,fontsize) parameter, equivalent to {PROP:Font,2}.
PROP:FontColor	FONT(,,fontcolor) parameter, equivalent to {PROP:Font,3}.
PROP:FontStyle	FONT(,,,fontstyle) parameter, equivalent to {PROP:Font,4}.

CLASS attribute properties:

PROP:Class	CLASS attribute. An array (2 values).
PROP:VbxFile	CLASS(vbxfile) parameter, equivalent to {PROP:Class,1}.
PROP:VbxName	CLASS(,vbxname) parameter, equivalent to {PROP:Class,2}.

All other attribute properties (in alphabetical order):

PROP:Absolute	ABSOLUTE attribute (‘’ if missing, else present).
PROP:Alone	ALONE attribute (‘’ if missing, else present).
PROP:Alrt	ALRT attribute. An array.
PROP:Auto	AUTO attribute (‘’ if missing, else present).
PROP:Ave	AVE attribute (‘’ if missing, else present).
PROP:Boxed	ABSOLUTE attribute (‘’ if missing, else present).
PROP:Cap	ABSOLUTE attribute (‘’ if missing, else present).
PROP:Center	CENTER attribute (‘’ if missing, else present).
PROP:CenterOffset	CENTER(offset) parameter, equivalent to {PROP:Center,2}.
PROP:Check	CHECK attribute, (‘’ if missing, else present).
PROP:Cnt	CNT attribute (‘’ if missing, else present).
PROP:Color	COLOR attribute (COLOR:none if none).
PROP:Column	COLUMN attribute (0 = off, else currently highlighted column number).
PROP:Cursor	CURSOR attribute (‘’ if missing, else present).
PROP:Decimal	DECIMAL attribute (‘’ if missing, else present).
PROP:DecimalOffset	DECIMAL(offset) parameter, equivalent to {PROP:Decimal,2}.
PROP:Default	DEFAULT attribute (‘’ if missing, else present).
PROP:Disable	DISABLE attribute (‘’ if missing, else present).
PROP:Double	DOUBLE attribute (‘’ if missing, else present).
PROP:Dragid	DRAGID attribute. An array.
PROP:Drop	DROP attribute (0 if none). You may not change this to or from zero (0).
PROP:Dropid	DROPID attribute. An array.
PROP:Fill	FILL attribute (COLOR:none if none).

PROP:First	FIRST attribute (‘’ if missing, else present).
PROP:Format	FORMAT attribute (‘’ if missing, else present). This property is updated whenever the user changes the format of the LIST at runtime.
PROP:From	FROM attribute (queue, queue field, or string). (WRITE-ONLY)
PROP:Full	FULL attribute (‘’ if missing, else present).
PROP:Gray	GRAY attribute (‘’ if missing, else present).
PROP:Hide	HIDE attribute (‘’ if missing, else present).
PROP:Hlp	HLP attribute (blank if none).
PROP:Hscroll	HSCROLL attribute (‘’ if missing, else present).
PROP:Icon	ICON attribute (blank if none).
PROP:Iconize	ICONIZE attribute (‘’ if missing, else present).
PROP:Imm	IMM attribute (‘’ if missing, else present).
PROP:Ins	INS attribute (‘’ if missing, else present).
PROP:Key	KEY attribute (blank if none).
PROP:Landscape	LANDSCAPE attribute, (‘’ if missing, else present).
PROP:Last	LAST attribute (‘’ if missing, else present).
PROP:Left	LEFT attribute (‘’ if missing, else present).
PROP:LeftOffset	LEFT(offset) parameter, equivalent to {PROP:Left,2}.
PROP:Mark	MARK attribute (queue or queue field). (WRITE-ONLY)
PROP:Mask	MASK attribute (‘’ if missing, else present).
PROP:Max	MAX attribute (‘’ if missing, else present).
PROP:Maximize	MAXIMIZE attribute (‘’ if missing, else present).
PROP:Mdi	MDI attribute (‘’ if missing, else present). (READ-ONLY)
PROP:Meta	META attribute (‘’ if missing, else present).
PROP:Min	MIN attribute (‘’ if missing, else present).
PROP:Mm	MM attribute (‘’ if missing, else present).
PROP:Modal	MODAL attribute (‘’ if missing, else present). (READ-ONLY)
PROP:Msg	MSG attribute (‘’ if missing, else present).
PROP:NoBar	NOBAR attribute (‘’ if missing, else present).
PROP:NoFrame	NOFRAME attribute (‘’ if missing, else present).

PROP:NoMerge	NOMERGE attribute (‘’ if missing, else present).
PROP:Ovr	OVR attribute (‘’ if missing, else present).
PROP:Page	PAGE attribute (‘’ if missing, else present).
PROP:PageAfter	PAGEAFTER attribute (‘’ if missing, else present).
PROP:PageAfterNum	PAGEAFTER(pageafternum) parameter, equivalent to {PROP:PageAfter,2}.
PROP:PageBefore	PAGEBEFORE attribute (‘’ if missing, else present).
PROP:PageBeforeNum	PAGEBEFORE(pagebeforenum) parameter, equivalent to {PROP:PageBefore,2}.
PROP:Pageno	PAGENO attribute (‘’ if missing, else present).
PROP:Palette	PALETTE attribute. Single value.
PROP:Password	PASSWORD attribute (‘’ if missing, else present).
PROP:Points	POINTS attribute (‘’ if missing, else present).
PROP:Preview	PREVIEW attribute (queue or queue field). (WRITE-ONLY)
PROP:Range	RANGE attribute. An array (2 values).
PROP:RangeHigh	RANGE(,rangehigh) parameter, equivalent to {PROP:Range,2}.
PROP:RangeLow	RANGE(rangelow) parameter, equivalent to {PROP:Range,1}.
PROP:ReadOnly	READONLY attribute (‘’ if missing, else present).
PROP:Req	REQ attribute (‘’ if missing, else present).
PROP:Reset	RESET attribute (0 = off, else breaklevel nesting depth).
PROP:Resize	RESIZE attribute (‘’ if missing, else present).
PROP:Right	RIGHT attribute (‘’ if missing, else present).
PROP:RightOffset	RIGHT(offset) parameter, equivalent to {PROP:Right,2}.
PROP:Round	ROUND attribute (‘’ if missing, else present).
PROP:Scroll	SCROLL attribute (‘’ if missing, else present).
PROP:Separate	SEPARATE attribute (‘’ if missing, else present).
PROP:Skip	SKIP attribute (‘’ if missing, else present).
PROP:Spread	SPREAD attribute (‘’ if missing, else present).
PROP:Status	STATUS attribute. An array (0 terminates).
PROP:StatusText	STATUS bar text. An array (0 terminates).

PROP:Std	STD attribute (‘’ if missing, else present).
PROP:Step	STEP attribute (‘’ if missing, else present).
PROP:Sum	SUM attribute (‘’ if missing, else present).
PROP:System	SYSTEM attribute (‘’ if missing, else present).
PROP:Thous	THOUS attribute (‘’ if missing, else present).
PROP:Timer	TIMER attribute (0 if none).
PROP:Toolbox	TOOLBOX attribute (‘’ if missing, else present).
PROP:ToolTip	TIP attribute (‘’ if missing, else present).
PROP:Trn	TRN attribute, (‘’ if missing, else present).
PROP:Upr	UPR attribute (‘’ if missing, else present).
PROP:Use	USE attribute (variable name). Writing to it changes the USE variable. Reading it returns the contents of the USE variable.
PROP:Value	VALUE attribute (‘’ if missing, else present).
PROP:Vcr	VCR attribute (‘’ if missing, else present).
PROP:VcrFeq	VCR(vcrfeq) parameter, equivalent to {PROP:Vcr,2}.
PROP:Vscroll	VSCROLL attribute (‘’ if missing, else present).
PROP:WithNext	WITHNEXT attribute (0 if none).
PROP:WithPrior	WITHPRIOR attribute (0 if none).
PROP:Wizard	WIZARD attribute (‘’ if missing, else present).

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name),REQ
          IMAGE('SomePic.BMP'),USE(?Image)
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  Screen{PROP:At,1} = 0               !Position window to top left corner
  Screen{PROP:At,2} = 0
  Screen{PROP:Gray} = 1               !Give window 3D look
  Screen{PROP:Status,1} = -1          !Create status bar with two sections
  Screen{PROP:Status,2} = 180
  Screen{PROP:Status,3} = 0           !Terminate staus bar array
  Screen{PROP:StatusText,2} = FORMAT(TODAY(),@D2)
                                      !Put date in status bar section 2
  ?CtlCode{PROP:Alrt,1} = F10Key      !Alert F10 on Ctl:Code entry control
  ?CtlCode{PROP:Text} = '@N4'         !Change entry picture token
  ?Image{PROP:Text} = 'MyPic.BMP'     !Change image control filename
  ?OkButton{PROP:Default} = '1'       !Put DEFAULT attribute on OK button
  ACCEPT
  END
```

List Box Format String Properties

The properties of individual fields in a multi-column LIST or COMBO control can also be set using property equates.Each of these properties relates to one element of the FORMAT attribute's string parameter. These properties eliminate the need to create a complete FORMAT attribute string just to change a single property of a single field in the LIST.

These are all property arrays that require an explicit array element number following the property equate (separated by a comma) to specify which field in the LIST or COMBO is affected.

PROPLIST:Center
: The **C** that indicates center justification, (blank if missing, 1 if present).

PROPLIST:CenterOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Color
: The ***** (asterisk) that indicates color information for the field is contained in four LONG fields that immediately follow the data field in the QUEUE (or FROM attribute string), (blank if missing, 1 if present).

PROPLIST:Decimal
: The **D** that indicates decimal justification, (blank if missing, 1 if present).

PROPLIST:DecimalOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Fixed
: The **F** that specifies the field remains fixed at left edge of the list, (blank if missing, 1 if present).

PROPLIST:Header
: The ~*header*~ text for the field or group, (blank if missing, 1 if present).

PROPLIST:HeaderCenter
: The **C** that indicates center header justification, (blank if missing, 1 if present).

PROPLIST:HeaderCenterOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:HeaderDecimal
: The **D** that indicates decimal header justification, (blank if missing, 1 if present).

PROPLIST:HeaderDecimalOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:HeaderLeft
The **L** that indicates left header justification, (blank if missing, 1 if present).

PROPLIST:HeaderLeftOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:HeaderRight
The **R** that indicates right header justification, (blank if missing, 1 if present).

PROPLIST:HeaderRightOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Icon The **I** that indicates an icon number for the field is contained in a LONG field that immediately follows the data field in the QUEUE (or FROM attribute string), (blank if missing, 1 if present).

PROPLIST:LastOnLine
The / (slash) that indicates the next field in the group appears on the next line, (blank if missing, 1 if present).

PROPLIST:Left The **L** that indicates left justification, (blank if missing, 1 if present).

PROPLIST:LeftOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Locator
The **?** (question mark) that specifies the field for a locator, (blank if missing, 1 if present).

PROPLIST:Picture
The *@picture@* display format for the field, (blank if missing, 1 if present).

PROPLIST:Resize
The **M** that allows the user to resize the field or group, (blank if missing, 1 if present).

PROPLIST:RightBorder
The | (vertical bar) that places a right border on the field or group, (blank if missing, 1 if present).

PROPLIST:Right The **R** that indicates right justification, (blank if missing, 1 if present).

PROPLIST:RightOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Scroll The **S(***integer***)** that puts a scroll bar on the field or group. Specifies the *integer* portion, (blank if missing, 1 if present).

PROPLIST:Tree The **T** that indicates the LIST is a tree control, (blank if missing, 1 if present).

PROPLIST:TreeLines
The **T(L)** that indicates the tree control suppresses the conecting lines between levels, (blank if missing, 1 if present).

PROPLIST:TreeBoxes
The **T(B)** that indicates the tree control suppresses the expansion boxes, (blank if missing, 1 if present).

PROPLIST:TreeIndent
The **T(I)** that indicates the tree control suppresses level indentation (which also implicitly suppresses both lines and boxes), (blank if missing, 1 if present).

PROPLIST:Underline
The **_** (underscore) that underlines the field or group, (blank if missing, 1 if present).

PROPLIST:Width The integer that specifies the width of the field or group.

Any of these properties can also apply to a field group by adding PROPLIST:Group to the property.

PROPLIST:Group Add this property to the PROPLIST field property to affect field group properties.

Example:

