CLARION
DATABASE DEVELOPER®
for WINDOWS®

BETA-1 RELEASE
LANGUAGE REFERENCE

CLARION SOFTWARE CORPORATION

Clarion Software Corporation
150 East Sample Road
Pompano Beach, Florida 33064
(305) 785-4555

Printed in the United States of America (1193)

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*). 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 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:

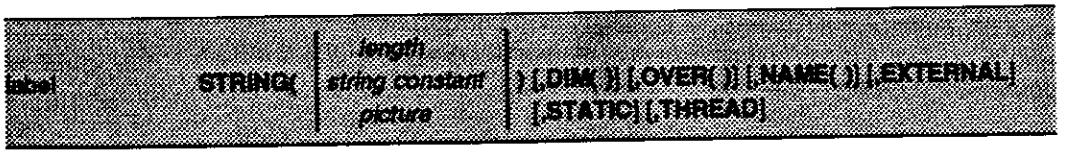
```
              PROGRAM
              MAP
                NewThread                       !A procedure
                FullName(STRING,STRING,STRING).STRING
                                                !Function prototype with parameters
                DayString.STRING                !Function prototype without parameters
              END
  CODE
    :                                           !Global executable statements
    :
  START(NewThread)                              !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                                           !End if
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                                           !END execute structure
```

See Also: FUNCTION and PROCEDURE Prototypes

Variable Declaration Statements

STRING (fixed-length string)



Format: A fixed number of bytes.

Range: 1 to 65,535 bytes.

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*. String variables are not initialized unless given a *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. 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.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Must be used with the STATIC attribute on Procedure Local data.

STRING declares a fixed-length character string.

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.

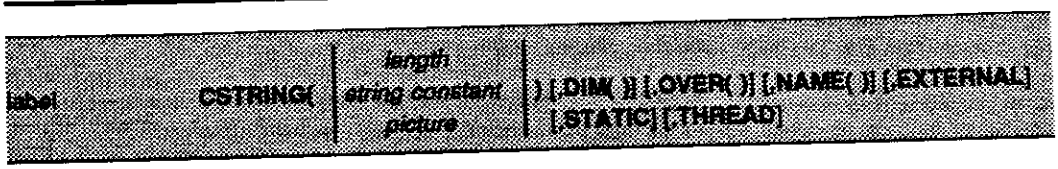
To take a "slice" of the STRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the STRING. The position numbers may be integer constants, variables, or expressions. If variables are used, 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)



Format: A fixed number of bytes.

Range: 1 to 65,535 bytes.

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 strin g is set to the length of the *string constant* plus the terminating null character. String variables are not initialized unless given a *string constant*.

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. 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.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Must be used with the STATIC attribute on Procedure Local data.

CSTRING declares a character string terminated by a null character (ASCII zero). It matches the 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 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 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.

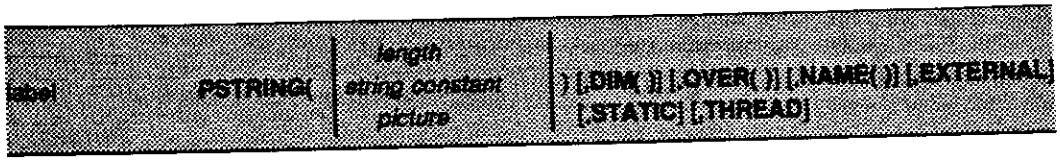
To take a "slice" of the CSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the CSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, 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).

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
```



Format: A fixed number of bytes.

Range: 1 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 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 strin g is set to the length of the *string constant* plus the length-byte. String variables are not initialized unless given a *string constant*.

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.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Must be used with the STATIC attribute on Procedure Local data.

PSTRING declares a character string with a leading length byte included in the number of bytes declared for the string. It matches the 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.

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 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
```

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. However, if the passed parameter is a *STRING or *GROUP, the compiler will issue a warning because it cannot determine at compile time whether the *overvariable* is as large as the variable declared with the OVER attribute.

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

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 Procedure Local data with the STATIC attribute. It may also be declared as Global data or Member Local data.

This attribute creates a lot of runtime "overhead," particularly on Global or Member Local data. Therefore, it should be used only when absolutely necessary.

Example:

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

See Also: Data Declarations and Memory Allocation

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

Member Local 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. Member Local data is also allocated Static memory.

Procedure 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. Procedure Local data is allocated Dynamic memory on the program's stack. This can be overridden by using the STATIC attribute, making its value "persistent" between calls to the procedure.

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

See Also: FUNCTION and PROCEDURE Prototypes, STATIC

Runtime Dynamic Expressions

Clarion Database Developer 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 construct expressions "on the fly." This also makes it possible to allow an end-user to enter the expression to evaluate.

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

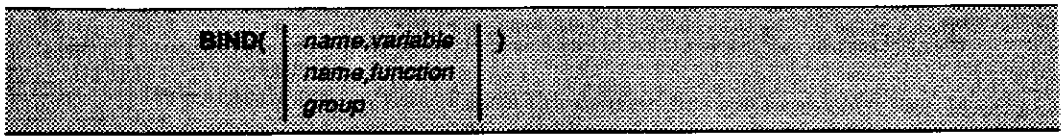
All of the standard Clarion expression syntax is available for use in dynamic expressions. This includes parenthetical grouping and all the arithmetic, logical, and string operators. Dynamic expressions are evaluated just as any other Clarion expression and all the standard operator precedence level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

It takes three steps to use dynamic expressions:

- 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 may be used just as the result of any hard-coded expression would be. For example, a dynamic expression could provide a filter expression to eliminate certain records when viewing or printing a database.

BIND (declare dynamic expression variable)



BIND Identifies variables allowed to be used in dynamic expressions.

name A string constant containing the identifier used in the dynamic expression.

variable The label of any variable 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 (FILE), or QUEUE structure declared with the BINDABLE attribute. *** NOT YET IMPLEMENTED.

The **BIND** statement declares logical names used to identify variables or user-defined functions in dynamic expressions.

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 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, FILE (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:

See Also: UNBIND, EVALUATE

UNBIND(*name*)

UNBIND Frees variables from use in dynamic expressions.

name A string constant that specifies the identifier used by the dynamic expression evaluator.

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 dynamic expressions.

*** NOT YET IMPLEMENTED.

Example:

See Also: BIND, EVALUATE

EVALUATE (return dynamic expression result)

EVALUATE(*expression*)

EVALUATE Evaluates runtime dynamic expressions.

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

The **EVALUATE** function returns the result of the dynamic *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: *** NO ERRORS YET IMPLEMENTED.

Example:

See Also: BIND, UNBIND

Assignment Statements

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.

A deep assignment statement usually 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,PRE(GOne)
S               SHORT
L               LONG
              END
G2            GROUP,PRE(GTwo)
T               LONG
S               REAL
L               SHORT
              END
C             SHORT,DIM(1000)
  CODE
  G2 :=: G1           !Is equivalent to:      G2:S = G1:S
                      !                       G2:L = G1:L
  C :=: 7             !Is equivalent to:      LOOP I# = 1 to 1000
                      !                         C[I#] = 7
                      !                       END
```

Clarion Windows

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, and 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 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 what will be used throughout this book.

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 File menu's Open selection is a 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.

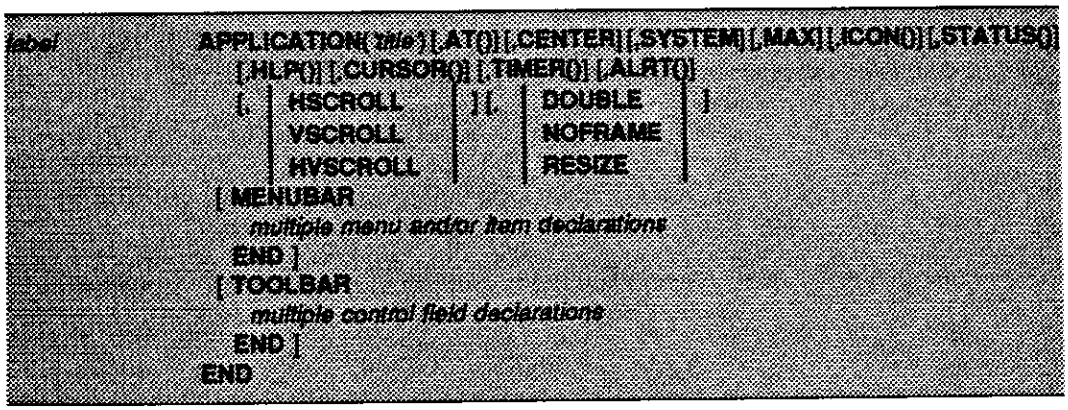
Field Equate Labels always begin with a question mark (?) followed by the name of the control's USE variable. The leading question mark indicates to the compiler a Field Equate Label. They are very similar to normal EQUATE compiler directives. The compiler substitutes the field number for the Field Equate Label at compile time. This makes it unnecessary to know field numbers in advance.

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.

APPLICATION (declare an MDI frame window)



APPLICATION Declares a Multiple Document Interface (MDI) frame window.

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.

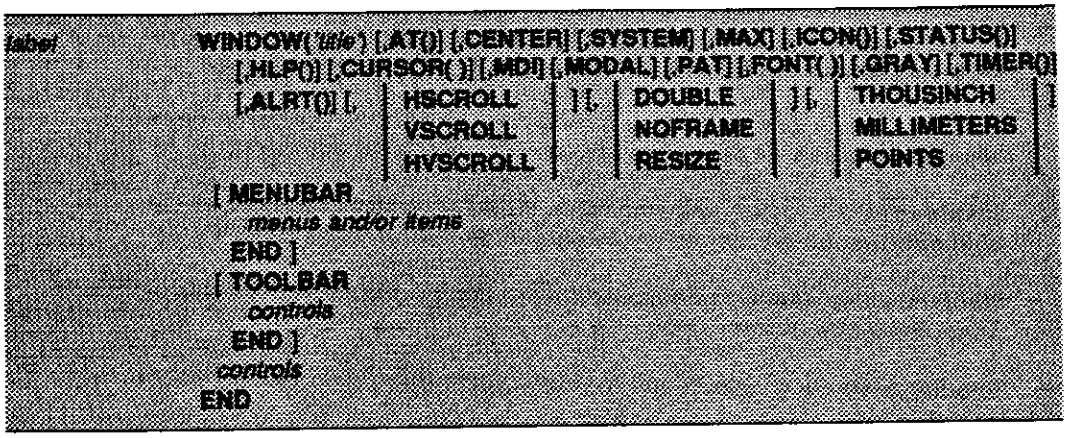
TIMER	Specifies periodic timed event generation.
ALRT	Specifies "hot" keys active for the entire APPLICATION.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the application frame when any portion of a child window lies horizontally outside the visible area.
VSCROLL	Specifies 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 the APPLICATION is opened. 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.

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 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 built-in procedures.

WINDOW (declare a dialog window)



WINDOW Declares a document window or dialog box.

label A valid Clarion label. A *label* is required on the WINDOW statement.

title A string constant or variable containing the title text for the window.

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 be displayed when the mouse is positioned over the window. This cursor is inherited by the WINDOW's controls unless it is 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.

PAT	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.
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.
THOUSINCH	Specifies thousandths of an inch as the measurement unit used for all attributes which use screen coordinates.
MILLIMETERS	Specifies millimeters as the measurement unit used for all attributes which use screen coordinates.
POINTS	Specifies points as the measurement unit used for all attributes which use screen coordinates. There are 72 points per inch, vertically and horizontally.
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. This may or may not be 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.

When the WINDOW is first opened, it remains hidden until the first 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 built-in procedures. Any previously opened WINDOW on the same 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 unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attributes. 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 program running in the background will halt 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, 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.

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.

ALRT (set "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, a field-independent event is generated (both the ACCEPTED and SELECTED functions return zero). You may have multiple ALRT attributes on one APPLICATION or WINDOW.

Example:

```
Screen      WINDOW,AT(6,40),PRE(Scr),ALRT(F10Key)          !F10 alerted for all fields
            END
            CODE
            OPEN(Screen)                                   !Open screen for processing
            ACCEPT                                         !  and process all fields
              IF KEYCODE() = F10Key                        !Check for the WINDOW's "hot" key
                RETURN
              END
            END
```

AT[(x] [,y] [,*width*] [,*height*])

AT Specifies the initial position and size of an APPLICATION or 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 place on an APPLICATION or WINDOW without the MDI attribute. They are relative to the top left hand corner of the APPLICATION when the AT attribute is place on a WINDOW with the MDI attribute.

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 default to dialog units unless the THOUSINCH, MILLIMETERS, 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 window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

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.

CURSOR (set mouse cursor type)

CURSOR(*file*)

CURSOR Specifies a mouse cursor to display for the window.

file A string constant or variable containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor.

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

DOUBLE, NOFRAME, RESIZE (set window border)

```
DOUBLE
NOFRAME
RESIZE
```

The **DOUBLE**, **NOFRAME**, and **RESIZE** attributes specify a WINDOW 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.

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 structure specifies the default display font for all controls in the WINDOW 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)
```

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.

The three-dimensional look may be disabled by SET3DLOOK.

HLP (set 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, WINDOW, or control. 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. 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 *helpID*, followed by a unique identifier associated with exactly one help topic. If the tilde is missing, the *helpID* 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."

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.

ICON([*file*])

ICON Specifies an icon to display for the APPLICATION or WINDOW.

file A string constant or EQUATE containing the name of the .ICO file or Windows standard icon to display.

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. 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.

The following EQUATES are in EQUATES.CLW:

```
ICON:None           EQUATE ('<0FFH,01H,00H,00H>')
ICON:Application    EQUATE ('<0FFH,01H,01H,7FH>')
ICON:Hand           EQUATE ('<0FFH,01H,02H,7FH>')
ICON:Question       EQUATE ('<0FFH,01H,03H,7FH>')
ICON:Exclamation    EQUATE ('<0FFH,01H,04H,7FH>')
ICON:Asterisk       EQUATE ('<0FFH,01H,05H,7FH>')
ICON:Pick           EQUATE ('<0FFH,02H,01H,7FH>')
ICON:Save           EQUATE ('<0FFH,02H,02H,7FH>')
ICON:Print          EQUATE ('<0FFH,02H,03H,7FH>')
ICON:Paste          EQUATE ('<0FFH,02H,04H,7FH>')
ICON:Open           EQUATE ('<0FFH,02H,05H,7FH>')
ICON:New            EQUATE ('<0FFH,02H,06H,7FH>')
ICON:Help           EQUATE ('<0FFH,02H,07H,7FH>')
ICON:Cut            EQUATE ('<0FFH,02H,08H,7FH>')
ICON:Copy           EQUATE ('<0FFH,02H,09H,7FH>')

ICON:VCRtop         EQUATE ('<0FFH,02H,81H,7FH>')
ICON:VCRrewind      EQUATE ('<0FFH,02H,82H,7FH>')
ICON:VCRback        EQUATE ('<0FFH,02H,83H,7FH>')
ICON:VCRplay        EQUATE ('<0FFH,02H,84H,7FH>')
ICON:VCRfastforward EQUATE ('<0FFH,02H,85H,7FH>')
ICON:VCRbottom      EQUATE ('<0FFH,02H,86H,7FH>')
ICON:VCRlocate      EQUATE ('<0FFH,02H,87H,7FH>')
```

PAT (set pattern editing data entry)

PAT

The **PAT** 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.

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. 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.

MDI (set Multiple Document Interface)

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.

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. It does not, however, prevent the user from changing to another application running under Windows.

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 (without the MDI attribute) opened in an MDI program. 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, other windows on other execution threads may be selected by means of the mouse, keyboard, or menu commands. If so, the other window will then take the focus and become uppermost on the video display. Any window opened previously on the same execution thread may not be selected to receive focus, even from a modeless window.

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 a number of zones specified by the *widths* parameters. The size of each zone is specified in dialog units unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attribute. 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 any zone of the status bar using the MESSAGETEXT statement. The text displayed in a zone may be retrieved using the GETMESSAGETEXT function. The text remains present until replaced.

The status bar configuration can also be changed dynamically by calling the SETSTATUSBAR procedure/function.

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.

THOUSINCH, MILLIMETERS, POINTS (set screen coordinate measure)

```
THOUSINCH
MILLIMETERS
POINTS
```

The **THOUSINCH**, **MILLIMETERS**, and **POINTS** attributes specify the coordinate measure used to position controls on the WINDOW.

The **THOUSINCH** attribute specifies thousandths of an inch, **MILLIMETERS** specifies millimeters, and **POINTS** specifies points (there are seventy-two points per inch, both vertically and horizontally).

If all these attributes are omitted, measurement defaults 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.

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 **TIMER** attribute specifies generation of a periodic event whenever the time *period* passes. EQUATES.CLW contains EVENT:Timer which equates the timer-generated event. The ACCEPTED() and SELECTED() functions both return zero (0) when the timed event occurs. The FIELD() function returns the number of the control that currently has focus.

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 affect only the currently displayed menu, even if global items are changed. The exception to this is when changes are made to the Global menu when the APPLICATION is the current window, from within the APPLICATION's own ACCEPT loop. These changes 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.

TOOLBAR (declare a tool bar)

```
TOOLBAR [,AT( )] [,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.

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 "hot" 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 affect only the currently displayed toolbar, even if global tools are changed. The exception to this is when changes are made to the Global tools when the APPLICATION is the

current window—from within the APPLICATION's own ACCEPT loop. These changes 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.

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.

MENU (declare a drop-down menu)

```
MENU(text) [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,STD( )] [,RIGHT]
                [, FIRST ]
                [, LAST  ]
```

MENU Declares a menu box within a MENUBAR.

text A string constant or variable 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.

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 "hot" 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 a "hot" key for the field. If the MENU is on the menu bar, pressing the Alt key together with the ampersand "hot" key highlights and displays the MENU. If the MENU is within another MENU, pressing the ampersand "hot" 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 "hot" key for the MENU. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

```
ITEM(text) [,USE()] [,KEY()] [,MSG()] [,HLP()] [,STD()] [,TOGGLE] [,DISABLE]
  [, | FIRST | ] [,SEPARATOR]
     | LAST  |
```

ITEM Declares a menu choice within a MENU structure.

text A string constant or variable containing the display text for the menu item.

USE A variable (only for use with the TOGGLE attribute) or field equate label to reference the menu item in executable code.

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.

TOGGLE 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 display 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 MENU structure. The *text* string may contain an ampersand (&) which designates the following character as a "hot" key for the field. Pressing the ampersand "hot" key, alone, highlights and executes the ITEM. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the "hot" key for the ITEM. 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 attribute 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.

BOX (declare a box control)

```
BOX ,AT( ) [,USE( )] [,DISABLE] [,COLOR( )] [,FILL( )] [,ROUND] [,FULL]
    [,SCROLL]
```

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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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.

```
BUTTON(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
  [,HLP( )] [,SKIP] [,STD( )] [,FONT( )] [,ICON( )] [,DEFAULT] [,IMM]
  [,REQ] [,FULL] [,SCROLL] [,ALRT( )]
```

BUTTON Places a pushbutton on the WINDOW or TOOLBAR.

text A string constant containing the text to display on the button. This may contain an ampersand (&) to indicate the "hot" letter for the pushbutton.

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 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.

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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

ALRT Specifies "hot" keys active for the control.

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 place of its *text* parameter. The *text* parameter then serves only for "hot" key definition.

Events Generated:

EVENT:Selected
The control has received input focus.
EVENT:Accepted
The control has been pressed by the user.

CHECK (declare a check box control)

```
CHECK(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
   [,HLP( )] [,SKIP] [,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,ALRT( )]
   [, | 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.

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.

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).

FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

ALRT Specifies "hot" keys active for the control.