```
?List{PROPLIST:Header,1} = 'First Field'          !Change first field's header text
?List{PROPLIST:Header + PROPLIST:Group,1} = 'First Group'
                                                 !Change first group's header text
```

See Also: FORMAT

Other Properties

List Box Mouse Click Properties

The following properties return the mouse position within the LIST or COMBO control when pressed or released.They can also be written to, which has no effect except to temporarily change the value that the property returns when next read (within the same ACCEPT loop iteration). This may make coding easier in some circumstances.

PROPLIST:MouseDownField
: Returns the field number when the mouse is pressed.

PROPLIST:MouseDownRow
: Returns the row number when the mouse is pressed.

PROPLIST:MouseDownZone
: Returns the zone number when the mouse is pressed.

PROPLIST:MouseMoveField
: Returns the field number when the mouse is moved.

PROPLIST:MouseMoveRow
: Returns the row number when the mouse is moved.

PROPLIST:MouseMoveZone
: Returns the zone number when the mouse is moved.

PROPLIST:MouseUpField
: Returns the field number when the mouse is released.

PROPLIST:MouseUpRow
: Returns the row number when the mouse is released.

PROPLIST:MouseUpZone
: Returns the zone number when the mouse is released.

The three “Row” properties all return -1 for header text and -2 if below the last displayed item.

Equates for the following Zones are listed on EQUATES.CLW:

LISTZONE:Field	On a field in the LIST
LISTZONE:Right	On the field’s right border resize zone
LISTZONE:Header	On a field or group header
LISTZONE:ExpandBox	On an expand box in a Tree
LISTZONE:Tree	On the connecting lines of a Tree
LISTZONE:Icon	On an icon (Tree or not)
LISTZONE:Nowhere	Anywhere else

Example:

```
Que          QUEUE
F1             STRING(50)
F2             STRING(50)
F3             STRING(50)
             END
WinView      WINDOW('View'),AT(,,340,200),SYSTEM,CENTER,ALRT(MouseLeft)
               LIST,AT(20,0,300,200),USE(?List),FROM(Que),IMM,HVSCROLL   |
                   FORMAT('80L~F1~80L~F2~80L~F3~'),IMM
             END
  CODE
  OPEN(WinView)
  DO BuildListQue
  X# = 0
  ACCEPT
    CASE EVENT()
    OF EVENT:AlertKey
      IF ?List{PROPLIST:MouseUpRow} = -1              !Check for click in header
        CASE ?List{PROPLIST:MouseDownField} + X#      !Check which header
        OF 1
          SORT(Que,Que:F1)
          ?List{PROP:Format} = '80L~F1~#1#80L~F2~#2#80L~F3~#3#'
          X# = 0
        OF 2
          SORT(Que,Que:F2)
          ?List{PROP:Format} = '80L~F2~#2#80L~F3~#3#80L~F1~#1#'
          X# = 1
        OF 3
          SORT(Que,Que:F3)
          ?List{PROP:Format} = '80L~F3~#3#80L~F1~#1#80L~F2~#2#'
          X# = 2
        END
        DISPLAY
  . . .
  FREE(Que)
```

Undeclared Properties

The following properties can only be accessed at runtime, and do not relate directly to data structure or control field attributes:

PROP:AcceptAll Returns one (1) if AcceptAll mode is active and zero (0) if it is not, and may also be used to toggle AcceptAll (non-stop) mode. SELECT with no parameters usually initiates AcceptAll mode. This is a field edit mode in which each control in the window is processed in TAB key sequence by generating EVENT:Accepted for each. This allows data entry validation code to execute for all controls, including those that the user has not touched.

AcceptAll mode immediately terminates when any of the following conditions is met:

SELECT(?)
Window{PROP:AcceptAll} = 0
A REQ control is left blank or zero.

The SELECT(?) statement selects the same control for the user to edit. This code usually indicates the value it contains is invalid and the user must re-enter data. The Window{PROP:AcceptAll} = 0 statement toggles AcceptAll mode off. Assigning values to this property can be used to initiate and terminate AcceptAll mode. When a control with the REQ attribute is left blank or zero, AcceptAll mode terminates with the control highlighted for user entry, without processing any more fields in the TAB key sequence.

When all controls have been successfully processed, EVENT:Completed is posted to the window.

Example:

```
Screen  WINDOW,PRE(Scr)
          ENTRY(@N3),USE(Ctl:Code)
          ENTRY(@S30),USE(Ctl:Name),REQ
          BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
          BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
        END
  CODE
  OPEN(Screen)
  ACCEPT
    IF EVENT() = EVENT:Completed THEN BREAK.    !AcceptAll mode terminated
    CASE ACCEPTED()
    OF ?Ctl:Code
      IF Ctl:Code > 150                    !If data entered is invalid
        BEEP                               ! alert the user and
        SELECT(?)                          ! make them re-enter the data
      END
    OF ?OkButton
      Screen{PROP:AcceptAll} = 1           !Initiate AcceptAll mode
  . .                                      !Terminate ACCEPT and CASE ACCEPTED
```

PROP:Active Returns 1 if the window is the active window, blank if not. Set to 1 to make the top window of a thread the active window.

Example:

```
  CODE
  OPEN(ThisWindow)
  X# = START(AnotherThread)              !Start another thread
  ACCEPT
    CASE EVENT()
    OF EVENT:LoseFocus                   !When this window is losing focus
      IF Y# <> X#                        ! check for the first focus change
        ThisWindpw{PROP:Active} = 1      ! and return focus to this thread
        Y# = X#                          ! then flag first focus change completed
  . . .
```

PROP:AppInstance
Returns the instance handle (HInstance) of the .EXE file for use in low-level API calls which require it. This is only used with the SYSTEM built-in variable. (READ-ONLY)

Example:

```
    PROGRAM
HInstance   LONG
  CODE
  OPEN(AppFrame)
  HInstance = SYSTEM{PROP:AppInstance}   !Get .EXE instance handle for later use
  ACCEPT
  END
```

PROP:ChoiceFeq Returns or sets the field number of the currently selected TAB in a SHEET, or RADIO in an OPTION structure.

Example:

```
WinView WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
            RADIO('Radio 1'),AT(0,0,20,20),USE(?R1)
            RADIO('Radio 2'),AT(20,0,20,20),USE(?R2)
          END
        END

  CODE
  OPEN(WinView)
  ?OptVar1{PROP:ChoiceFeq} = ?R1    !Select radio one
  ACCEPT
  END
```

PROP:ClientHandle

Returns the client window handle (the area of the window that contains the controls) for use with low-level Windows API calls that require it. (READ-ONLY)

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            END
MessageText     CSTRING('You cannot exit the program from this window ')
MessageCaption  CSTRING('No EVENT:CloseDown Allowed ')
TextAddr        LONG
CaptionAddr     LONG
RetVal          SHORT
  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseDown
      TextAddress = ADDRESS(MessageText)
      CaptionAddress = ADDRESS(MessageCaption)
      RetVal  =  MessageBox(WinView{PROP:ClientHandle},TextAddr,CaptionAddr,MB_OK)
                                  !Windows API call using a window handle
      CYCLE                       !Disallow program closedown from this window
    END
  END
```

PROP:ClientWndProc

Sets or gets the client window messaging procedure for use with low-level Windows API calls that require it. Generally used with sub-classing to track all Windows messages.

Example:

```
    PROGRAM
    MAP
      main
      SubClassFunc(USHORT,SHORT,SHORT,LONG),LONG,PASCAL
      MODULE('Windows')                          !TopSpeed Win31 Library
        CallWindowProc(LONG,USHORT,SHORT,SHORT,LONG),LONG,PASCAL
      END
    END
SavedProc   LONG
PT          GROUP,PRE(PT)
X             SHORT
Y             SHORT
            END
  CODE
  Main

Main        PROCEDURE
WinView     WINDOW('View'),AT(0,0,320,200),HVSCROLL,MAX,TIMER(1)
              STRING('X  Pos'),AT(1,1,,),USE(?String1)
              STRING(@n3),AT(24,1,,),USE(PT:X)
              STRING('Y  Pos'),AT(44,1,,),USE(?String2)
              STRING(@n3),AT(68,1,,),USE(PT:Y)
              BUTTON('Close'),AT(240,180,60,20),USE(?Close)
            END
  CODE
  OPEN(WinView)
  SavedProc = WinView{PROP:ClientWndProc}            !Save this procedure
  WinView{PROP:ClientWndProc} = ADDRESS(SubClassFunc)
                                                     !Change to subclass procedure
  ACCEPT
    CASE ACCEPTED()
    OF ?Close
      BREAK
    END
  END

SubClassFunc    FUNCTION(hWnd,wMsg,wParam,lParam)    !Sub class procedure
WM_MOUSEMOVE    EQUATE(0200H)                        ! to track mouse movement in
  CODE                                               ! client area of window
  CASE wMsg
  OF  WM_MOUSEMOVE
    PT:X = MOUSEX()
    PT:Y = MOUSEY()
  END
  RETURN(CallWindowProc(SavedProc,hWnd,wMsg,wParam,lParam))
                                                     !Pass control back to
                                                     ! saved procedure
```

PROP:ClipBits Property of an IMAGE control that allows bitmap images to be moved into (but not out of) the Windows clipboard when set to one (1). Only .BMP, .PCX, or .GIF image types can be stored as a bitmap (.BMP) image in the Clipboard.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(),AT(0,0,,),USE(?Image)
              BUTTON('Save  Picture'),AT(80,180,60,20),USE(?SavePic)
              BUTTON('New  Picture'),AT(160,180,60,20),USE(?NewPic)
            END

FileName    STRING(64)                        !Filename variable

  CODE
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                    !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                                 !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
    OF ?SavePic
      ?Image{PROP:ClipBits} = 1               !Put image into Clipboard
      ENABLE(?LastPic)                        ! activate Last Picture button
    END
  END
```

PROP:ConnectString

Property of a FILE usingthe ODBC diriver that returns the connection string (normally stored in the file's OWNER attribute) that would allow a complete connection. If the OWNER attribute contains only a data source name, a login screen appears to ask for the rest of the required details before the connection is made. This login window appears every time you log on. With this property, the developer can enter information in the login screen once, then set the OWNER attribute to the return value from PROP:ConnectString, eliminating the login.

Example:

```
OwnerString   STRING(20)
Customer  FILE,DRIVER('ODBC'),OWNER(OwnerString)
Record       RECORD
Name           STRING(20)
          . .
  CODE
  OwnerString = 'DataSourceName'
  OPEN(Customer)
  OwnerString = Customer{PROP:ConnectString}    !Get full connect string
  MESSAGE(OwnerString)                             !Display it for future use
```

PROP:DDETimeOut

A property of the SYSTEM built-in variable that allows you to set and get the DDE timeout used when using DDEWRITE in DDE:manual mode. This value is in hundredths of seconds and the default value is 500.

Example:

```
DDERetVal   STRING(20)
WinOne      WINDOW,AT(0,0,160,400)
              ENTRY(@s20),USE(DDERetVal)
            END
MyServer    LONG
  CODE
  OPEN(WinOne)
  MyServer = DDESERVER('MyApp','DataEntered')     !Open as server
  SYSTEM{PROP:DDETimeOut} = 1000                  !Set time out to ten seconds
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest                           !Data requested once
      DDEWRITE(MyServer,DDE:manual,'DataEntered',DDERetVal)
                                                  !Provide data once
    END
  END
```

PROP:DeferMove A property of the SYSTEM built-in variable that defers the resizing and/or movement of controls until the end of the ACCEPT loop or SYSTEM{PROP:DeferMove} is reset to zero (0). This disables the immediate effect of all assignments to position and size properties, and enables the library to perform all the moves at once (eliminating possible temporarily overlapping controls).

The absolute value of the number assigned to SYSTEM{PROP:DeferMove} defines the number of deferred moves for which space is pre-allocated (automatically expanded when necessary, but less efficient and may fail). Assigning a positive number automatically resets it to zero at the next ACCEPT, while a negative number leaves it set until explicitly reset to zero (0).

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(),AT(0,0,,),USE(?Image)
              BUTTON('New  Picture'),AT(160,180,60,20),USE(?NewPic)
              BUTTON('Close'),AT(80,180,60,20),USE(?Close)
            END
FileName    STRING(64)                        !Filename variable
ImageWidth  SHORT
ImageHeight SHORT
  CODE
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                      !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                                   !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
      SYSTEM{PROP:DeferMove} = 4                !Defer move and resize
      ImageWidth = ?Image{PROP:Width}
      ImageHeight = ?Image{PROP:Height}
      IF ImageWidth > 320
        ?Image{PROP:Width} = 320
        ?Image{PROP:XPos} = 0
      ELSE
        ?Image{PROP:XPos} = (320 - ImageWidth) / 2      !Center horizontally
      END
      IF ImageHeight > 180
        ?Image{PROP:Height} = 180
        ?Image{PROP:YPos} = 0
      ELSE
        ?Image{PROP:YPos} = (180 - ImageHeight) / 2     !Center vertically
      END
    OF ?Close
      BREAK
  . .                       !Moves and resizing happen at end of ACCEPT loop
```

PROP:Edit Specifies the field equate label of the control to perform edit-in-place for a LIST box column. This is an array whose element number indicates the column number to edit. When non-zero, the control is unhidden and moved/resized over the current row in the column indicated to allow the user to input data. Assign zero to re-hide the data entry control.