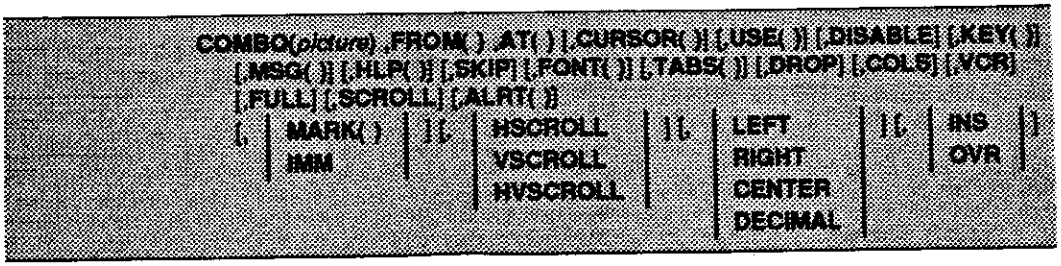
The **CHECK** control places a check box on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

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).

Events Generated:

EVENT:Selected
The control has received input focus.
EVENT:Accepted
The control has been toggled by the user.

COMBO (declare an entry/list control)



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 does not receive input focus.

FONT Specifies the display font for the control.

TABS Specifies the display format of the data.

DROP Specifies a drop-down list box and the number of elements the drop-down portion

contains.

COLS 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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

ALRT Specifies "hot" keys active for the control.

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 PAT attribute).

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. The user may type in data or select an item from the list. The entered data is not 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.

Events Generated:

EVENT:Selected
The control has received input focus.
EVENT:Accepted
The user has selected an entry.

A COMBO with the IMM attribute also generates the following events:

EVENT:NewSelection
The current selection in the list has changed (highlight has moved up or down).
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:Locate
The user pressed the locator VCR button.

CUSTOM (declare a .VBX custom control)

```
CUSTOM(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
[,HLP( )] [,SKIP] [,FULL] [,SCROLL] [,ALRT( )] [,CLASS( )] [,PROPERTY( )]
```

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.
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.
FULL	Specifies the control occupies the entire WINDOW.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
CLASS	Specifies the .VBX filename and type of control.
PROPERTY	Specifies properties specific to the custom control.

The **CUSTOM** control places a Visual Basic .VBX control on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

*** NOT YET FULLY IMPLEMENTED.

ELLIPSE (declare an ellipse control)

```
ELLIPSE,AT( ) [,USE( )] [,DISABLE] [,COLOR( )] [,FILL( )] [,FULL] [,SCROLL]
```

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.

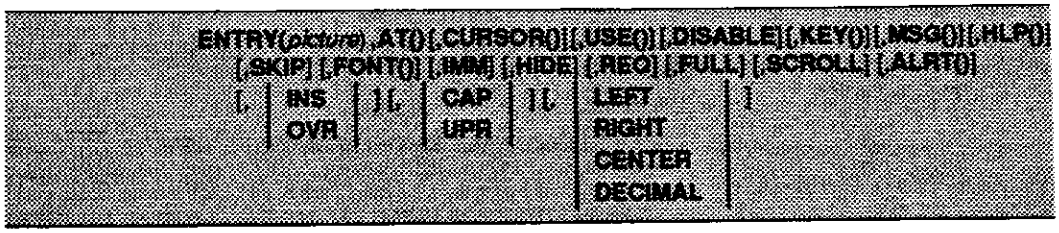
FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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 its AT 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.

ENTRY (declare a data entry control)



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 does not receive input focus.

FONT Specifies the display font for the control.

IMM Specifies immediate event generation whenever the user presses any key.

HIDE Specifies non-display of the data entered (password mode).

REQ Specifies the control may not be left blank or zero.

FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

ALRT Specifies "hot" keys active for the control.

INS / OVR Specifies Insert or Overwrite entry mode (valid only on windows with the PAT 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 HIDE 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 display-only. This differs from a STRING control with a USE attribute only in the appearance on screen.

Events Generated:

EVENT:Selected
The control has received input focus.
EVENT:Accepted
The user has completed data entry in the control.

An ENTRY with the IMM attribute also generates the following events:

EVENT:NewSelection
The user has pressed a key.

GROUP. (declare a group of controls)

```
GROUP(text) ,AT() [,CURSOR()] [,USE()] [,DISABLE] [,KEY()] [,MSG()] [,HLP()]
   [,FONT()] [,BOXED] [,HIDE] [,REQ] [,FULL] [,SCROLL]
      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 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 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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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 and does not generate events.

IMAGE (declare a graphic image control)

```
IMAGE(file) ,AT( ) [,USE( )] [,DISABLE] [,FULL] [,SCROLL]
              [, | 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.
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 occupies the entire WINDOW.
SCROLL	Specifies the control scrolls with the window.
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 area specified for display.

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), or windows metafile (.WMF).

This control cannot receive input focus and does not generate events.

```
LINE ,AT( ) [,USE( )] [,DISABLE] [,COLOR( )] [,FULL] [,SCROLL]
```

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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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* 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

This control cannot receive input focus and does not generate events.

LIST (declare a list control)

```
LIST ,FROM( ) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
     [,HLP( )] [,SKIP] [,FONT( )] [,TABS( )] [,DROP] [,COLS] [,VCR] [,FULL]
     [,SCROLL] [,HIDE] [,ALRT( )]
     [, | MARK( ) | ] [, | HSCROLL | ] [, | LEFT   | ]
        | IMM       |    | VSCROLL  |    | RIGHT  |
                         | HVSCROLL |    | CENTER |
                                         | DECIMAL|
```

LIST Places a 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.

FONT Specifies the display font for the control.

TABS Specifies the display format of the data.

DROP Specifies a drop-down list box and the number of elements the drop-down portion contains.

COLS 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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

HIDE Specifies the highlight bar is displayed only when the LIST has focus.

ALRT Specifies "hot" keys active for 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 list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

Events Generated:

EVENT:Selected
The control has received input focus.
EVENT:Accepted
The user has selected an entry from the control.

A LIST with the IMM attribute also generates the following events:

EVENT:NewSelection
The current selection in the list has changed (highlight has moved up or down).
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:Locate
The user pressed the locator VCR button.

```
OPTION(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
   [,HLP( )] [,BOXED] [,FULL] [,SCROLL]
  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 string variable to receive the value of the RADIO 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 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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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. The USE variable receives the text displayed for the RADIO selected by the user.

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. 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.

PROMPT (declare a prompt control)

```
PROMPT(text) ,AT() [,CURSOR( )] [,USE( )] [,DISABLE] [,FONT( )]
[,FULL] [,SCROLL]
```

PROMPT Places a prompt for the next 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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

The **PROMPT** control places a prompt for the next 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. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

This control cannot receive input focus and does not generate events.

RADIO (declare a radio button control)

```
RADIO(text),AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
[,HLP( )] [,SKIP] [,FONT( )] [,ICON( )] [, | LEFT | ] [,FULL] [,SCROLL]
                                           | 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.

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.

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).

FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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* is placed in the OPTION's USE variable.

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.

Events Generated:

EVENT:Selected
The control has received input focus.
EVENT:Accepted
The control has been selected by the user.

REGION (declare a window region control)

```
REGION [,AT( )] [,CURSOR( )] [,USE( )] [,DISABLE] [,FILL] [,COLOR( )] [,IMM]
  [,FULL] [,SCROLL]
```

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 occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

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.

Events Generated:

EVENT:Accepted
The mouse has been clicked by the user in the region.
EVENT:MouseIn (when IMM present, only)
The mouse has entered the region.
EVENT:MouseOut (when IMM present, only)
The mouse has left the region.
EVENT:MouseMove (when IMM present, only)
The mouse has moved within the region.

```
SPIN(picture),AT() [,CURSOR()] [,USE()] [,DISABLE] [,KEY()] [,MSG()] [,HLP()]
   [,SKIP] [,FONT()] [,FULL] [,SCROLL] [,ALRT()]
   [, | LEFT   | ][, | INS | ], | RANGE(),STEP |
      | RIGHT  |     | OVR |    | FROM()      |
      | 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 does not receive input focus.

FONT Specifies the display font for the control.

FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

ALRT Specifies "hot" keys active for the control.

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 PAT 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.

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).

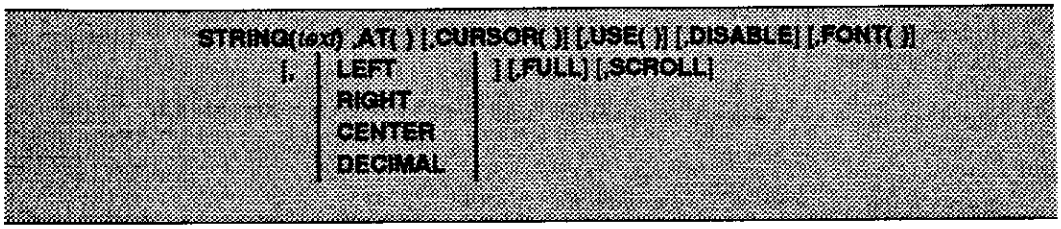
If the SPIN control offers the user regularly spaced numeric choices, the RANGE attribute specifies the valid range of values from which the user may choose. The STEP attribute then works in conjunction with RANGE to increment/decrement those values by the specified amount. If the choices are not regular, 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:NewSelection
The user has changed the displayed value.

STRING (declare a string control)



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*.

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.

FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

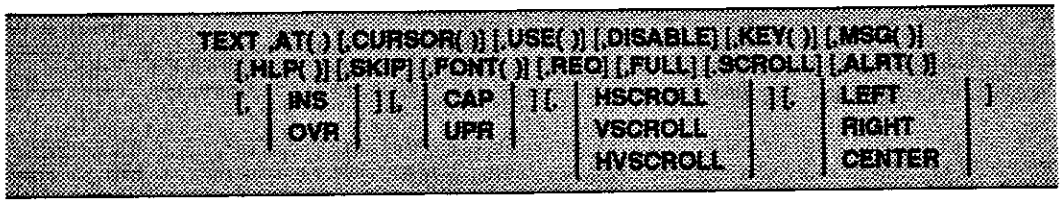
The **STRING** control places the *text* on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

If the *text* parameter is a picture token instead of a string constant 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. This makes the STRING with a USE variable a "display-only" control for the variable.

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.

This control cannot receive input focus and does not generate events.

TEXT (declare a multi-line data entry control)



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 does not receive input focus.

FONT Specifies the display font for the control.

REQ Specifies the control may not be left blank or zero.

FULL Specifies the control occupies the entire WINDOW.

SCROLL Specifies the control scrolls with the window.

ALRT Specifies "hot" keys active for the control.

INS / OVR Specifies Insert or Overwrite entry mode (valid only on windows with the PAT attribute).

UPR / CAP Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) 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.

ALRT (set "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, a field-accepted event is generated (ACCEPTED returns the field number of the control and SELECTED returns zero). You may have multiple ALRT attributes on one control.

Example:

```
Screen      WINDOW,AT(6,40),PRE(Scr)
              ENTRY,AT(6,40),USE(SomeVar),ALRT(F10Key)      !F10 alerted for control
            END
            CODE
            OPEN(Screen)
            ACCEPT
              CASE ACCEPTED()
              OF ?SomeVar
                IF EVENT() = EVENT:AlertKey AND KEYCODE() = F10Key
                                                            !Check for the "hot" key
                  RETURN
                END
              END
            END
```

AT (set control position and size)

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.

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.

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 default to dialog units unless the THOUSINCH, MILLIMETERS, or POINTS attribute is also present on the window containing the control. 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.

BOXED (set 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 case)

```
CAP
UPR
```

The **CAP** and **UPR** attributes specify the automatic case of text entered into ENTRY or TEXT controls. 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.

CLASS (set 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.

COLOR (set color)

```
COLOR(rgb)
```

COLOR Specifies display color.

rgb 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 **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.

COLS (set list box highlight bar)

COLS

The **COLS** 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 or variable containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the control.

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

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.

FILL (set fill color)

FILL(*rgb*)

FILL Specifies display fill color.

rgb 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 **FILL** attribute specifies the display fill color of a BOX, ELLIPSE, or REGION control. If omitted, the control is not filled with color.

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:

Global selections with FIRST attribute
Local selections with FIRST attribute

Global selections without FIRST or LAST attributes
Local selections without FIRST or LAST attributes

Global selections with LAST attribute
Local selections with LAST attribute

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 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, constant expression, or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

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)
```

FROM (set list box 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 or variable 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 TABS 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 or variable 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 (| |).

FULL (set full-screen)

FULL

The **FULL** attribute specifies the control occupies the entire WINDOW. FULL may not be specified for TOOLBAR controls.

HIDE (set data non-display)

```
HIDE
```

The **HIDE** 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.

HIDE may also be placed on a LIST control. This indicates the currently selected element in the LIST is only highlighted when the LIST has focus.

HSCROLL, VSCROLL, HVSCROLL (set control scroll bars)

```
HSCROLL
VSCROLL
HVSCROLL
```

The **HSCROLL, VSCROLL,** and **HVSCROLL** parameters 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.

ICON (set control icon)

```
ICON( file )
```

ICON	Specifies an icon to display as the control.
file	A string constant or variable or EQUATE containing the name of an .ICO file or Windows standard icon to display.

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.

IMM (set immediate event notification)

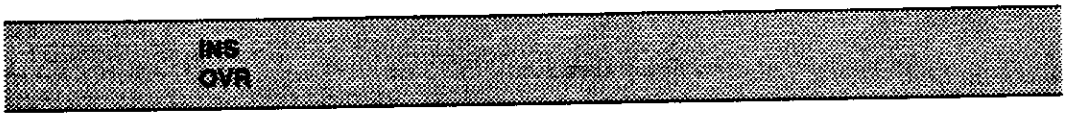
The **IMM** attribute specifies immediate generation of an event.

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 determined 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. It does the same thing on an ENTRY control.

INS, OVR (set typing mode)



The **INS** and **OVR** attributes specify the typing mode for an ENTRY or TEXT control. INS specifies insert mode while OVR specifies overwrite mode. These modes are only active on windows with the PAT attribute.

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 receive focus and immediately execute:

BUTTON
CHECK
CUSTOM
RADIO

LEFT, RIGHT, CENTER, DECIMAL (set 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 unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attribute.

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. On the DECIMAL attribute, *indent* specifies the offset of the decimal point from the right.

The following controls allow LEFT or RIGHT only (without an *indent* parameter):

CHECK
RADIO

The following controls allow LEFT(*indent*), RIGHT(*indent*), or CENTER(*indent*):

COMBO
ENTRY
LIST
SPIN
STRING
TEXT

The following controls allow DECIMAL(*indent*):

COMBO
ENTRY
LIST
SPIN
STRING

MARK (set multiple selection mode)

MARK(*flag*)

MARK Enables multiple items selection.

flag The label of a QUEUE field or array.

The **MARK** attribute enables multiple items selection from a LIST or COMBO control. When an item in the LIST is selected, the appropriate *flag* element is set to true (1). Each marked entry is highlighted in the LIST or COMBO. Changing the value of an element of the *flag* also changes the screen display for the related LIST or COMBO entry.

MSG (set 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.

PROPERTY (set custom control property)

PROPERTY(*property,value*)

PROPERTY Specifies additional properties for a CUSTOM control.

property A string constant containing the property number or EQUATE.

value A string constant containing the property value number or EQUATE.

The **PROPERTY** attribute specifies any additional properties needed for a CUSTOM control. 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.

A single CUSTOM control declaration may have multiple PROPERTY attributes.

RANGE (set SPIN range limits)

RANGE(*lower,upper*)

RANGE Specifies the valid range of data values the user may select in a SPIN 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. This attribute works in conjunction with the STEP attribute to provide the user with choices in the SPIN control. When RANGE and STEP are used, the SPIN control's FROM attribute is not.

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 immediately 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 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.

SKIP (set display-only)

```
SKIP
```

The **SKIP** attribute specifies a control that does not receive focus. This creates a "display-only" control.

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.

Standard actions include:

Tile Windows
Cascade Windows
Arrange Icons
Help Contents
... and more to come

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.

TABS(*format*)

TABS Specifies the format of the data in the LIST or COMBO control.

format A string constant specifying the column or multi-column format.

The **TABS** attribute specifies the 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" with a one-to-one mapping to the fields of the QUEUE, and/or "region-specifiers" that group together one or more fields displayed as a unit.

The following describes the components allowed in a *format* string:

"Field-specifier" format: *width justification* [(*indent*)] [*modifiers*]

width An integer defining the width of the field in dialog units unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attribute. Required.

justification A single capital letter (**L** , **R** , **C** , or **D**) that specifies Left, Right, Center, Or decimal justification. One is required.

indent An optional integer, enclosed in parentheses, that specifies an indent from the justification. This is measured in dialog units and 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 display format of the field. Multiple *modifiers* may be used on one field.

_ An underscore underlines the field.

| A vertical bar places a vertical line to the right of the field.

/ A slash causes the next field to appear on a new line (can only be used within a "region-specifier").

~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 overridden.

@picture@[**E**]
A *picture* token formats the field for display. The trailing @ is required to define the end of the *picture* token, so that *picture* tokens like @N12~Kr~ may be used

in the *format* string without creating ambiguity. If the trailing @ is followed by an **'E'**, the field can be directly edited in the list box, and the appropriate QUEUE entry is automatically updated with the edited value.

? 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.

"Region-specifier" format: **[** *multiple field-specifiers* **]** [*modifiers*]

A "region-specifier" differs from a "field-specifier" in that it relates to groups of fields. The required square brackets (**[ ]**) that enclose the fields cause them to be treated as a single display unit.

modifiers The "region-specifier" *modifiers* act on the entire group of fields.

_ An underscore underlines the group of fields.

| A vertical bar places a vertical line to the right of the group of fields.

~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* is centered unless a justification is defined.

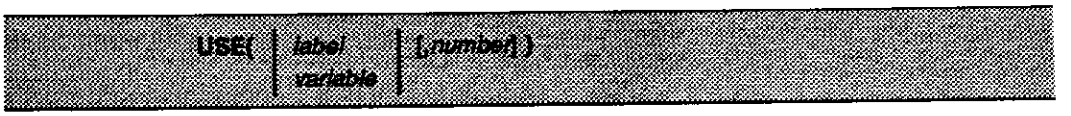
F An **F** creates a fixed column in the list that stays on screen when the user horizontally pages through the fields (by the HSCROLL attribute). Fixed regions must be at the start of the list.

M An **M** allows the group of fields to be dynamically re-sized at runtime. This allows the right vertical bar to be dragged.

S*integer*
An **S** followed by an *integer* adds a scroll bar to the group. The *integer* defines the total number of dialog units to scroll for the group of fields. This allows large fields to be displayed in a small column width.

TOGGLE

The **TOGGLE** 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).



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 a variable to receive the value entered into the control.

number An integer constant that specifies the number the compiler equates to the field equate label for the control.

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 TOGGLE 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 structure.

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.

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 Video Cassette Recorder (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
<< Page Up
< Entry Up
> Entry Down
>> Page Down
>| Bottom of list

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.

Event Processing

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. Only those events that actually 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 three types of events passed on to the program by ACCEPT: **Field-selection**, **Field-action**, and **Field-independent** events.

- A **Field-selection** event occurs when a control has just received input focus.
- A **Field-action** event occurs when the user presses a key that requires 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).

```
ACCEPT
  statements
END
```

ACCEPT The event handler.

statements Executable code statements.

The **ACCEPT** loop is the event handler required to process events generated by Windows for the APPLICATION or WINDOW structures. 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 the USE variables of all controls are automatically displayed 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.

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 of the control on the current window that has input focus.
ACCEPTED()	returns the field number for the control to which the event refers, if the event is a field-action event.
SELECTED()	returns the field number of the control receiving input focus, if the event is a field-selection event.
MOUSEX()	returns the x-coordinate of the mouse cursor at the time of the event.
MOUSEY()	returns the y-coordinate of the mouse cursor at the time of the event.

Two events cause an implicit BREAK from the ACCEPT loop. These are the events that signal the close of a window or close of a program. The program's code need not check for these events as they are handled automatically. However, the program 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 would return to the top of the ACCEPT loop without exit).

Example:

```
CODE
OPEN(Window)
ACCEPT
  CASE SELECTED()
  OF ?field1
      ! pre-edit code for field1
  OF ?field2
      ! pre-edit code for field2
  END
  CASE ACCEPTED()
  OF 0
    CASE EVENT()
      ! Handle any messages that are not field related here
    END
  OF ?field1
      ! completion code for field1
  OF ?field2
      ! completion code for field2
  END
END
```

See Also: EVENT, ACCEPTED, SELECTED

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. 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. When an event 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).

Example:

```
ALERT                          !Turn off all alerted keys
ALERT(F1Key,F12Key)            !Alert all function keys
ALERT(279)                     !Alert the Ctrl-Esc key
```

See Also: UPDATE

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* **[,** *position* **])**

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 or MENU to contain the new *control*, or the control immediately preceding the new *control* in the TAB key sequence.

position An integer constant, expression, or variable that specifies the position within the *parent* that the new *control* will occupy.

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 with SETPROPERTY, 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*. The EQUATES.CLW file contains equates for the *type* parameter for the following controls:

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:

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:

```
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,?Pause)           !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 DISABLEd. 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:

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 default to dialog units unless the THOUSINCH, MILLIMETERS, 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 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:

See Also: SETPOSITION

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:

```
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

MESSAGETEXT (set status bar contents)

MESSAGETEXT(*text*[,*zone*])

MESSAGETEXT Places *text* in the status bar.

text A string constant or variable containing the message to display.

zone An integer constant or variable containing the number of a status bar zone. If omitted, the *text* is placed in zone 1.

The **MESSAGETEXT** procedure displays the *text* in the specified *zone* of the status bar. The first zone of the status bar always displays MSG attributes of window controls. Text may be placed in any zone of the status bar using this procedure. Once placed in the status bar, the text remains present until replaced.

Example:

OPEN (open screen 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 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*]])

SELECT Sets the next control to receive input focus.

control A field number or field equate label of the next control to process.

position Specifies a position within the *control* to place the cursor. For an ENTRY or TEXT control this is a character position. For an OPTION structure, this is the selection number within the OPTION. For a LIST control, this is the QUEUE entry number.

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 receive focus because a DISABLE or HIDE statement has been issued, focus goes to the next control following it in the window's source code that can receive focus.

Example:

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.

Example:

SETCURSOR (set temporary mouse cursor)

SETCURSOR([*cursor*])

SETCURSOR Specifies a temporary mouse cursor to display.

cursor An EQUATE naming a Windows-standard mouse cursor.

The **SETCURSOR** statement specifies a temporary mouse cursor to be displayed only until the next ACCEPT loop begins. This cursor overrides all CURSOR attributes. When the next ACCEPT loop is encountered, SETCURSOR is done and 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 cannot break into.

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:

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.

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:

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 default to dialog units unless the THOUSINCH, MILLIMETERS, or POINTS attribute is specified on the APPLICATION or WINDOW. 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:

See Also: GETPOSITION

SETPROPERTY(*control* ,*property* [, *value*] [,*index*])

SETPROPERTY Dynamically specifies the appearance and/or behavior of any attribute 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.

property An integer constant, expression, EQUATE, or variable that specifies the property to change.

value An integer constant, expression, EQUATE, or variable that specifies the new value of the *property*.

index An integer constant, expression, or variable that specifies the index into a .VBX custom control array of properties.

SETPROPERTY dynamically specifies the appearance and/or behavior of any attribute of an APPLICATION, WINDOW, or control.

The EQUATES.CLW file contains the following equates for the *property* and *value* parameters:

PROP:Text	*text* parameter of the control
PROP:Msg	MSG() attribute
PROP:Hlp	HLP() attribute
PROP:Key	KEY() attribute
PROP:USE	USE() attribute
PROP:Cursor	CURSOR() attribute
PROP:Color	COLOR() attribute
PROP:Fill	FILL() attribute
PROP:Std	STD() attribute
PROP:Icon	ICON() attribute
PROP:RangeHigh	RANGE() attribute 2nd parameter
PROP:RangeLow	RANGE() attribute 1st parameter
PROP:Step	STEP() attribute
PROP:From	FROM() attribute
PROP:Mark	MARK() attribute
PROP:Tabs	TABS() attribute
PROP:Drop	DROP() attribute
PROP:Cols	COLS attribute
PROP:Align	LEFT, RIGHT, CENTER, or DECIMAL attribute
PROP:Indent	LEFT, RIGHT, CENTER, or DECIMAL attribute *indent* parameter
PROP:Round	ROUND attribute
PROP:Skip	SKIP attribute
PROP:Imm	IMM attribute

PROP:Hide	HIDE attribute
PROP:Req	REQ attribute
PROP:InsOvr	INS or OVR attribute
PROP:CapUpr	CAP or UPR attribute
PROP:Default	DEFAULT attribute
PROP:Boxed	BOXED attribute
PROP:Hscroll	HSCROLL attribute
PROP:Vscroll	VSCROLL attribute
PROP:System	SYSTEM attribute
PROP:Max	MAX attribute
PROP:Resize	RESIZE attribute
PROP:Double	DOUBLE attribute
PROP:Noframe	NOFRAME attribute
PROP:VCR	VCR() attribute
PROP:VcrFeq	VCR() attribute *field* parameter
PROP:Disable	DISABLE attribute
PROP:FontName	FONT() attribute *typeface* parameter
PROP:FontStyle	FONT() attribute *style* parameter
PROP:FontSize	FONT() attribute *size* parameter
PROP:FontColor	FONT() attribute *color* parameter
PROP:Picture	*picture* parameter of the control
PROP:Full	FULL attribute

PROPVAL:TypeDft	remove INS or OVR attribute
PROPVAL:TypeOvr	OVR attribute
PROPVAL:TypeIns	INS attribute
PROPVAL:NoCase	remove CAP or UPR attribute
PROPVAL:UprCase	UPR attribute
PROPVAL:CapCase	CAP attribute
PROPVAL:NOALIGN	remove LEFT, RIGHT, CENTER, or DECIMAL attribute
PROPVAL:LEFT	LEFT attribute
PROPVAL:RIGHT	RIGHT attribute
PROPVAL:CENTER	CENTER attribute
PROPVAL:DECIMAL	DECIMAL attribute

Example:

See Also: GETPROPERTY

SETSTATUSBAR (set status bar configuration)

SETSTATUSBAR([*widths*])

SETSTATUSBAR Dynamically changes the status bar configuration.

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 **SETSTATUSBAR** procedure/function dynamically changes the status bar configuration. The status bar may be divided into a number of zones specified by the *widths* parameter. The size of each zone is specified in dialog units unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attribute. A negative value indicates the zone is expandable, but has a minimum width indicated by the parameter's absolute value. If no *widths* parameter is specified, a single expanding zone with no minimum width is created, which is equivalent to a SETSTATUSBAR(-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 any zone of the status bar using the MESSAGETEXT statement. The text displayed in a zone may be retrieved using the GETMESSAGETEXT function. The text remains present until replaced.

Example:

SETWINDOW (set current window or report)

SETWINDOW([*label*] [,*thread*])

SETWINDOW Sets the current window (or report) for drawing graphics and other window-interaction statements.

label 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 *label* structure is contained in the topmost procedure or function. If omitted, the current execution thread is used.

The **SETWINDOW** procedure makes the *label* the structure which is current for windows-interaction statements or drawing with the graphics primitives functions. These windows-interaction statements are: CREATE, SETPROPERTY, GETPROPERTY, SETPOSITION, GETPOSITION, SETFONT, GETFONT, DISABLE, HIDE, CONTENTS, DISPLAY, ERASE, and UPDATE. Using these statements with SETWINDOW allows you to manipulate the window display in the topmost window of any execution thread.

This *label* 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.

SETWINDOW does not change procedures—it does not change which ACCEPT loop receives the events generated by Windows. SETWINDOW without any parameters resets to the procedure and execution thread with the currently active ACCEPT loop.

Example:

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 city
```

See Also: Field equate Labels

ACCEPTED (return control just completed)

ACCEPTED( )

The **ACCEPTED** function returns the field number of the control on which an event occurred, if the event is a field-action event.

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([*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-action 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
```

CHOOSECOLOR (return chosen color)

CHOOSECOLOR([*title*] [,*rgb*])

CHOOSECOLOR Returns the color chosen by the user.

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 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 **CHOOSECOLOR** function displays the Windows standard color choice dialog box and returns the color chosen by the user. The *rgb* parameter sets the default color choice presented to the user in the color choice dialog.

CHOOSECOLOR returns a specific negative number (EQUATEd to COLOR:None) if the user pressed the Cancel button, or the number of the chosen color if the user pressed the Ok button.

Return Data Type: LONG

Example:

CHOOSEFONT (return font chosen by user)

CHOOSEFONT([*title*] [,*typeface*] [,*size*] [,*color*] [,*style*])

CHOOSEFONT 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 containing the name of the font. If omitted, the system font is used.

size An integer variable containing the size (in points) of the font. If omitted, the system default font size is used.

color A LONG integer 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 variable specifying the strike weight and style of the font. If omitted, the weight is normal.

The **CHOOSEFONT** function displays the Windows standard font choice dialog box to allow the user to choose a font. 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.

CHOOSEFONT returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button.