Example:

```
Q     QUEUE
f1      STRING(15)
f2      STRING(15)
      END
Win1  WINDOW('List Edit In Place'),AT(0,1,308,172),SYSTEM
       LIST,AT(6,6,120,90),USE(?List),COLUMN,FORMAT('60L@s15@60L@s15@'),FROM(Q),IMM
      END
?EditEntry  EQUATE(100)
  CODE
  OPEN(Win1)
  CREATE(?EditEntry,CREATE:Entry)
  ACCEPT
    CASE FIELD()
    OF ?List
      CASE EVENT()
      OF EVENT:NewSelection
        IF ?List{PROP:edit,?List{PROP:column}}
          GET(Q,CHOICE())
        END
      OF EVENT:Accepted
        IF KEYCODE() = MouseLeft2
          GET(Q,CHOICE())
          ?EditEntry{PROP:text} = ?List{PROPLIST:picture,?List{PROP:column}}
          CASE ?List{PROP:column}
          OF 1
            ?EditEntry{PROP:use} = F1
          OF 2
            ?EditEntry{PROP:use} = F2
          END
          ?List{PROP:edit,?List{PROP:column}} = ?EditEntry
      . .
    OF ?EditEntry
      CASE EVENT()
      OF EVENT:Selected
        ?EditEntry{PROP:Touched} = 1
      OF EVENT:Accepted
        PUT(Q)
        ?List{PROP:edit,?List{PROP:column}} = 0
  . . .
```

PROP:Enabled Returns an empty string if the control is not enabled either because it itself has been disabled, or because it is a member of a "parent" control (OPTION, GROUP, MENU, SHEET, or TAB) that has been disabled. (READ-ONLY)

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            SHEET,AT(0,0,320,175),USE(SelectedTab)
              TAB('Tab One'),USE(?TabOne)
                PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
                ENTRY(@S8),AT(100,140,32,20),USE(E1)
                PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
                ENTRY(@S8),AT(100,240,32,20),USE(E2)
              END
              TAB('Tab Two'),USE(?TabTwo)
                PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
                ENTRY(@S8),AT(100,140,32,20),USE(E3)
                PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
                ENTRY(@S8),AT(100,240,32,20),USE(E4)
              END
            END
            BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
            BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
          END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE EVENT()
    OF EVENT:Completed
      BREAK
    END
    CASE FIELD()
    OF ?Ok
      CASE EVENT()
      OF EVENT:Accepted
        SELECT
      END
    OF ?E3
      CASE EVENT()
      OF EVENT:Accepted
        IF ?E3{PROP:Enabled} AND MDIChild{PROP:AcceptAll}
                                        !Check for visibility during AcceptAll mode
          E3 = UPPER(E3)                !Convert the data entered to Upper case
          DISPLAY(?E3)                  ! and display the upper cased data
        END
      END
    OF ?Cancel
      CASE EVENT()
      OF EVENT:Accepted
        BREAK
      END
    END
  END
```

PROP:Filter Sets the FILTER attribute of a VIEW structure.

Example:

```
BRW1::View:Browse VIEW(Members)
                    PROJECT(Mem:MemberCode,Mem:LastName,Mem:FirstName)
                  END
KeyValue  STRING(20)

  CODE
  KeyValue = 'Smith'
  BIND('KeyValue',KeyValue)
  BIND('Mem:LastName',Mem:LastName)
  Mem:LastName = KeyValue
  SET(Mem:LastNameKey,Mem:LastNameKey)
  BRW1::View:Browse{PROP:Filter} = 'Mem:LastName = KeyValue'
  OPEN(BRW1::View:Browse)
```

PROP:FlushPreview

Flushes the REPORT structure's PREVIEW attribute metafiles to the printer (0 = off, else on, always 0 at report open).

Example:

```
SomeReport  PROCEDURE

WMFQue      QUEUE                   !Queue to contain .WMF filenames
              STRING(64)
            END

NextEntry   BYTE(1)                 !Queue entry counter variable

Report    REPORT,PREVIEW(WMFQue)    !Report with PREVIEW attribute
DetailOne   DETAIL
              !Report controls
            END
          END

ViewReport  WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(),AT(0,0,320,180),USE(?ImageField)
              BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
              BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
              BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
            END

  CODE
  OPEN(Report)
  SET(SomeFile)                             !Code to generate the report
  LOOP
    NEXT(SomeFile)
      IF ERRORCODE() THEN BREAK.
    PRINT(DetailOne)
  END
  ENDPAGE(Report)
  OPEN(ViewReport)                          !Open report preview window
  GET(WMFQue,NextEntry)                     !Get first queue entry
  ?ImageField{PROP:text} = WMFQue           !Load first report page
  ACCEPT
    CASE ACCEPTED()
    OF ?NextPage
      NextEntry += 1                        !Increment entry counter
      IF NextEntry > RECORDS(WMFQue) THEN CYCLE. !Check for end of report
      GET(WMFQue,NextEntry)                 !Get next queue entry
      ?ImageField{PROP:text} = WMFQue       !Load next report page
      DISPLAY                               ! and display it
    OF ?PrintReport
      Report{PROP:FlushPreview} = 1         !Flush files to printer
      BREAK                                 ! and exit procedure
    OF ?ExitReport
      BREAK                                 !Exit procedure
    END
  END
  RETURN                                    !Return to caller, automatically
                                            ! closing the window and report
                                            ! freeing the queue and automatically
                                            ! deleting all the temporary .WMF files
```

PROP:Follows Changes the tab order to specify the position within the parent that the control will occupy. The control follows the control number you specify in the tab order. This must specify an existing control within the parent (window, option, group). (WRITE-ONLY)

Example:

```
WinView     WINDOW('View  Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              BUTTON('View  Next  Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
              BUTTON('Print  Report'),AT(80,180,60,20),USE(?PrintReport)
              BUTTON('Exit  Without  Printing'),AT(160,180,60,20),USE(?ExitReport)
            END
  CODE
  OPEN(WinView)
                        !Print Report button normally follows View button
  ?PrintReport{PROP:Follows} = ?ExitReport
                        !Now Print Report button follows Exit button in the tab order
  ACCEPT
  END
```

PROP:Handle Returns the window or control handle for use with low-level Windows API calls that require it.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            END
MessageText     CSTRING('You cannot exit the program from this window ')
MessageCaption  CSTRING('No EVENT:CloseDown Allowed ')
TextAddress     LONG
CaptionAddress  LONG
RetVal          SHORT
  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseDown
      TextAddress = ADDRESS(MessageText)
      CaptionAddress = ADDRESS(MessageCaption)
      RetVal  =  MessageBox(WinView{PROP:Handle},TextAddress,CaptionAddress,MB_OK)
                                  !Windows API call using a window handle
      CYCLE                       !Disallow program closedown from this window
    END
  END
```

PROP:HscrollPos Returns the position of the horizontal scroll bar's "thumb" (from 0 to 255) on a window, IMAGE, TEXT, LIST or COMBO with the HSCROLL attribute. Setting this property causes the control or window's contents to scroll horizontally.

Example:

```
Que          QUEUE
F1             STRING(50)
F2             STRING(50)
F3             STRING(50)
             END
WinView      WINDOW('View'),AT(,,340,200),SYSTEM,CENTER
               LIST,AT(20,0,300,200),USE(?List),FROM(Que),IMM,HVSCROLL   |
                   FORMAT('80L#1#80L#2#80L#3#')
             END
  CODE
  OPEN(WinView)
  DO BuildListQue
  ACCEPT
    CASE FIELD()
    OF ?List
      CASE EVENT()
      OF EVENT:ScrollDrag
        CASE ?List{PROP:HscrollPos} % 200) + 1
        OF 1
          ?List{PROP:Format} = '80L#1#80L#2#80L#3#'
        OF 2
          ?List{PROP:Format} = '80L#2#80L#3#80L#1#'
        OF 3
          ?List{PROP:Format} = '80L#3#80L#1#80L#2#'
        END
        DISPLAY
  . . .
  FREE(Que)

BuildListQue  ROUTINE
  LOOP 15 TIMES
    Que:F1 = 'F1F1F1F1'
    Que:F2 = 'F2F2F2F2'
    Que:F3 = 'F3F3F3F3'
    ADD(Que)
  END
```

PROP:IconList An array that sets the icons displayed in a LIST formatted to display icons (usually a tree control).

Example:

```
      PROGRAM
      MAP
        RandomAlphaData(*STRING)
      END

TreeDemo        QUEUE,PRE()         !Data list box FROM queue
FName             STRING(20)
ColorNFG          LONG              !Normal Foreground color for FName
ColorNBG          LONG              !Normal Background color for FName
ColorSFG          LONG              !Selected Foreground color for FName
ColorSBG          LONG              !Selected Background color for FName
IconField         LONG              !Icon number for FName
TreeLeve          LONG              !Tree Level
LName             STRING(20)
Init              STRING(4)
                END

Win WINDOW('List Boxes'),AT(0,0,366,181),SYSTEM,DOUBLE
      LIST,AT(0,34,366,146),FROM(TreeDemo),USE(?Show),HVSCROLL,    |
            FORMAT('80L*IT~First Name~*80L~Last Name~16C~Initials~')
    END

  CODE
  LOOP 20 TIMES
    RandomAlphaData(FName)
    ColorNFG = COLOR:White          !Assign FNAME's colors
    ColorNBG = COLOR:Maroon
    ColorSFG = COLOR:Yellow
    ColorSBG = COLOR:Blue
    IconField = ((x#-1) % 4) + 1    !Assign icon number
    TreeLevel = ((x#-1) % 4) + 1    !Assign tree level
    RandomAlphaData(LName)
    RandomAlphaData(Init)
    ADD(TD)
  END
  OPEN(Win)
  ?Show{PROP:iconlist,1} = ICON:VCRback          !Icon 1 = <
  ?Show{PROP:iconlist,2} = ICON:VCRrewind        !Icon 2 = <<
  ?Show{PROP:iconlist,3} = ICON:VCRplay          !Icon 3 = >
  ?Show{PROP:iconlist,4} = ICON:VCRfastforward   !Icon 4 = >>
  ACCEPT
  END

RandomAlphaData PROCEDURE(Field)        !MAP Prototype is: RandomAlphaData(*STRING)
  CODE
  y# = RANDOM(1,SIZE(Field))            !Random fill size
  LOOP x# = 1 to y#                     !Fill each character with
    Field[x#] = CHR(RANDOM(97,122))     ! a random lower case letter
  END
```

PROP:ImageBits Property of an IMAGE control that allows bitmap images displayed in the control to be moved into and out of memo fields. Any image displayed in the control can be stored.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(),AT(0,0,,),USE(?Image)
              BUTTON('Save  Picture'),AT(80,180,60,20),USE(?SavePic)
              BUTTON('New  Picture'),AT(160,180,60,20),USE(?NewPic)
              BUTTON('Last  Picture'),AT(240,180,60,20),USE(?LastPic)
            END

SomeFile    FILE,DRIVER('Clarion'),PRE(Fil)   !A file with a memo field
MyMemo        MEMO(65520),BINARY
Rec           RECORD
F1              LONG
            . .