Return Data Type: LONG

Example:

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
```

EVENT (return what happened)

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 three types of events generated by ACCEPT. The type of event can be determined by the values returned by the ACCEPTED and SELECTED functions.

Field-action events:
The ACCEPTED function returns the field number of the control on which the event occurred, and the SELECTED function returns zero (0).

Field-selection events:
The SELECTED function returns the field number of the control that just received input focus, and the ACCEPTED function returns zero (0).

Field-independent events:
Both the ACCEPTED and SELECTED functions return zero (0).

Return Data Type: SHORT

Example:

```
ACCEPT
  CASE SELECTED()
  OF ?Control1
    !Pre-edit code here
  OF ?Control2
    !Pre-edit code here
  END
  CASE ACCEPTED()
  OF 0
    CASE EVENT()
    OF EVENT:Suspend
      !Save some stuff
    OF EVENT:Resume
      !Restore the stuff
    END
  OF ?Control1
      !Post-edit code here
  OF ?Control2
    CASE EVENT()
    OF EVENT:Accepted
      !Post-edit code here
    OF EVENT:NewSelection
      !Do something specific for the currently highlighted record
    END
  END
END
```

FIELD( )

The **FIELD** function returns the field number of the control which has focus at the time of the event.

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)
             .

  CODE
  ACCEPT
    CASE FIELD()                        !Control edit control
    OF ?Control1                        ! Field number 1
      IF Control1 = 0                   !  if no entry
        BEEP                            !    sound alarm
        SELECT(?)                       !    stay on control
      .
    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
    OF ?Control4                        ! Field number 4
      BREAK                             !  exit processing loop
  . .                                   !  end case, end loop
```

FIRSTFIELD (return first window control)

The **FIRSTFIELD** function returns the lowest field number in the currently active window.

Return Data Type: LONG

Example:

GETPROPERTY (return control property)

GETPROPERTY(*control* ,*property* [,*index*])

GETPROPERTY Returns the current value of an attribute.

control A field number or field equate label for the control to affect. If *control* is zero (0), it specifies the window.

property An constant, expression, EQUATE, or integer variable that specifies the property to get.

index An integer constant, expression, or variable that specifies the index into a .VBX custom control array of properties.

GETPROPERTY returns the current value of any attribute of an APPLICATION, WINDOW, or control.

The EQUATES.CLW file contains equates for the *property* parameter:

PROP:Text	*text* parameter of the control
PROP:Msg	MSG() attribute
PROP:Hlp	HLP() attribute
PROP:Key	KEY() attribute
PROP:USE	USE() attribute
PROP:Cursor	CURSOR() attribute
PROP:Color	COLOR() attribute
PROP:Fill	FILL() attribute
PROP:Std	STD() attribute
PROP:Icon	ICON() attribute
PROP:RangeHigh	RANGE() attribute 2nd parameter
PROP:RangeLow	RANGE() attribute 1st parameter
PROP:Step	STEP() attribute
PROP:From	FROM() attribute
PROP:Mark	MARK() attribute
PROP:Tabs	TABS() attribute
PROP:Drop	DROP() attribute
PROP:Cols	COLS attribute
PROP:Align	LEFT, RIGHT, CENTER, or DECIMAL attribute
PROP:Indent	LEFT, RIGHT, CENTER, or DECIMAL attribute *indent* parameter
PROP:Round	ROUND attribute
PROP:Skip	SKIP attribute
PROP:Imm	IMM attribute
PROP:Hide	HIDE attribute
PROP:Req	REQ attribute
PROP:InsOvr	INS or OVR attribute
PROP:CapUpr	CAP or UPR attribute
PROP:Default	DEFAULT attribute

PROP:Boxed	BOXED attribute
PROP:Hscroll	HSCROLL attribute
PROP:Vscroll	VSCROLL attribute
PROP:System	SYSTEM attribute
PROP:Max	MAX attribute
PROP:Resize	RESIZE attribute
PROP:Double	DOUBLE attribute
PROP:Noframe	NOFRAME attribute
PROP:VCR	VCR() attribute
PROP:VcrFeq	VCR() attribute *field* parameter
PROP:Disable	DISABLE attribute
PROP:FontName	FONT() attribute *typeface* parameter
PROP:FontStyle	FONT() attribute *style* parameter
PROP:FontSize	FONT() attribute *size* parameter
PROP:FontColor	FONT() attribute *color* parameter
PROP:Picture	*picture* parameter of the control
PROP:Full	FULL attribute

Return Data Type: Dynamically Typed

Example:

See Also: SETPROPERTY

GETMESSAGETEXT (return status bar contents)

GETMESSAGETEXT([*zone*])

GETMESSAGETEXT Retrieves text from the status bar.

zone An integer constant or variable containing the number of a status bar zone. If omitted, the *text* is placed in zone 1.

The **GETMESSAGETEXT** function returns the text displayed in the specified *zone* of the status bar. The first zone of the status bar always displays MSG attributes of window controls. Text may be placed in any zone of the status bar using the MESSAGETEXT procedure. Once placed in the status bar, the text remains present until replaced.

Return Data Type: STRING

Example:

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:

LASTFIELD (return last window control)

LASTFIELD( )

The **LASTFIELD** function returns the highest field number in the currently active window.

Return Data Type: LONG

Example:

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, unless the THOUSINCH, MILLIMETERS, or POINTS parameter is on the currently active window.

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, unless the THOUSINCH, MILLIMETERS, or POINTS parameter is on the currently active window.

Return Data Type: SHORT

Example:

```
SaveMouseY = MOUSEY()                    !Save mouse position
```

REFER()

The **REFER** function returns 1 (true) if the last control completed was referenced, and 0 (false) if it was not referenced. A control is referenced by: typing any displayable character; completing an OPTION control; or moving the highlight bar in a LIST or COMBO box. REFER returns true if the same characters are retyped in the control.

Return Data Type: LONG

Example:

```
Screen       WINDOW
               ENTRY(@N4),USE(Control1)      !Entry control
               ENTRY(@N4),USE(Control2)      !Entry control
               ENTRY(@N4),USE(Control3)      !Entry control
               ENTRY(@N4),USE(Control4)      !Entry control
               .

  CODE
  ACCEPT
    CASE FIELD()                 !Control edit control
    OF ?Control1                 !The first control
      IF NOT REFER()             !  if no entry
        BEEP                     !    sound alarm
        SELECT(?)                !    stay on control
      .
    OF ?Control4                 !Control4
      BREAK                      !  exit processing loop
    END
  END
```

SELECTED (return control that has received focus)

SELECTED( )

The **SELECTED** function returns the field number of the control receiving input focus. It returns zero (0) if the current value returned by the EVENT function is not EVENT:Selected.

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

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.

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 is used by procedures and functions whose action may be performed on any execution thread, such as SETWINDOW. The maximum number of simultaneously executing threads in a single application is 64.

The first execution thread in any program is the main program code, and 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 also return zero (0) to indicate failure to open the thread. This can occur by attempting to START a 65th thread, or by running out of memory.

Return Data Type: LONG

Example:

```
SaveThread1    LONG                    !Declare thread number save variable
SaveThread2    LONG                    !Declare thread number save variable
  CODE
  OPEN(ApplicationWindow)              !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
```

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.

Example:

```
ASK                                    !Wait for a keystroke

LOOP WHILE KEYBOARD()                  !Empty the keyboard buffer
  ASK                                  !  without processing keystrokes
END
```

PRESS(*string*)

PRESS Places keystrokes in the keyboard input buffer.

string A string constant, variable, or expression.

PRESS places keystrokes in the internal Clarion keyboard input buffer (not the DOS keyboard 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 Action = 'AddRecord'                 !On the way into a memo on adding a record
  TempStrong = 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 an internal Clarion keyboard input buffer (not the DOS keyboard buffer). Once placed in the keyboard buffer, the *keycode* is processed just as if the user had pressed the key.

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.

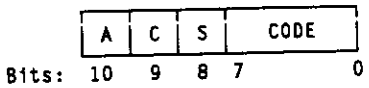
Example:

```
SETKEYCODE(999)                      !Set up the keycode function to return 999
```

See Also: KEYCODE, Keycode Equate Labels

Keycodes (keypress representation)

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:




```
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.

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

Reports in Windows

Clarion Database Developer 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( )] [ABORT] [DEFAULT] [LANDSCAPE]
                           [ | THOUSINCH | ]
                           [ | MILLIMETERS | ]
                           [ | 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 Windows system font is used.

PRE Specifies the label prefix for the report or structure.

ABORT Specifies that printing may be interrupted by the user.

DEFAULT Specifies printing to the current printer without prompting for destination. If omitted, the standard Windows Print dialog box is displayed when the report is opened.

LANDSCAPE Specifies printing in landscape mode. If omitted, printing defaults to portrait mode.

THOUSINCH Specifies thousandths of an inch as the measurement unit used for all attributes which use coordinates.

MILLIMETERS 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.

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.

Example:

```
CustRpt     REPORT                          !Declare customer report
              !report declarations
            END                             !End report declaration
```

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 THOUSINCH, MILLIMETERS, 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.

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 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 REPORT structure specifies the default print font for all controls in the REPORT 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)
```

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.

The **PRE** attribute on a REPORT provides a label prefix for DETAIL and BREAK structures. It is used to distinguish between identical variable names that occur in different structures. When referenced in executable statements, the *prefix* is attached to a label by a colon (Pre:Label).

Example:

```
Report      REPORT,PRE('Rpt')
DetailOne     DETAIL                !Always referenced as Rpt:DetailOne
              END                   !  in executable code
            END
```

See Also: Reserved Words

ABORT (set report abortable)

ABORT

The **ABORT** attribute on a REPORT provides the end-user the ability to abort report generation before it is finished. A (standard) dialog box is displayed during report generation that allows the user to cancel the report.

When the user aborts the report, all subsequent PRINT statements return immediately without printing.

Example:

```
Report      REPORT,PRE('Rpt'),ABORT       !Abortable report
DetailOne     DETAIL
              END
            END

  CODE
  OPEN(Report)                            !Cancel dialog displayed
```

DEFAULT (set current printer)

DEFAULT

The **DEFAULT** attribute on a REPORT indicates the report is to print on the currently set Windows default printer. If omitted, the standard Windows Print... dialog box is displayed to allow the user to choose the Windows standard report choices, such as destination, page orientation, and print-to-file.

Example:

```
Report      REPORT,PRE('Rpt'),DEFAULT
DetailOne     DETAIL                !Always referenced as Rpt:DetailOne
              END                   !  in executable code
            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 DEFAULT attribute is not set, this attribute sets the default page orientation selection on the Windows Print... dialog, which may be overridden by the user.

If the LANDSCAPE attribute is omitted, printing defaults to portrait mode.

Example:

```
Report      REPORT,PRE('Rpt'),LANDSCAPE
DetailOne     DETAIL                !Always referenced as Rpt:DetailOne
              END                   !  in executable code
            END
```

THOUSINCH, MILLIMETERS, POINTS (set report coordinate measure)

```
THOUSINCH
MILLIMETERS
POINTS
```

The **THOUSINCH**, **MILLIMETERS**, and **POINTS** attributes specify the coordinate measures used to position controls on the REPORT.

THOUSINCH specifies thousandths of an inch, **MILLIMETERS** 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.

FORM (page layout structure)

```
FORM ,AT( ) [,FONT( )]
  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.

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 is printed automatically when the first DETAIL structure (without an ALONE attribute) is PRINTed. FORM 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.

HEADER (page or group header structure)

```
HEADER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
       [,KEEPPRIOR( )] [,KEEPNEXT( )]
   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.

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.

KEEPPRIOR 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.

KEEPNEXT 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.

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.

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. 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 header declaration
            END                             !  end group break declaration
          END                               !End report declaration
```

DETAIL (report detail line structure)

```
label   DETAIL [AT( )] [,FONT( )] [,ALONE] [,ABSOLUTE] [,PAGEBEFORE( )]
            [,PAGEAFTER( )] [,KEEPPRIOR( )] [,KEEPNEXT( )]
          controls
        END
```

DETAIL Declares a line-item detail structure.

label The name by which the structure is addressed in executable code.

AT Specifies the 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 by itself without FORM, HEADER, or 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.

KEEPPRIOR Declares the DETAIL prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it.

KEEPNEXT Declares the DETAIL prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it.

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
GroupHead      DETAIL                      !  begin detail declaration
                 !structure elements
               END                         !  end detail declaration
CustDetail     DETAIL                      !  begin detail declaration
                 !structure elements
               END                         !  end detail declaration
             END                           !End report declaration
```

See Also: PRINT

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
```

FOOTER (page or group footer structure)

```
FOOTER AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
    [,KEEPPRIOR( )] [,KEEPNEXT( )]
  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.

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.

KEEPPRIOR 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.

KEEPNEXT 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.

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.

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. 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
```

Print Structure Attributes

AT (set print structure size)

AT([*x*] [,*y*] [,*width*] [,*height*])

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 minimum 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 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 THOUSINCH, MILLIMETERS, 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.

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 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 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)
```

ALONE (set to print without page header or footer)

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                          !Declare customer report
TitlePage      DETAIL,ALONE                  !Title page detail structure
                 !structure elements
                 .
CustDetail     DETAIL                        !  end detail declaration
                 !structure elements         !  begin detail declaration
                 .
               FOOTER                        !  end detail declaration
                 !structure elements         !  begin page footer declaration
                 .
             . .                             !  end footer and report
```

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                          !Declare customer report
               HEADER                        !  begin page header declaration
                 !structure elements
                 .                           !  end header declaration
GroupHead      DETAIL                        !  begin detail declaration
                 !structure elements
                 .                           !  end detail declaration
CustDetail     DETAIL,ABSOLUTE               !  fixed position detail
                 !structure elements
                 .                           !  end detail declaration
               FOOTER                        !  begin page footer declaration
                 !structure elements
             . .                             !  end footer and report
```

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                        !Declare customer report
               HEADER                      !Page header
                 !structure elements
               .                           !
GroupHead      DETAIL,PAGEBEFORE           !Group Header, initiates page overflow
                 !structure elements
               .                           !
CustDetail     DETAIL                      !Line item detail
                 !structure elements
               .                           !
GroupFoot      DETAIL                      !Group Footer
                 !structure elements
               .                           !
               FOOTER                      !
                 !structure elements
             . .                           !End report declaration
```

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                          !Declare customer report
               HEADER                        !Page header
                 !structure elements
               .                             !
GroupHead      DETAIL                        !Group Header
                 !structure elements
               .                             !
CustDetail     DETAIL                        !Line item detail
                 !structure elements
               .                             !
GroupFoot      DETAIL,PAGEAFTER              !Group Footer, initiates page overflow
                 !structure elements
               .                             !
               FOOTER                        !
                 !structure elements
             . .
                                             !End report declaration
```

KEEPPRIOR (set orphan elimination)

KEEPPRIOR([*siblings*])

KEEPPRIOR 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 **KEEPPRIOR** 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                        !Declare customer report
               BREAK(SomeVariable)         !Group Break structure
                 HEADER                    !Group Header
                   !structure elements
                 .                         !
CustDetail       DETAIL,KEEPPRIOR()        !Always print with 1 sibling
                   !structure elements
                 .                         !
                 FOOTER,KEEPPRIOR(2)       !Always print with 2 siblings
                   !structure elements
                 .                         !
             . .                           !End report declaration
```

KEEPNEXT[(*siblings*)]

KEEPNEXT 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 **KEEPNEXT** 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                          !Declare customer report
               BREAK(SomeVariable)           !Group Break structure
                 HEADER,KEEPNEXT(2)          !Always print with 2 siblings
                   !structure elements
                   .                         !
CustDetail         DETAIL,KEEPNEXT()         !Always print with 1 sibling
                     !structure elements
                     .                       !
                   FOOTER
                     !structure elements
                     .                       !
             . .                             !End report declaration
```

BOX (declare a box control)

```
BOX AT( ) [,USE( )] [,COLOR( )] [,FILL( )] [,ROUND]
```

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.

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.

CHECK (declare a check box control)

```
CHECK(text) AT( ) [,USE( )] [,FONT( )] [, 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.

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.

ELLIPSE (declare an ellipse control)

```
ELLIPSE ,AT( ) [,USE( )] [,COLOR( )] [,FILL( )]
```

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.

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."

GROUP (declare a group of controls)

```
GROUP(text),AT( ) [,USE( )] [,FONT( )] [,BOXED]
   [, | LEFT  | ]
      | RIGHT |
   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.

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.

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.

IMAGE (declare a graphic image control)

IMAGE(*file*) ,AT( ) [,USE( )]

IMAGE Places a graphic image on the REPORT.

file A string constant containing the name of the file to print.

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.

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), icon (.ICO), or windows metafile (.WMF).

LINE ,AT( ) [,USE( )] [,COLOR( )]

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.

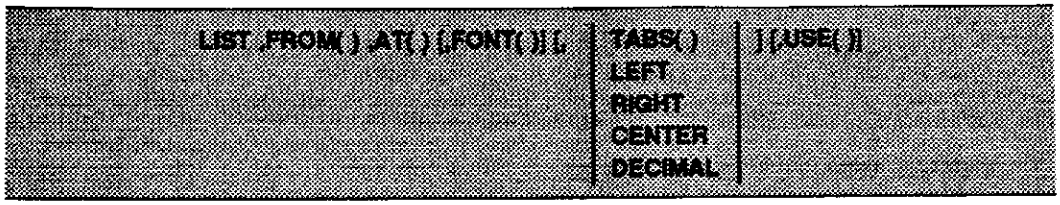
COLOR Specifies the color for the line. If omitted, the color is black.

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

LIST (declare a list control)



LIST

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.

TABS Specifies the display format of the data.

LEFT Specifies that the data is left justified within the LIST. Valid only without a TABS attribute.

RIGHT Specifies that the data is right justified within the LIST. Valid only without a TABS attribute.

CENTER Specifies that the data is centered within the LIST. Valid only without a TABS attribute.

DECIMAL Specifies that the data is aligned on the decimal point within the LIST. Valid only without a TABS attribute.

USE Specifies a field equate label for the control.

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. Its purpose is to allow the report format to duplicate the screen appearance of the LIST's TABS setting.

OPTION (declare a group of RADIO controls)

```
OPTION(text) ,AT( ) [,USE( )] [,BOXED]
  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.

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.

RADIO (declare a radio button control)

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.

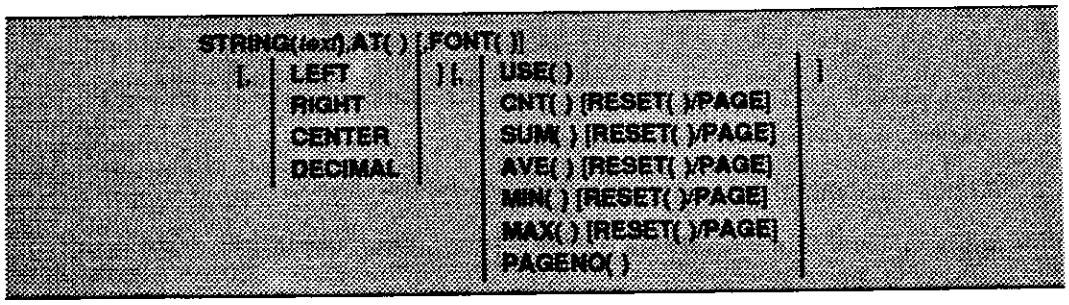
LEFT Specifies that the *text* appears to the left of the radio button.

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.

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.

STRING (declare a string control)



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*.

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.

USE Specifies a variable whose contents are 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 a variable is printed in the format of the picture token declared instead of string *text*.

AVE Specifies the average value of a variable is printed in the format of the picture token declared instead of string *text*.

MIN Specifies the minimum value of a variable is printed in the format of the picture token declared instead of string *text*.

MAX Specifies the maximum value of a variable is printed in the format of the picture

token declared instead of string *text*.

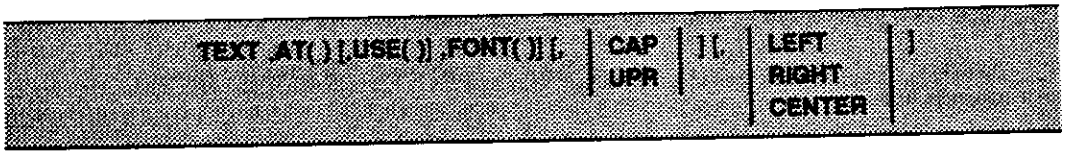
PAGENO Specifies the current page number is printed in the format of the picture token declared instead of string *text*.

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.

TEXT (declare a multi-line data entry control)



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.

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.

AT (set position and size)

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.

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 THOUSINCH, MILLIMETERS, 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.