FileName    STRING(64)                        !Filename variable

  CODE
  OPEN(SomeFile)
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                    !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                                 !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
    OF ?SavePic
      Fil:MyMemo = ?Image{PROP:ImageBits}     !Put image into memo
      ADD(SomeFile)                           ! and save it to the file on disk
      ENABLE(?LastPic)                        ! activate Last Picture button
    OF ?LastPic
      ?Image{PROP:ImageBits} = Fil:MyMemo     !Put last saved memo into image
    END
  END
```

PROP:ImageBlob Property of an IMAGE control that allows bitmap images displayed in the control to be moved into and out of BLOB fields. Any image displayed in the control can be stored.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(),AT(0,0,,),USE(?Image)
              BUTTON('Save  Picture'),AT(80,180,60,20),USE(?SavePic)
              BUTTON('New  Picture'),AT(160,180,60,20),USE(?NewPic)
              BUTTON('Last  Picture'),AT(240,180,60,20),USE(?LastPic)
            END

SomeFile    FILE,DRIVER('TopSpeed'),PRE(Fil)    !A file with a memo field
MyBlob        BLOB,BINARY
Rec           RECORD
F1              LONG
            . .

FileName    STRING(64)                        !Filename variable

  CODE
  OPEN(SomeFile)
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                          !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                                       !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
    OF ?SavePic
      Fil:MyBlob{PROP:Handle} = ?Image{PROP:ImageBlob}     !Put image into BLOB
      ADD(SomeFile)                                 ! and save it to the file on disk
      ENABLE(?LastPic)                              ! activate Last Picture button
    OF ?LastPic
      ?Image{PROP:ImageBlob} = Fil:MyBlob{PROP:Handle}
                                                    !Put last saved BLOB into image
    END
  END
```

PROP:Items Returns the number of entries visible in a LIST or COMBO control. (READ-ONLY)

Example:

```
Que           QUEUE
              STRING(30)
              END

WinView       WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM
               LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
              END

  CODE
  OPEN(WinView)
  SET(SomeFile)
  LOOP ?List{PROP:Items} TIMES    !Fill display queue to limit of displayable items
    NEXT(SomeFile)
    Que = Fil:Record
    ADD(Que)
  END
  ACCEPT
  END
```

PROP:LazyDisplay

Disables (when set to 1) or enables (when set to 0, the default) the feature where all window re-painting is completely done before processing continues with the next statement following a DISPLAY. Setting PROP:LazyDisplay = 1 creates seemingly faster video processing, since the re-paints occur at the end of the ACCEPT loop if there are no other messages pending. This can improve the performance of some applications, but can also have a negative impact on appearance.

Example:

```
WinView       APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
              END

  CODE
  OPEN(WinView)
  SYSTEM{PROP:LazyDisplay} = 1    !Disable extra paint message display
                                  ! throughout entire application
  ACCEPT
  END
```

PROP:Line An array whose elements each contain one line of the text in a TEXT control. (READ ONLY)

PROP:LineCount Returns the number of lines of text in a TEXT control. (READ ONLY)

Example:

```
LineCount SHORT
MemoLine  STRING(80)

CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail1     DETAIL,AT(0,0,6500,6000)
              TEXT,AT(0,0,6500,6000),USE(Fil:MemoField)
            END
Detail2     DETAIL,AT(0,0,6500,125)
              STRING(@s80),AT(0,0,6500,125),USE(MemoLine)
            END
          END

  CODE
  OPEN(File)
  SET(File)
  OPEN(CustRpt)
  LOOP
    NEXT(File)
    LIneCount = ?Fil:MemoField{PROP:LineCount}
    LOOP X# = 1 TO LIneCount
      MemoLine = ?Fil:MemoField{PROP:Line,X#}
      PRINT(Detail2)
    END
  END
```

PROP:LoginTimeOut

Property of a FILE using the ODBC driver that allows you to set the login timeout. This value is in seconds.

Example:

```
OwnerString   STRING(20)
Customer  FILE,DRIVER('ODBC'),OWNER(OwnerString)
Record       RECORD
Name           STRING(20)
          . .
  CODE
  OwnerString = 'DataSourceName'
  Customer{PROP:LoginTimeOut} = 30    !Set timeout to 30 seconds
  OPEN(Customer)
  OwnerString = Customer{PROP:ConnectString}    !Get full connect string
    MESSAGE(OwnerString)                              !Display it for future use
```

PROP:MaxHeight Sets or returns the maximum height of a resizable window.

PROP:MaxWidth Sets or returns the maximum width of a resizable window.

PROP:MinHeight Sets or returns the minimum height of a resizable window.

PROP:MinHeight Sets or returns the minimum width of a resizable window.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
              LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
            END

  CODE
  OPEN(WinView)
  WinView{PROPMaxHeight} = 200  !Set boundaries beyond which the user cannot
  WinView{PROPMaxWidth} = 320   ! resize the window
  WinView{PROPMinHeight} = 90
  WinView{PROPMinWidth} = 120
  ACCEPT
  END
```

PROP:NoTips Disables (when set to 1) or re-enables (when set to 0) tooltip display (TIP attribute) for the SYSTEM, window, or control.

Example:

```
WinView     APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
            END
  CODE
  OPEN(WinView)
  SYSTEM{PROP:NoTips} = 1   !Disable TIP display throughout entire application
  ACCEPT
  END
```

PROP:Progress You can directly update the display of a PROGRESS control by assigning a value (which must be within the range defined by the RANGE attribute) to the control's PROP:progress property.

Example:

```
BackgroundProcess   PROCEDURE         !Background processing batch process

Win   WINDOW('Batch  Processing...'),AT(,,400,400),TIMER(1),MDI,CENTER
        PROGRESS,AT(100,140,200,20),USE(?ProgressBar),RANGE(0,200)
        BUTTON('Cancel'),AT(190,300,20,20),STD(STD:Close)
      END

  CODE
  OPEN(Win)
  OPEN(File)
  ?ProgressBar{PROP:rangehigh} = RECORDS(File)
  SET(File)                           !Set up a batch process
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseWindow
      BREAK
    OF EVENT:Timer                          !Process records when timer allows it
      ProgressVariable += 3                 !Auto-updates 1st progress bar
      LOOP 3 TIMES
        NEXT(File)
        IF ERRORCODE() THEN BREAK.
        ?ProgressBar{PROP:progress} += 1    !Manually update progress bar
        !Perform some batch processing code
  . . .
  CLOSE(File)
```

PROP:ScreenText Returns the text displayed on screen in the specified ENTRY or entry-like (SPIN/COMBO) control.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              SPIN(@n3),AT(0,0,320,180),USE(Fil:Field),RANGE(0,255)
            END
  CODE
  OPEN(WinView)
  ACCEPT
    CASE FIELD()
    OF ?Fil:Field
      CASE EVENT()
      OF EVENT:Rejected
        MESSAGE(?Fil:Field{PROP:ScreenText} & ' is not in the range 0-255')
        SELECT(?)
        CYCLE
      END
    END
  END
```

PROP:SelStart Sets or retrieves the beginning (inclusive) character to mark as a block in an ENTRY or TEXT control. It positions the data entry cursor left of the character, and sets PROP:SelEnd to zero (0) to indicate no block is marked.

PROP:SelEnd Sets or retrieves the ending (inclusive) character to mark as a block in an ENTRY or TEXT control.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              ENTRY(@S30),AT(0,0,320,180),USE(Fil:Field),ALRT(F10Key)
            END
  CODE
  OPEN(WinView)
  ACCEPT
    CASE ACCEPTED()
    OF ?Fil:Field
      SETCLIPBOARD(Fil:Field[?Fil:Field{PROP:SelStart}:?Fil:Field{PROP:SelEnd}])
                    !Place highlighted string slice in Windows' clipboard
    END
  END
```

PROP:Size Returns (or sets) the size of a BLOB field.

Example:

```
Names     FILE,DRIVER('TopSpeed')
NbrKey      KEY(Names:Number)
Notes       BLOB              !Can be larger than 64K
Rec         RECORD
Name          STRING(20)
Number        SHORT
          . .

BlobSize    LONG
BlobBuffer1 STRING(65520),STATIC    !Maximum size string
BlobBuffer2 STRING(65520),STATIC    !Maximum size string

WinView     WINDOW('View BLOB Contents'),AT(0,0,320,200),SYSTEM
              TEXT,AT(0,0,320,180),USE(BlobBuffer1),VSCROLL
              TEXT,AT(0,190,320,180),USE(BlobBuffer2),VSCROLL,HIDE
            END
  CODE
  OPEN(Names)
  SET(Names)
  NEXT(Names)
  OPEN(WinView)
  BlobSize = Names:Notes{PROP:Size}   !Get size of BLOB contents
  IF BlobSize > 65520
    BlobBuffer1 = Names:Notes[1:65520]
    BlobBuffer2 = Names:Notes[65521:BlobSize]
    WinView{PROP:Height} = 400
    UNHIDE(?BlobBuffer2)
  ELSE
    BlobBuffer1 = Names:Notes[1:BlobSize]
  END
  ACCEPT
  END
```

PROP:Thread Returns the thread number of a window. This is not necessarily the currently executing thread, if you've used SETTARGET to set the TARGET built-in variable. (READ-ONLY)

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM
            END
ToolboxThread BYTE
  CODE
  OPEN(WinView)
  ToolboxThread = ToolboxWin{PROP:Thread}     !Get window thread number
  ACCEPT
  END
```

PROP:TipDelay Sets the time delay before tooltip display (TIP attribute) for the SYSTEM (16-bit only).

PROP:TipDisplay Sets the duration of tooltip display (TIP attribute) for the SYSTEM (16-bit only).

Example:

```
WinView     APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
            END

  CODE
  OPEN(WinView)
  SYSTEM{PROP:TipDelay} = 50      !Delay TIP display for 1/2 second
  SYSTEM{PROP:TipDisplay} = 500   !TIP display for 5 seconds
  ACCEPT
  END
```

PROP:Touched When non-zero, indicates the data in the ENTRY, TEXT, SPIN, or COMBO control with input focus has been changed by the user since the last EVENT:Accepted. Automatically reset to zero each time the control generates an EVENT:Accepted.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              ENTRY(@S30),AT(0,0,320,180),USE(Fil:Field)
            END
  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:Selected
      ?Fil:Field{PROP:Touched} = 1      !Force an EVENT:Accepted to generate
    OF EVENT:Accepted
      !Process the data, whether entered by the user or in the field at the start
    END
  END
```

PROP:TrueValue Sets the value received by the USE variable of a CHECK box when the user checks it on. This overrides the default assigned value of one (1).

PROP:FalseValue Sets the value received by the USE variable of a CHECK box when the user checks it off. This overrides the default assigned value of zero (0).

Example:

```
CheckField  STRING(1)

WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              CHECK('True or False'),AT(0,0,,),USE(CheckField)
            END
  CODE
  OPEN(WinView)
  ?CheckField{PROP:TrueValue} = 'T'
  ?CheckField{PROP:FalseValue} = 'F'
  ACCEPT
  END
```

PROP:VBXEvent Returns the name of a VBX event. (READ-ONLY)

PROP:VBXEventArg VBX event parameters. An array.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
              CUSTOM,USE(?Graph),CLASS('graph.vbx','graph'),'graphstyle'('2')
            END

  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:VBXEvent
      IF ?Graph{PROP:VBXEvent} = 'FooEvent'                !Check event name
        ProcessFoo(?Graph{PROP:VBXEventArg,1},?Graph{PROP:VBXEventArg,2})
                  !Get 1st and 2nd event parameters and pass to process procedure
      END
    END
  END
```

PROP:Visible Returns an empty string if the control is not visible because either because it has been hidden, or it is a member of a "parent" control (OPTION, GROUP, MENU, SHEET, or TAB) that is hidden, or is on a TAB control page that is not currently selected. (READ-ONLY)

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            SHEET,AT(0,0,320,175),USE(SelectedTab)
              TAB('Tab One'),USE(?TabOne)
                PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
                ENTRY(@S8),AT(100,140,32,20),USE(E1)
                PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
                ENTRY(@S8),AT(100,240,32,20),USE(E2)
              END
              TAB('Tab Two'),USE(?TabTwo)
                PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
                ENTRY(@S8),AT(100,140,32,20),USE(E3)
                PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
                ENTRY(@S8),AT(100,240,32,20),USE(E4)
              END
            END
            BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
            BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
          END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE EVENT()
    OF EVENT:Completed
      BREAK
    END
    CASE FIELD()
    OF ?Ok
      CASE EVENT()
      OF EVENT:Accepted
        SELECT
      END
    OF ?E3
      CASE EVENT()
      OF EVENT:Accepted
        E3 = UPPER(E3)                  !Convert the data entered to Upper case
        IF ?E3{PROP:Visible} AND MDIChild{PROP:AcceptAll}
                                        !Check for visibility during AcceptAll mode
          DISPLAY(?E3)                  ! and display the upper cased data
        END
      END
    OF ?Cancel
      CASE EVENT()
      OF EVENT:Accepted
        BREAK
      END
    END
  END
```

PROP:VscrollPos Returns the position of the vertical scroll bar's "thumb" (from 0 to 255) on a window, IMAGE, TEXT, LIST, or COMBO control with the VSCROLL attribute. Setting this property causes the control or window's contents to be scrolled vertically (unless the IMM attribute is on the LIST or COMBO, then only the "thumb" moves).

Example:

```
Que         QUEUE
              STRING(50)
            END
WinView     WINDOW('View'),AT(0,0,320,200),MDI,SYSTEM
              LIST,AT(0,0,320,200),USE(?List),FROM(Que),IMM,VSCROLL
            END
  CODE
  OPEN(WinView)
  Fil:KeyField = 'A' ;  DO BuildListQue
  ACCEPT
    CASE FIELD()
    OF ?List
      CASE EVENT()
      OF EVENT:ScrollDrag
        EXECUTE INT(?List{PROP:VscrollPos}/10) + 1
          Fil:KeyField = 'A'
          Fil:KeyField = 'B'
          Fil:KeyField = 'C'
          Fil:KeyField = 'D'
          Fil:KeyField = 'E'
          Fil:KeyField = 'F'
          Fil:KeyField = 'G'
          Fil:KeyField = 'H'
          Fil:KeyField = 'I'
          Fil:KeyField = 'J'
          Fil:KeyField = 'K'
          Fil:KeyField = 'L'
          Fil:KeyField = 'M'
          Fil:KeyField = 'N'
          Fil:KeyField = 'O'
          Fil:KeyField = 'P'
          Fil:KeyField = 'Q'
          Fil:KeyField = 'R'
          Fil:KeyField = 'S'
          Fil:KeyField = 'T'
          Fil:KeyField = 'U'
          Fil:KeyField = 'V'
          Fil:KeyField = 'W'
          Fil:KeyField = 'X'
          Fil:KeyField = 'Y'
          Fil:KeyField = 'Z'
        END
        DO BuildListQue
  . . .
  FREE(Que)
BuildListQue  ROUTINE
  FREE(Queue)
  SET(Fil:SomeKey,Fil:SomeKey)    !Set to selected key field
  LOOP ?List{PROP:Items} TIMES    !Process number of records visible in list
    NEXT(SomeFile) ;  IF ERRORCODE() THEN BREAK.    !Break at end of file
    Que = Fil:KeyField            !Assign field to display to QUEUE
    ADD(Que)                      ! and add it to the QUEUE
  END
```

PROP:WndProc Sets or gets the window (not the client area) messaging procedure for use with low-level Windows API calls that require it. Generally used with sub-classing to track all Windows messages. (READ-ONLY)

Example:

```
    PROGRAM
    MAP
      main
      SubClassFunc1(USHORT,SHORT,SHORT,LONG),LONG,PASCAL
      SubClassFunc2(USHORT,SHORT,SHORT,LONG),LONG,PASCAL
      MODULE('Windows')                          !TopSpeed Win31 Library
        CallWindowProc(LONG,USHORT,SHORT,SHORT,LONG),LONG,PASCAL
      END
    END
SavedProc1  LONG
SavedProc2  LONG
WM_MOUSEMOVE    EQUATE(0200H)
PT          GROUP,PRE(PT)
X             SHORT
Y             SHORT
            END
  CODE
  Main

Main        PROCEDURE
WinView     WINDOW('View'),AT(0,0,320,200),HVSCROLL,MAX,TIMER(1),STATUS
              STRING('X  Pos'),AT(1,1,,),USE(?String1)
              STRING(@n3),AT(24,1,,),USE(PT:X)
              STRING('Y  Pos'),AT(44,1,,),USE(?String2)
              STRING(@n3),AT(68,1,,),USE(PT:Y)
              BUTTON('Close'),AT(240,180,60,20),USE(?Close)
            END
  CODE
  OPEN(WinView)
  SavedProc1 = WinView{PROP:WndProc}                  !Save this procedure
  WinView{PROP:WndProc} = ADDRESS(SubClassFunc1)      !Change to subclass procedure
  SavedProc2 = WinView{PROP:ClientWndProc}            !Save this procedure
  WinView{PROP:ClientWndProc} = ADDRESS(SubClassFunc2) !Change to subclass proc
  ACCEPT
    CASE ACCEPTED()
    OF ?Close
      BREAK
  . .
SubClassFunc1     FUNCTION(hWnd,wMsg,wParam,lParam)   !Sub class procedure
  CODE                                                ! to track mouse movement in
  IF  wMsg = WM_MOUSEMOVE                             ! window's status bar (only)
    PT:X = MOUSEX()
    PT:Y = MOUSEY()
  END
  RETURN(CallWindowProc(SavedProc1,hWnd,wMsg,wParam,lParam))

SubClassFunc2     FUNCTION(hWnd,wMsg,wParam,lParam)   !Sub class procedure
  CODE                                                ! to track mouse movement in
  IF  wMsg = WM_MOUSEMOVE                             ! window's client area
    PT:X = MOUSEX()
    PT:Y = MOUSEY()
  END
  RETURN(CallWindowProc(SavedProc2,hWnd,wMsg,wParam,lParam))
                                                      !Pass control back to
                                                      ! saved procedure
```

Printer Control Properties

These properties control report and printer behavior. All of these properties can be used with either the PRINTER built-in variable or the label of the report as the *target*, however they may not all make sense with both.

PROPPRINT:Collate
: Specify the printer should collate the output: 0=off, 1=on (not supported by all printers).

PROPPRINT:Color
: Color or monochrome print flag:1=mono, 2=color (not supported by all printers).

PROPPRINT:Context
: Returns the handle to the printer's device context after the first PRINT statement for the report, or an information context before the first PRINT statement. This may not be set for the built-in Global PRINTER variable and is normally only read (not set).

PROPPRINT:Copies
: The number of copies to print (not supported by all printers).

PROPPRINT:Device
: The name of the Printer as it appears in the Windows Printer Dialog. If multiple printer names start with the same characters, the first encountered is used (not case sensitive). May be set for the PRINTER built-in variable only before the report is open.

PROPPRINT:DevMode
: The entire device mode (devmode) structure as defined in the Windows Software Development Kit. This provides direct API access to all printer properties. Consult a Windows API manual before using this.

```
DevMode       GROUP
DeviceName      STRING(32)    !PROPPRINT:Device
SpecVersion     USHORT
DriverVersion   USHORT
Size            USHORT
DriverExtra     USHORT
Fields          ULONG
Orientation     SHORT
PaperSize       SHORT       !PROPPRINT:Paper
PaperLength     SHORT       !PROPPRINT:PaperHeight
PaperWidth      SHORT       !PROPPRINT:PaperWidth
Scale           SHORT       !PROPPRINT:Percent
Copies          SHORT       !PROPPRINT:Copies
DefaultSource   SHORT       !PROPPRINT:PaperBin
PrintQuality    SHORT       !PROPPRINT:Resolution
Color           SHORT       !PROPPRINT:Color
Duplex          SHORT       !PROPPRINT:Duplex
              END
```

PROPPRINT:Driver The printer driver‘s filename (without the .DLL extension).

PROPPRINT:Duplex
The duplex printing mode (not supported by all printers). Equates (DUPLEX::xxx) for the standard choices are listed in the PRNPROP.CLW file.

PROPPRINT:FontMode
The TrueType font mode. Equates (FONTMODE:xxx) for the modes are listed in the PRNPROP.CLW file.

PROPPRINT:FromMin
When set for the built-in PRINTER variable, this forces the value into the “From:” page number in the PRINTERDIALOG. Specify -1 to disable ranges

PROPPRINT:FromPage
The page number on which to start printing. Specify -1 to print from the start.

PROPPRINT:Paper Standard paper size. Equates (PAPER:xxx) for the standard sizes are listed in the PRNPROP.CLW file. This defines the dimensions of the .WMF files that are created by the Clarion runtime library’s “print engine.”

PROPPRINT:PaperBin
The paper source. Equates (PAPERBIN:xxx) for the standard locations are listed in the PRNPROP.CLW file.

PROPPRINT:PaperHeight
The paper height in tenths of millimeters (mm/10). There are 25.4 mm per inch. Used when setting PROPPRINT:Paper to PAPER:Custom (not normally used for laser printers).

PROPPRINT:PaperWidth
The paper width in tenths of millimeters (mm/10). There are 25.4 mm per inch. Used when setting PROPPRINT:Paper to PAPER:Custom (not normally used for laser printers).

PROPPRINT:Percent
The scaling factor used to enlarge or reduce the printed output, in percent (not supported by all printers). This defaults to 100 percent. Set this value to print at the desired percentage (if your printer and driver support scaling). For example, set to 200 to print at double size, or 50 to print at half size.

PROPPRINT:Port Output port name (LPT1, COM1, etc.).

PROPPRINT:PrintToFile
The Print to File flag: 0=off, 1=on.

PROPPRINT:PrintToName
: The output filename when printing to a file.

PROPPRINT:Resolution
: The print resolution in Dots Per Inch (DPI). Equates (RESOLUTION:xxx) for the standard resolutions are listed in the PRNPROP.CLW file.

PROPPRINT:ToMax
: When set for the built-in PRINTER variable, this forces the value into the "To:" page number in the PRINTERDIALOG. Specify -1 to disable ranges

PROPPRINT:ToPage
: The page number on which to end printing. Specify -1 to print to end.

PROPPRINT:Yresolution
: Vertical print resolution in Dots Per Inch (DPI). Equates (RESOLUTION:xxx) for the standard resolutions are listed in the PRNPROP.CLW file.

Example:

```
SomeReport  REPORT
            END

  CODE
  PRINTER{PROPPRINT:Device} = 'Epson'            !Pick 1st Epson in the list

  PRINTER{PROPPRINT:Port} = 'LPT2:'              !Send report to LPT2

  SomeReport{PROPPRINT:Paper} = PAPER:User       !Custom paper size
  SomeReport{PROPPRINT:PAPERHeight} = 6 * 254    !6" form height
  SomeReport{PROPPRINT:PAPERWidth} = 3.5 * 254   !3.5" form width