AVE (set total average)

AVE(*variable*)

AVE Prints the average value of a variable.

variable The label of the variable.

The **AVE** attribute specifies the average (arithmetic mean) of the *variable* is calculated each time any DETAIL structure in the BREAK structure containing the control is PRINTed. The tally 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 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 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 o the page.

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 tally 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, 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 **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. Windows automatically finds the closest match to the specified *rgb* color value for the hardware on which the report is printed.

FILL (set fill color)

FILL(*rgb*)

FILL Specifies the print fill color of a BOX or ELLIPSE control.

rgb 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 **FILL** attribute specifies the print fill color of a BOX or ELLIPSE control. If omitted, the control is not filled with color.

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 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, 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)
```

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 TABS 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.

```
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 THOUSINCH, MILLIMETERS, 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*), or CENTER(*indent*):

LIST
STRING
TEXT

The following controls allow DECIMAL(*indent*):

LIST
STRING

MAX (set total maximum)

MAX(*variable*)

MAX Prints the maximum value of a variable.

variable The label of the variable.

The **MAX** attribute specifies printing the maximum value the *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 the maximum value of the variable in the page. 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.

MIN (set total minimum)

MIN(*variable*)

MIN Prints the minimum value of a variable.

variable The label of the variable.

The **MIN** attribute specifies printing the minimum value the *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 the minimum value of the variable in the page. 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 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).

ROUND (set round-cornered BOX)

ROUND

The **ROUND** attribute specifies a BOX control with rounded corners.

SUM(*variable*)

SUM Prints the sum of the values of a variable.

variable The label of the variable.

The **SUM** attribute specifies printing the sum of the values contained in the *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 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.

TABS(*format*)

TABS Specifies the print format for the data.

format A string constant specifying the column or multi-column format.

The **TABS** 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" with a one-to-one mapping to the fields of the QUEUE or GROUP, and/or "region-specifiers" that group together one or more fields displayed as a unit.

The following describes the components allowed in a *format* string:

"Field-specifier" format: *width justification* [(*indent*)] [*modifiers*]

width An integer defining the width of the field in dialog units unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attribute. Required.

justification A single capital letter (**L** , **R** , **C** , or **D**) that specifies Left, Right, Center, Or decimal justification. One is required.

indent An optional integer, enclosed in parentheses, that specifies an indent from the justification. This is measured in dialog units and 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 display format of the field. Multiple *modifiers* may be used on one field.

_ An underscore underlines the field.

| A vertical bar places a vertical line to the right of the field.

/ A slash causes the next field to appear on a new line (can only be used within a "region-specifier").

~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 overridden.

@picture@
A *picture* token formats the field for display. The trailing @ is required to define the end of the *picture* token, so that *picture* tokens like @N12~Kr~ may be used

in the *format* string without creating ambiguity.

"Region-specifier" format: **[** *multiple field-specifiers* **]** [*modifiers*]

A "region-specifier" differs from a "field-specifier" in that it relates to groups of fields. The required square brackets ([ ]) that enclose the fields cause them to be treated as a single display unit.

modifiers The "region-specifier" *modifiers* act on the entire group of fields.

_ An underscore underlines the group of fields.

| A vertical bar places a vertical line to the right of the group of fields.

~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* is centered unless a justification is defined.

USE(*variable*)

USE Specifies the data to print in the control or a field equate label with which to reference the control.

variable The label of a variable or a field equate label.

The **USE** attribute specifies the *variable* containing the data to print in the control or a field equate label with which to reference the control.

OPEN (open a report structure for processing)

OPEN(*report*)

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
```

CLOSE (close an active report structure)

CLOSE(*report*)

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. RETURN from a report procedure automatically closes a REPORT.

Example:

```
CLOSE(CustRpt)                                   !Close the report
```

PRINT (print a report structure)

PRINT(*structure*)

PRINT Prints a DETAIL structure.

structure The label of a DETAIL structure.

The **PRINT** statement prints a DETAIL to the destination specified by the user in the Windows Print... dialog. PRINT automatically activates group breaks and page overflow as needed.

Example:

```
PRINT(OrderDtl)                          !Print order detail line to DEVICE
```

See Also: Page Overflow

Graphics Overview

Clarion supplies the set of "graphics primitives" defined in this chapter to allow drawing in windows and reports.

Graphics are always drawn to the "current" window. Unless overridden with SETWINDOW, the "current" window 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, SETWINDOW must be used to nominate the REPORT as the "current window."

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 THOUSINCH, MILLIMETERS, or POINTS attributes. 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).

Controls always appear on top of any graphics drawn to the window. This means the graphics appear to underlie any controls in the window, so they don't get in the way of the controls the user needs to access.

Every window 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, SETPENWIDTH, SETPENCOLOR, and SETPENSTYLE statements should be issued for each window.

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 constant or variable that specifies the horizontal position of the starting point.
y	An integer constant or variable that specifies the vertical position of the starting point.
width	An integer constant or variable that specifies the width.
height	An integer constant or variable that specifies the height.
startangle	An integer constant or variable 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 constant or variable 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.

BLANK

The **BLANK** procedure erases all graphics written to the current window or report. Controls are not erased.

To erase only a portion of the window, use the other graphics commands (BOX, ELLIPSE, CHORD, POLYGON) to draw a figure the same color as the window background color.

BOX(*x* ,*y* ,*width* ,*height* [,*fill*])

BOX Draws a rectangular box on the current window or report.

x An integer constant or variable that specifies the horizontal position of the starting point.

y An integer constant or variable that specifies the vertical position of the starting point.

width An integer constant or variable that specifies the width.

height An integer constant or variable 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.

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 constant or variable that specifies the horizontal position of the starting point.

y An integer constant or variable that specifies the vertical position of the starting point.

width An integer constant or variable that specifies the width.

height An integer constant or variable that specifies the height.

startangle An integer constant or variable 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 constant or variable 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.

ELLIPSE (draw an ellipse)

ELLIPSE(*x* ,*y* ,*width* ,*height* [,*fill*])

ELLIPSE Draws an ellipse on the current window or report.

x An integer constant or variable that specifies the horizontal position of the starting point.

y An integer constant or variable that specifies the vertical position of the starting point.

width An integer constant or variable that specifies the width.

height An integer constant or variable 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.

LINE (draw a straight line)

LINE(*x* ,*y* ,*width* ,*height*)

LINE Draws a straight line on the current window or report.

x An integer constant or variable that specifies the horizontal position of the starting point.

y An integer constant or variable that specifies the vertical position of the starting point.

width An integer constant or variable that specifies the width. This may be a negative number.

height An integer constant or variable that specifies 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.

CHORD(*x* ,*y* ,*width* ,*height* ,*slices* ,*colors* [,*depth*] [,*wholevalue*] [,*startangle*])

PIE Draws a pie chart on the current window or report.

x An integer constant or variable that specifies the horizontal position of the starting point.

y An integer constant or variable that specifies the vertical position of the starting point.

width An integer constant or variable that specifies the width.

height An integer constant or variable that specifies the height.

slices An array of numeric values that specify the relative size of each slice of the pie.

colors An integer array that specifies the fill color for each slice.

depth An integer constant or variable 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.

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 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.

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 constant or variable that specifies the horizontal position of the starting point.

y An integer constant or variable that specifies the vertical position of the starting point.

width An integer constant or variable that specifies the width.

height An integer constant or variable 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.

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.

SETPENSTYLE (set line draw style)

```
SETPENSTYLE( [style] )
```

SETPENSTYLE Sets the current pen style.

style An integer constant 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.

SETPENWIDTH (set line draw thickness)

SETPENWIDTH(*width*)

SETPENWIDTH Sets the current pen width.

width An integer constant or variable that specifies the pen's thickness, measured in dialog units unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attributes. 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.

PENCOLOR (return line draw color)

```
PENCOLOR( )
```

The **PENCOLOR** function returns the current pen color set by SETPENCOLOR.

Return Data Type: LONG

PENSTYLE (return line draw style)

```
PENSTYLE( )
```

The **PENSTYLE** function returns the current line draw style set by SETPENSTYLE.

Return Data Type: LONG

PENWIDTH (return line draw thickness)

```
PENWIDTH( )
```

The **PENWIDTH** function returns the current line draw thickness set by SETPENWIDTH. The return value is dialog units unless overridden by the THOUSINCH, MILLIMETERS, or POINTS attributes.

Return Data Type: LONG

Queue Structure

QUEUE (declare a memory QUEUE structure)

```
label       QUEUE [,PRE] [,STATIC] [,THREAD]
fieldlabel    variable [,NAME( )]
            END
```

QUEUE Declares a memory queue structure.

label The name of the QUEUE.

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.

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.

QUEUE declares a memory QUEUE structure. A QUEUE is a doubly-linked list; each entry has a chain pointer to the entry before and after it. 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.

A QUEUE may be thought of as a "memory file." When a QUEUE is declared, a data buffer is assigned. The maximum allowable size of a QUEUE data buffer is 65,000 bytes.

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). 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, Member local, or Procedure local 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 is 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 65,535.

Generally, the memory used by each entry in the QUEUE is equal to the total of the field sizes plus 28 bytes overhead (system and pointers), rounded up to the nearest number evenly divisible by 16. For example: a QUEUE with a LONG and a SHORT (6 bytes total) occupies 48 bytes per entry (6 + 28 = 34, which rounds up to 48). The rounding is due to memory allocation for each entry always beginning at a 16-byte "paragraph boundary." However, before the entry is assigned its memory, any trailing spaces in the data buffer are automatically "clipped." This is more efficient, allowing more entries in less memory. Because of this, a good QUEUE design practice would be to place long STRING variables at the end of the data buffer.

In addition to the data buffer and memory for each entry, there is also a small amount of system "overhead" memory for the QUEUE. This is allocated when the QUEUE is first accessed and de-allocated only by the FREE statement.

Example:

```
NameQue     QUEUE,PRE(Nam)                          !Declare a queue
Name          STRING(20)
Zip           DECIMAL(5,0),NAME('SortField')
            END                                     !End queue structure
```

See Also: PRE, STATIC, NAME, FREE, THREAD