  PRINTER{PROPPRINT:Percent} = 250               !page printed 2.5 times normal

  PRINTER{PROPPRINT:Copies} = 3                  !print 3 copies of each page
  PRINTER{PROPPRINT:Collate} = False             !print 1,1,1,2,2,2,3,3,3,...
  PRINTER{PROPPRINT:Collate} = True              !print 1,2,3..., 1,2,3...,

  PRINTER{PROPPRINT:PrintToFile} = True          !print to a file
  PRINTER{PROPPRINT:PrintToName} = 'OUTPUT.RPT'  !filename to print to

  OPEN(SomeReport)                          !Open report after setting properties
```

Embedded SQL

Clarion's property syntax can be used to execute SQL statements in your program code by using PROP:SQL naming the file as the *target*. This is only appropriate when using an SQL file driver (such as the ODBC, AS/400, or Oracle drivers).

You may embed any SQL statements supported by the back-end SQL server. If you issue an SQL statement that causes a result set to be returned (such as an SQL SELECT statement), you use NEXT(file) to retrieve the result set (one row at a time) into the file's record buffer. The FILEERRORCODE() and FILEERROR() functions will return any error code and error string set by the back-end SQL server.

You may also query the contents of PROP:SQL to get the last SQL statement issued by the file driver.

Example:

```
SQLFile{PROP:SQL} = 'SELECT field1,field2 FROM table1'       |
                    & 'WHERE field1 > (SELECT max(field1)'  |
                    & 'FROM table2'
                                                !Returns a result set that you
                                                ! get one row at a time using
                                                ! NEXT(SQLFile)

SQLFile{PROP:SQL} = 'CALL GetRowsBetween(2,8)'   !Call a stored procedure

SQLFile{PROP:SQL} = 'CREATE INDEX ON table1 (field1, field2 DESC)"
                                                !No result set

SQLString = SQLFile{PROP:SQL}            !Get last SQL statement issued by driver
```

Run Time Errors

Contents

Trappable Run Time Errors

The following errors can be trapped in code with the ERRORCODE and ERROR functions. Each error has a code number (returned by the ERRORCODE function) and an associated text message (returned by the ERROR function) indicating what the problem is.

2 File Not Found
: The requested file does not exist in the specifed directory.

3 Path Not Found
: The directory name specified as part of the path does not exist.

4 Too Many Open Files
: The total number of file handles available has been used. Check the FILES= setting in the CONFIG.SYS file, or the user's or network's simultaneous open files setting in a network environment.

5 Access Denied
: The file has already been opened by another user for exclusive access, has been left in a locked state, or you do not have network rights to open the file. This error can also occur when no disk space is available.

7 Memory Corrupted
: Some unknown memory corruption has occurred.

8 Insufficient Memory
: There is not enough unallocated memory left to perform the operation. Closing other applications may free up enough memory . With Btrieve, this indicates that you do not have enough real mode memory left to load BTR32.EXE. IN Win95, loading WBTR32.EXE in WINSTART.BAT can avoid this problem.

30 Entry Not Found
: A GET to QUEUE has failed. For GET(Q,*key*), the matching *key* value was not found, and for GET(Q,*pointer*), the *pointer* is out of range.

32 File Is Already Locked
: An attempt to LOCK a file has failed because another user has already locked it.

33 Record Not Available
Usually an attempt to read past the end or beginning of file with NEXT or PREVIOUS. May also be posted by PUT or DELETE when no record was read before the attempted PUT or DELETE.

35 Record Not Found
For a GET(File,*key*), the matching *key* field value was not found.

36 Invalid Data File
Some unknown data file corruption has occurred.

37 File Not Open
An attempt to perform some operation that requires the file be already open has failed because the file is not open.

38 Invalid Key File
Some unknown key file corruption has occurred.

40 Creates Duplicate Key
An attempt to ADD or PUT a record with key field values that duplicate another existing record in the file has been made to a file with a key that does not allow duplicate entires.

43 Record Is Already Held
An attempt to HOLD a record has failed because another user has already held it.

45 Invalid Filename
The filename does not meet the definition of a valid DOS filename.

46 Key File Must Be Rebuilt
Some unknown key corruption has occurred that requires the BUILD statement to re-build the key.

47 Invalid Record Declaration
The data file on disk does not match the file's declaration in the .EXE, usually because you have changed the file's definition in the Data Dictionary and have not yet converted the file to the new format.

48 Unable To Log Transaction
A transaction logout or pre-image file cannot be written to disk. This usually occurs because no disk space is available, or the user does not have the proper network rights.

52 File Already Open
An attempt to OPEN a file that has already been opened by this user.

53 Invalid Clarion File
Indicates a file with a corrupt dBase header. This error only occurs with the xBase drivers.

54 No Create Attribute
An attempt to execute the CREATE procedure on a file whose declaration does not inlcude the CREATE attribute.

56 LOGOUT Already Active
An attempt to issue a second LOGOUT statement while a transaction is already in progress.

57 Invalid Memo File
Some unknown memo file corruption has occurred. For Clarion data files, this could come from a corrupt .MEM file "signature" or pointers to the memo file in the data file that are "out of sync" (usually due to copying files from one location to another and copying the wrong .MEM file).

63 Exclusive Access Required
An attempt to perform a BUILD(file), BUILD(key), EMPTY(file) or PACK(file) was made when the file had not been opened with exclusive access.

64 Sharing Violation
An attempt to perform some action on a file which requires that the file be opened for shared access.

65 Unable To ROLLBACK Transaction
An attempt to ROLLBACK a transaction has failed for some unknown reason.

73 Memo File Missing
An attempt to OPEN a file that has been declared with a MEMO field and the file containing that memo data does not exist.

75 Invalid Field Type Descriptor
Either the type descriptor is corrupt, you have used a *name* that does not exist in GET(Q,*name*), or the file definition is not valid for the file driver. For example, trying to define a LONG field in an xBase file without a matching MEMO field.

76 Invalid Index String
The index *string* passed to BUILD(DynIndex,*string*) was invalid.

77 Unable To Access Index
An attempt to retrieve records using a dynamic index failed because the dynamic index could not be found.

78 Invalid Number Of Parameters
You did not pass the correct number of parameters to a function called in an EVALUATE statement.

79 Unsupported Data Type In File
The file driver has detected a field in the file declared with a data type that is not supported by the file system the driver is designed to access.

80 Unsupported File Driver Function
The file driver has detected a file access statement that is not supported. This is frequently an unsupported form (different parameters) of a statement that is supported.

81 Unknown Error Posted
The file driver has detected some error from the backend file system that it cannot get further information about.

88 Invalid Key Length
An attempt to CREATE a Clarion file driver KEY or INDEX with more than 245 characters.

89 Record Changed By Another Station
The WATCH statement has detected a record on disk that does not match the original version of the record about to be updated in a network situation.

90 File Driver Error
The file driver has detected some other error reported by the file system. You can use the FILEERRORCODE and FILEERROR functions to determine exactly what native error the file system is reporting.

Non-Trappable Run Time Errors

The following errors occur at run time and cannot be trapped with the ERRORCODE or ERROR functions.

Mismatch with CWVBX.DLL detected
: The first CWVBX.DLL file encountered in the path is not the same version (usually an earlier version) than was used to create the .EXE.

VBX control is too complex
: A .VBX control containing more than 64 dialogs (hidden or visible). This limit exists only in 16-bit.

Event posted to a report control
: An attempt to POST an event to a control in a REPORT structure.

Metafile record too large in report
: A .WMF file is too large to print in the report.

Unexpected error opening printer device
: An unexpected error occurred while attempting to open a printer.

Report is already open
: An attempt to OPEN a REPORT that has already been opened and not yet closed.

Unable to open APPLICATION (APPLICATION already active)
: An attempt to OPEN an APPLICATION in a program that has already opened an MDI application frame window.

Unable to open APPLICATION (system is MODAL)
: An attempt to OPEN an APPLICATION in a program that has already opened a MODAL window.

Unable to open APPLICATION
: A failed attempt to OPEN an APPLICATION.

Unable to open WINDOW
: A failed attempt to OPEN a WINDOW.

Unable to open MDI window (No APPLICATION active)
: An attempt to OPEN an MDI WINDOW in a program that has not yet opened an MDI APPLICATION frame window.

Unable to open MDI window (system is MODAL)
: An attempt to OPEN an MDI WINDOW in a program that has already opened a MODAL window.

Unable to open MDI window on APPLICATION's thread
: An attempt to OPEN an MDI WINDOW in the same execution thread as the MDI APPLICATION frame window.

Unable to open MDI WINDOW
: A failed attempt to OPEN an MDI WINDOW.

Too many keystrokes PRESSed
: The parameter to the PRESS statement contains too many characters.

Window is already open
: An attempt to OPEN a WINDOW that is already open.

Window is not open
: An attempt has been made to perform some action that requires a window be opened first. Usually a property assignment statement.

ACCEPT loop requires a window
: An ACCEPT loop that has no associated window.

PRINT must only be called for reports
: An attempt to PRINT a structure that is not part of a REPORT.

ENDPAGE must only be called for reports
: An attempt to execute the ENDPAGE statement when no REPORT is active.

Unable to process ACCEPT (system is MODAL)
: An attempt to perform an illegal action in a program that has already opened a MODAL window.

Unable to create control (system is MODAL)
: An attempt to CREATE a control in a program that has already opened a MODAL window.

Unable to complete operation (system is MODAL)
: An attempt to perform an illegal action in a program that has already opened a MODAL window.

Compiler Errors

The compiler generates an error message at exactly the point in the source code where it determines that something has gone wrong. Therefore, the problem is always either right at that point, or somewhere in the code preceding that point. For most error messages, the problem exists right at the point at which it is detected, but some error messages are typically generated by problems that far precede their detection by the compiler, making some "detective work" necessary, along with an understanding of what the compiler is trying to tell you in the error message itself.

Deciphering compiler error messages to determine exactly what syntax error needs to be corrected can be a bit of an arcane science. The major reason for this is that a single (relatively minor) error can create a "cascade effect;" a long list of error messages that all have one root cause. This is typically the case in the situation where there are a very large number of compiler errors reported in the same source module. To handle this, you should correct just the first error reported then re-compile to see how many errors are left (quite often, none). If you have just a couple of errors reported that are widely separated in the source code, it is likely that each is a discrete error and you should correct them all before re-compiling.

Specific Errors

The following error messages occur when the compiler has detected a specific syntax problem and is attempting to alert you to exactly what the problem is so that you may correct it.

Some of the following error messages contain a "%V" token. The compiler substitutes an explicit label indicating what problem is occurring for this token when it generates the error message, which should help point to the cause of the error.

! introduces a comment
: This is a common C programmer's error. If you type IF A != 1 THEN you get this warning.

Actual value parameter cannot be array
: The passed parameter must not be an array.

ADDRESS parameter ambiguous
: ADDRESS(*MyLabel*) where *MyLabel* is the label of both a procedure and a data item.

All fields must be declared before JOINs
: All PROJECT statements for the file must precede any JOIN statements in the VIEW structure.

Ambiguous label
The field qualification syntax has come up with more that one solution for the label you have supplied. For example:

```
G   GROUP
S:T   SHORT     !Referenced as G:S:T
    END
G:S GROUP
T     SHORT     !Referenced as G:S:T
    END
  CODE
  G:S:T = 7     !Which are you talking about?
```

Array too big
Arrays are limited to 64K in 16 bit.

Attribute parameter must be QUEUE, QUEUE field or constant string
The parameter must be the label of a previously declared QUEUE structure, a field within a QUEUE structure, or a string constant.

Attribute requires more parameters
You must pass all required parameters to an attribute that takes parameters.

Attribute string must be constant
The parameter must be a string constant, not the label of a variable.

Attribute variable must be global
The parameter must be a variable declared in the PROGRAM module as global data.

Attribute variable must have string type
The parameter must be a variable declared as a STRING, CSTRING, or PSTRING.

BREAK statement must be within LOOP
BREAK is only valid within a LOOP or ACCEPT structure.

BREAK structure must enclose DETAIL
There must be at least one DETAIL structure within nested BREAK structures (at the lowest level).

Calling function as procedure
A Warning that a FUNCTION is being called as a PROCEDURE and the return value will be lost.

Cannot call procedure as function
You can call a FUNCTION as a PROCEDURE, but you cannot call a PROCEDURE as a FUNCTION.

Cannot declare KEY in a VIEW
A KEY declaration is not valid in a VIEW structure.

Cannot EXIT from here
Only a ROUTINE may contain the EXIT statement.

Cannot GOTO into ROUTINE
: The target of GOTO must be the label of an executable code statement within the same procedure or ROUTINE, and may not be the label of a ROUTINE.

Cannot have default parameter here
: You may only have a default value on non-omittable integer data type parameters passed by value.

Cannot have initial values with OVER
: A variable declaration with the OVER attribute may not also have an initial value parameter.

Cannot have statement here
: This happens if the compiler thinks you have tried to define a code label inside the global data section.

Cannot initialize variable reference
: A reference variable cannot have an initial value.

Cannot return CSTRING from CLARION function
: CSTRING is not a valid return data type for a FUNCTION written in Clarion (only for functions written in other languages).

Cannot RETURN value from procedure
: Only a FUNCTION may contain the RETURN statement with a return value parameter.

CLARION function cannot use RAW or NAME
: These attributes are not appropriate for a PROCEDURE or a FUNCTION written in Clarion (only for functions written in other languages).

CYCLE statement must be within LOOP
: CYCLE is only valid within a LOOP or ACCEPT structure.

DECIMAL has too many places
: A DECIMAL or PDECIMAL declaration may only have a maximum of 30 decimal places, and the decimal portion must be less than the length.

DECIMAL too long
: A DECIMAL or PDECIMAL declaration may have a maximum length of 31 digits.

Declaration not valid in FILE structure
: This data declaration may not be contained within a FILE structure.

Declaration too big
: The compiler has detecteda PSTRING > 255 or MEMO > 64K in 16 bit, etc.

DLL attribute requires EXTERNAL attribute
: The DLL attribute further defines the EXTERNAL attribute.

Duplicate code label: %V
: Two lines of executable source code have the same named label.

Dynamic INDEX must be empty
: An attempt to use the 2 parameter form of BUILD on a KEY or INDEX declared with component fields..

Embedded OVER must name field in same structure
: The parameter to the OVER attribute must be the label of a previously declared variable in the same structure.

ENCRYPT attribute requires OWNER
: The ENCRYPT attribute and OWNER attribute function together.

Entity-parameter cannot be an array
: You cannot pass an array of entity parameters (FILE, QUEUE, etc.).

Expected a PROJECT statement
: A VIEW structure must contain at least one PROJECT statement.

Expected: %V
: This is one of the most common errors. The compiler was expecting to find something (one of the items in the list substituted for the %V token) as the next code to compile, but instead found the code at the point in the source that the error is generated.

Expression cannot be picture
: You have attempted to use an EQUATE label to a picture token in a place where a picture token is not valid.

Expression cannot have conditional type
: An expression is not a numeric value. For example, MyValue = A > B is invalid.

Expression must be constant
: Variables are not valid in this expression.

Field equate label not defined: %V
: The named field equate label has not been previously declared.

Field not found in parent FILE
: A JOIN statement must declare all the linking fields between the parent and child files.

Field requires (more) subscripts
: This is referencing an array with multiple dimensions, and you must supply an index into each dimension.

FILE must have DRIVER attribute
: The DRIVER attribute is required to declare the file system for whcih the data file is formatted.

FILE must have RECORD structure
: It is invalid to declare a FILE which does not contain a RECORD structure.

FILEs must have same DRIVER attribute
: All files named in a LOGOUT statement must use the same file system.

FUNCTION must have return type
: If you declare a prototype with a return data type in the MAP, you must create it as a FUNCTION.

Function result is not of correct type
: The RETURN statement must return a value consistent with the return data type prototyped in the MAP structure.

Group too big
: GROUPs are limited to 64K in 16 bit.

Ignoring EQUATE redefinition: %V
: A Warning that the named equate is being ignored. This is really a label-redefined error except that the definition is not thrown away.

Illegal array assignment
: An assignment to an array must reference a single element, not the entire array.

Illegal character
: A non-valid lexical token. For example, an ASCII 255 in your source.

Illegal data type: %V
: The named data type is inappropriate for the structure in which it is placed.

Illegal key component
: A KEY has any type of illegal component.

Illegal nesting of window controls
: Window controls other than RADIO have been placed within an OPTION structure, or controls other than TAB have been placed directly within a SHEET structure.

Illegal parameter for LIKE
: An illegal parameter to a LIKE declaration. For example, LIKE(7).

Illegal parameter type for STRING
: An illegal parameter to a STRING declaration. For example, STRING(MyVar).

Illegal reference assignment
: A reference variable may only be assigned another reference variable of the same type, or the label of a variable of the type it references.

Illegal return type or attribute
: The prototype contains an invalid data type as the return data type (such as *CSTRING).

Illegal target for DO
: The target of DO must be the label of a ROUTINE.

Illegal target for GOTO
: The target of GOTO must be the label of an executable code statement within the same procedure or ROUTINE, and may not be the label of a ROUTINE.

INCLUDE invalid, expected: %V
: The INCLUDE statement's parameter must be a well formed Clarion string. In particular, type conversion is not valid, so INCLUDE('MyFile'&MyValue) is invalid.

INCLUDE misplaced
: INCLUDE has to follow a line-break, or a semi-colon (possibly followed by white space).

INCLUDE nested too deep
: You can only nest INCLUDEs 3 deep. In other words you can INCLUDE a file that INCLUDEs a file that INCLUDEs a file, but the last file must not INCLUDE anything.

Incompatible assignment types
: An attempt to assign between incompatible data types.

Incorrect procedure profile
: An attempt to pass a procedure with the wrong prototype as a procedural parameter.

Indices must be constant
: An attempt has been made to have a USE variable that is an array element with variable indices.

Integer expression expected
: The expression must evaluate to an integer.

Invalid data declaration attribute
: An attribute that is inappropriate on the data declaration.

Invalid data type for value parameter
: The data type prototyped in the MAP may not be passed by value and must be passed by address. For example, to pass a CSTRING parameter to a Clarion procedure, it may only be prototyped as *CSTRING.

Invalid FILE attribute
: An attribute that is inappropriate on a FILE declaration.

Invalid first parameter of ADD
: The statement's first parameter is not appropriate.

Invalid first parameter of DELETE
: The statement's first parameter is not appropriate.

Invalid first parameter of FREE
: The statement's first parameter is not appropriate.

Invalid first parameter of NEXT
: The statement's first parameter is not appropriate.

Invalid first parameter of POSITION
: The statement's first parameter is not appropriate.

Invalid first parameter of PREVIOUS
: The statement's first parameter is not appropriate.

Invalid first parameter of PUT
: The statement's first parameter is not appropriate.

Invalid first parameter of SET
: The statement's first parameter is not appropriate.

Invalid GROUP/QUEUE/RECORD attribute
: An attribute that is inappropriate on a GROUP, QUEUE, or RECORD declaration.

Invalid KEY/INDEX attribute
: An attribute that is inappropriate on a KEY or INDEX declaration.

Invalid label
: A label that contains characters other than letters, numbers, underscore (_), or colon (:), or does not start with a letter or underscore.

Invalid LOOP variable
: An attempt to use an illegal data type (DATE, TIME, STRING, etc.) as a LOOP variable.

Invalid MEMBER statement
: The parameter to the MEMBER statement is not a string constant or does not reference the PROGRAM module for the current project.

Invalid number
: A number is required, for example inside the repeat character notation ({}) in a string constant.

Invalid OMIT expression
: The parameter to the OMIT statement is invalid.

Invalid parameters for attribute
: You must pass valid parameters to an attribute that takes them.

Invalid picture token
: A picture token that contains inappropriate characters.

Invalid printer control token
: A PRINT statement containing a printer control token other than @CR, @LF, or @FF.

Invalid QUEUE/RECORD attribute
: An attribute that is inappropriate on a QUEUE or RECORD declaration.

Invalid SIZE parameter
: SIZE(Junk+SomeMoreJunk)

Invalid string (misused <...> or {...})
: A string constant contains a single beginning bracket (< or {) without a matching terminating bracket (> or }). These characters must have two together (<< or {{) if intended to be part of the string.

Invalid structure as first parameter
: The statement's first parameter is not appropriate.

Invalid structure within property syntax
: A structure that is inappropriate in a proeprty assignment statement.

Invalid USE attribute parameter
: The parameter is not appropriate for a USE attribute.

Invalid variable data parameter type
: When passing parameters by address, you must pass the same data type as prototyped in the MAP structure.

Invalid WINDOW control
: A control that is inappropriate in a WINDOW structure.

ISL error: %V
: Contact Technical Support and provide all details of the error message.

KEY must have components
: You cannot declare a KEY without naming the component fields that establish the KEY's sort order.

Label duplicated, both removed: %V
: The named field equate label is used multiple times within the same module and has been removed from the list of equate labels that may be used within the executable code. Correctable with the third parameter to the USE attribute.

Label not defined: %V
: The named label has not been previously declared.

Mis-placed string slice operator
: A string slice that is not the last array index. For example, MyStringArray[3:4,5].

Missing procedure definition: %V
: The named procedure is not prototyped in a MAP structure.

Must be dimensioned variable
: This must be an array.

Must be field of a FILE or VIEW
: Must be a field that is a member of a FILE or VIEW structure. For example NULL(LocalVariable) with give this error.

Must be variable
: This must be the label of a previously declared variable.

Must have constant string parameter
: The parameter must be a string constant, not the label of a variable.

Must have one field for each key component
: A JOIN statement must declare all the linking fields between the parent and child files.

Must RETURN value from function
: A FUNCTION must contain the RETURN statement with a return value parameter.

Must specify DECIMAL size
: A DECIMAL or PDECIMAL declaration must declare the maximum number of digits it stores.

Must specify identifier
: An indentifier was required but not supplied.

Must specify print-structure
: A PRINT statement may only print a structure in a REPORT.

No prototype available
: All PROCEDUREs and FUNCTIONs must be prototyped in a MAP structure.

Not valid inside structure
: A data type is inappropriate for the structure in which it is placed.

OMIT cannot be nested
: You are in an OMIT (or COMPILE) that is *not* omitting code and the compiler encounter another OMIT.

OMIT misplaced
: OMIT has to follow a line-break, or a semi-colon (possibly followed by white space).

OMIT not terminated: %V
: The referenced OMIT parameter was not found before the end of the source module.

Order is MENUBAR, TOOLBAR, Controls
: The MENUBAR structure must come before the TOOLBAR, and the TOOLBAR structure must come before the controls in a WINDOW or APPLICATION.

OVER must name variable
: The parameter to the OVER attribute must be the label of a previously declared variable.

OVER must not be larger than target variable
: The parameter to the OVER attribute must be the label of a previously declared variable that is greater than or equal to the size of the variable being declared OVER it.

OVER not allowed with STATIC or THREAD
: A variable declaration with the OVER attribute may not also have the STATIC or THREAD attribute (these must be on the initial declaration).

Parameter cannot be omitted
: The procedure or function call must pass all parameters that have not been prototyped as omittable parameters.

Parameter cannot be omitted
: The procedure or function call must pass all parameters that have not been prototyped as omittable parameters.

Parameter kind does not match
: When passing parameters by address, you must pass the same data type as prototyped in the MAP structure.

Parameter must be picture
: This must be a display picture token.

Parameter must be procedure label
: This must be the label of a procedure.

Parameter must be report DETAIL label
: A PRINT statement may only print a structure in a REPORT.

Parameter type label ambiguous (CODE or DATA)
: You many have a PROCEDURE and data declaration with the same name, but then you cannot use that name in a procedure prototype.

Parameter type must be GROUP or QUEUE
: The passed parameter must be a GROUP or QUEUE structure.

PROCEDURE cannot have return type
: If you declare a prototype without a return data type in the MAP, you must create it as a PROCEDURE.

Procedure doesn't belong to module: %V
: An attempt to define a procedure that has a prototype that says it belongs in another module.

Redefining system intrinsic: %V
: A Warning that the named procedure (part of your source code) has the same name as a Clarion run time library procedure or function and that your procedure or function will be called instead of the built-in library's.

Routine label duplicated
: The label of a ROUTINE statement has been previously used on another statement.

Routine not defined: %V
: The named ROUTINE does not exist.

SECTION duplicated: %V
: The named SECTION exists twice in the INCLUDE file.

SECTION not found: %V
: The named SECTION does not exist in the INCLUDE file.

Statement label duplicated
: Two lines of executable source code have the same label.

Statement must have label
: The statement (such as a ROUTINE or PROCEDURE statement) must have a label.

String not terminated
: A string constant without a terminating single quote (').

Subscript out of range
: An attempt to reference an array element beyond the valid number of elements dimensioned in the data declaration.

Too few indices
: This is referencing an array with multiple dimensions, and you must supply an index into each dimension.

Too few parameters
: The procedure or function call must pass all parameters that have not been prototyped as omittable parameters.

Too few parameters
: The procedure or function call must pass all parameters that have not been prototyped as omittable parameters.

Too many indices
: This is referencing an array and you are supplying too many indexes into the dimensions.

Too many parameters
: The procedure or function call may noy pass more parameters than have been prototyped.

Unable to verify validity of OVER attribute
: A Warning that you are declaring a variable OVER a passed parameter and the data types may not match at run time.

Unknown attribute: %V
: The named attribute is not part of the Clarion language.

Unknown function label
: The FUNCTION has not been previously prototyped in the program's MAP structure.

Unknown identifier
: The label has not been previously declared.

Unknown identifier: %V
: The named identifier has not been previously declared.

Unknown key component: %V
: The named key component does not exist within the FILE structure.

Unknown procedure label
: The PROCEDURE has not been previously prototyped in the program's MAP structure.

Value requires (more) subscripts
: This is referencing an array with multiple dimensions, and you must supply an index into each dimension.

Variable expected
: This must be the label of a previously declared variable.

Variable-size must be constant
: The variable declaration must contain a constant expresion for its size parameter.

VIEW must have a FILE as parameter
: A VIEW structure must declare the primary file from which it derives data.

Wrong number of parameters
: The procedure or function call must pass all parameters that have not been prototyped as omittable parameters.

Wrong number of parameters
: The procedure or function call must pass all parameters that have not been prototyped as omittable parameters.

Wrong number of subscripts
: An attempt to access a multi-dimensioned array without providing an element number for each dimension. For example:

```
MyShort  SHORT,DIM(8,2)
  CODE
  MyValue = MyShort[7]  !Wrong number of subscripts
```

Unknown errors

All of the error messages listed below are errors where the compiler has tried to give *the compiler writer* some clue as to what is wrong. Report the problem immediately to TopSpeed together with the source file that generated the error.

Inconsistent scanner initialization

Unknown operator

Unknown expression type

Unknown expression kind

Unknown variable context

Unknown parameter kind

Unknown assignment operator

Unknown variable type

Unknown case type

Unknown equate type

Unknown string kind

Unknown picture type

Unknown descriptor type

Unknown initializer type

Unknown designator kind

Unknown structure field

Unknown formal entity

Type descriptor not static

Unknown clear type

Unkonwn simple formal type

Out of attribute space

Unknown label/routine

Unknown special identifier

Value not static

Unknown static label

Unknown screen structure kind

Corrupt pragma string

Old symbol non-NIL

Events

Contents

In Clarion Windows programs, most of the messages from Windows are automatically handled internally by the ACCEPT event processor. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by ACCEPT to your Clarion code. The net effect of this is to make your programming job easier by removing the low-level "drudgery" code from your program, allowing you to concentrate on the high-level aspects of programming, instead. Of course, it is also possible to handle these low-level messages yourself by "sub-classing" the window, but that is a low-level technique that should only be used if absolutely necessary. Consult Charles Petzold's book *Programming Windows* published by Microsoft Press if you need more information on sub-classing.

There are two types of events passed on to the program by ACCEPT: **Field-specific** and **Field-independent** events. The following lists are the event EQUATEs that are contained in EQUATES.CLW.

Field-Independent Events

A **Field-independent** event does not relate to any one control but requires some program action (for example, to close a window, quit the program, or change execution threads). Most of these events cause the system to become modal for the period during which they are processing, since they require a response before the program may continue.

EVENT:PreAlertKey
: The user pressed an ALRT attribute hot key for an ALRT attribute on the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:AlertKey is not generated and the action is aborted.

EVENT:AlertKey
: The user pressed an ALRT attribute hot key for an ALRT attribute on the window. This is the event on which you perform the action the user has requested by pressing the alert key.

EVENT:CloseWindow
: The window is closing. POSTing this event closes the window. This is the event on which you perform any window cleanup code.

EVENT:CloseDown
: The application is closing. POSTing this event closes the application. This is the event on which you perform any application cleanup code.

EVENT:OpenWindow
: The window is opening. This is the event on which you perform any window initialization code.

EVENT:LoseFocus
: The window is losing input focus to another thread. This is the event on which you save any data that could be at risk of being changed by another thread.

EVENT:GainFocus
: The window is gaining input focus from another thread. This is the event on which you restore any data you saved in EVENT:LoseFocus.

EVENT:Suspend
: The window still has input focus but is giving control to another thread to process timer events.

EVENT:Resume
: The window still has input focus and is regaining control from an EVENT:Suspend.

EVENT:Timer
: The TIMER attribute has triggered. This is the event on which you perform any timed actions, such as clock display, or background record processing for reports or batch processes.

EVENT:Move
: The user is moving the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Moved is not generated and the action is aborted. This is the event on which you can prevent users from moving a window.

EVENT:Moved
: The user has moved the window. This is the event on which you readjust anything that is screen-position-dependent.

EVENT:Size
: The user is resizing the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Sized is not generated and the action is aborted. This is the event on which you can prevent users from resizing a window.

EVENT:Sized
: The user has resized the window. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Restore
: The user is restoring the window's previous size. If a CYCLE statement is encountered in the code to process this event, the EVENT:Restored is not generated and the action is aborted. This is the event on which you can prevent users from restorng a window.

EVENT:Restored
: The user has restored the window's previous size. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Maximize The user is maximizing the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Maximized is not generated and the action is aborted. This is the event on which you can prevent users from maximizing a window.

EVENT:Maximized The user has maximized the window. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Iconize The user is minimizing the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Iconized is not generated and the action is aborted. This is the event on which you can prevent users from minimizing a window.

EVENT:Iconized The user has minimized the window. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Completed AcceptAll (non-stop) mode has finished processing all the window's controls. This is the event on which you have executed all data entry validation code for the controls in the window and can safely write to disk.

EVENT:DDErequest
A client has requested a data item from this Clarion DDE server application. This is the event on which you execute DDEWRITE to provide the data to the client once.

EVENT:DDEadvise
A client has requested continuous updates of a data item from this Clarion DDE server application. This is the event on which you execute DDEWRITE to provide the data to the client every time it changes.

EVENT:DDEexecute
A client has sent a command to this Clarion DDE server application (if the client is another Clarion application, it has executed a DDEEXECUTE statement). This is the event on which you determine the action the client has requested and perform it, then execute a CYCLE statement to signal positive acknowledgement to the client that sent the command.

EVENT:DDEpoke A client has sent unsolicited data to this Clarion DDE server application. This is the event on which you determine what the client has sent and where to place it, then execute a CYCLE statement to signal positive acknowledgement to the client that sent the data.

EVENT:DDEdata A DDE server has supplied an updated data item to this Clarion client application.

EVENT:DDEclose A DDE server has terminated the DDE link to this Clarion client application.

Field-Specific Events

A **Field-specific** event occurs when the user presses a key that may require the program to perform a specific action related to that control.

EVENT:Selected The control has received input focus. This is the event on which you should perform any data initialization code.

EVENT:Accepted The user has entered data or made a selection then pressed TAB or CLICKED the mouse to move on to another control. This is the event on which you should perform any data input validation code.

EVENT:Rejected The user has entered an invalid value for the entry picture, or an out-of-range number on a SPIN control. The REJECTCODE function returns the reason the user's input has been rejected and you can use the PROP:ScreenText property to get the user's input from the screen. This is the event on which you alert the user to the exact problem with their input.

EVENT:PreAlertKey The user pressed an ALRT attribute hot key for an ALRT attribute on the control. If a CYCLE statement is encountered in the code to process this event, the EVENT:AlertKey is not generated and the action is aborted.

EVENT:AlertKey The user pressed an ALRT attribute hot key for an ALRT attribute on the control. This is the event on which you perform the action the user has requested by pressing the alert key.

EVENT:Dragging The user is dragging the mouse from a control with the DRAGID attribute and the mouse cursor is over a valid potential drop target. This event is posted to the control from which the user is dragging. This is the event on which you can change the mouse cursor to indicate a valid drop target.

EVENT:Drag The user released the mouse button over a valid drop target. This event is posted to the control from which the user is dragging. This is the event on which you set the program to pass the dragged data to the drop target.

EVENT:Drop The user released the mouse button over a valid drop target. This event is posted to the drop target control. This is the event on which you receive the dragged data.

EVENT:NewSelection
: The current selection in the LIST or COMBO control has changed (the highlight bar has moved up or down). This is the event on which you perform any "housekeeping" to synchronize other controls with the currently highlighted record in the list.

EVENT:ScrollUp
: On a LIST or COMBO control with the IMM attribute, the user has attempted to move the highlight bar off the top of the LIST. This is the event on which you get a previous record when "page-loading" the list.

EVENT:ScrollDown
: On a LIST or COMBO control with the IMM attribute, the user has attempted to move the highlight bar off the bottom of the LIST. This is the event on which you get the next record when "page-loading" the list.

EVENT:PageUp
: On a LIST or COMBO control with the IMM attribute, the user pressed PGUP. This is the event on which you get the previous page of records when "page-loading" the list.

EVENT:PageDown
: On a LIST or COMBO control with the IMM attribute, the user pressed PGDN. This is the event on which you get the next page of records when "page-loading" the list.

EVENT:ScrollTop
: On a LIST or COMBO control with the IMM attribute, the user pressed CTRL+PGUP. This is the event on which you get the first page of records when "page-loading" the list.

EVENT:ScrollBottom
: On a LIST or COMBO control with the IMM attribute, the user pressed CTRL+PGDN. This is the event on which you get the last page of records when "page-loading" the list.

EVENT:Locate
: On a LIST control with the VCR attribute, the user pressed the locator (?) VCR button. This is the event on which you can unhide the locator entry control, if it is kept hidden.

EVENT:DroppingDown
: On a LIST or COMBO control with the DROP attribute, the user pressed the down arrow button. This is the event on which you get the records when "demand-loading" the list.

EVENT:DroppedDown
: On a LIST or COMBO control with the DROP attribute, the list has dropped. This is the event on which you can hide other controls that the droplist covers to prevent "screen clutter" from distracting the user.

EVENT:VBXevent
: On a CUSTOM control, a VBX-specific event occurred. This is the event on which you query the PROP:VBXEvent and PROP:VBXEventArg properties to determine what event occurred and its parameters.

EVENT:Expanding
: On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box. If a CYCLE statement is encountered in the code to process this event, the EVENT:Expanded is not generated and the expansion is aborted.

EVENT:Expanded
: On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box.

EVENT:Contracting
: On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree contraction box. If a CYCLE statement is encountered in the code to process this event, the EVENT:Contracted is not generated and the contraction is aborted.

EVENT:Contracted
: On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box.

EVENT:MouseIn
: On a REGION with the IMM attribute, the mouse cursor has entered the region.

EVENT:MouseOut
: On a REGION with the IMM attribute, the mouse cursor has left the region.

EVENT:MouseMove
: On a REGION with the IMM attribute, the mouse cursor has moved within the region.

EVENT:TabChanging
: On a SHEET control, focus is passing to another tab. This is the event on which you perform any necessary "housekeeping" code.

Index

Contents

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y