CA-Clipper®

For DOS

Version 5.3

Reference Guide

Volume 2

June 1995

COMPUTER ASSOCIATES®
Software superior by design.

PRINTED ON
RECYCLED PAPER

Printed in the United States of America
Computer Associates International, Inc.
Publisher

Contents

Volume 1

Chapter 1: Introduction

Chapter 2: Language Reference

Volume 2

Chapter 2: Language Reference (cont.)

Categorized Contents

Volume 2

Statements

Commands

Functions

Classes

JOIN command

Create a new database file by merging records/fields from two work areas

Syntax

```
JOIN WITH <xcAlias> TO <xcDatabase>
   FOR <lCondition> [FIELDS <idField list>]
```

Arguments

WITH <xcAlias> is the name of the work area to merge with records from the current work area. You can specify it either as a literal alias or as a character expression enclosed in parentheses.

TO <xcDatabase> is the name of the target database file specified either as a literal filename or as a character expression enclosed in parentheses.

FOR <lCondition> selects only records meeting the specified condition.

FIELDS <idField list> is the projection of fields from both work areas into the new database file. To specify any fields in the secondary work area, reference them with the alias. If the FIELDS clause is not specified, all fields from the primary work area are included in the target database file.

Description

JOIN creates a new database file by merging selected records and fields from two work areas based on a general condition. JOIN works by making a complete pass through the secondary work area for each record in the primary work area, evaluating the condition for each record in the secondary work area. When the *<lCondition>* is true (.T.), a new record is created in the target database file using the FIELDS specified from both work areas.

If SET DELETED is OFF, deleted records in both source files (i.e., the two files being JOINed) are processed. However, their deleted status is not retained in the target *<xcDatabase>*. No record in the target file is marked for deletion regardless of its deleted status in either of the source files.

If SET DELETED is ON, no deleted records are processed in either of the source files. Thus, deleted records do not become part of the target <*xcDatabase*>. Similarly, filtered records are not processed and do not become part of the target file.

Warning! *The number of records processed will be the LASTREC() of the primary work area multiplied by the LASTREC() of the secondary work area. For example, if you have two database files with 100 records each, the number of records JOIN processes is the equivalent of sequentially processing a single database file of 10,000 records. Therefore, use this command carefully.*

Examples

- This example joins Customer.dbf to Invoices.dbf to produce Purchases.dbf:

```
USE Invoices NEW
USE Customers NEW
JOIN WITH Invoices TO Purchases;
    FOR Last = Invoices->Last;
    FIELDS First, Last, Invoices->Number, ;
       Invoices->Amount
```

Files

Library is CLIPPER.LIB.

See Also

SET RELATION

KEYBOARD command

Stuff a string into the keyboard buffer

Syntax

```
KEYBOARD <cString>
```

Arguments

<cString> is the string to stuff into the keyboard buffer.

Description

KEYBOARD is a keyboard command that stuffs *<cString>* into the keyboard buffer after clearing any pending keystrokes. The characters remain in the keyboard buffer until fetched by a wait state such as ACCEPT, INPUT, READ, ACHOICE(), MEMOEDIT(), and DBEDIT(), or INKEY(). In addition to display characters, *<cString>* may include control characters.

Generally, to convert an INKEY() code of a control key to a character value, use CHR(). There are, however, some limitations. You can only stuff characters with INKEY() codes between zero and 255, inclusive, into the keyboard buffer.

Typically, KEYBOARD is used in a SET KEY procedure to reassign keys in a wait state. Another use within the ACHOICE() user function is to input the keys you want ACHOICE() to execute before returning control to it. The same concept applies to DBEDIT().

Examples

- This example uses KEYBOARD to stuff a GET with an item selected from a picklist each time the GET is entered:

```
#include "Inkey.ch"
LOCAL cVar1 := SPACE(10), nVar := 2500, ;
      cVar2 := SPACE(10)
CLEAR
@ 09, 10 GET cVar1
@ 10, 10 GET cVar2 WHEN PickList()
@ 11, 10 GET nVar
READ
RETURN

FUNCTION PickList
   STATIC aList := { "First", "Second", ;
      "Three", "Four" }
   LOCAL cScreen, nChoice, nKey := LASTKEY()
   cScreen := SAVESCREEN(10, 10, 14, 20)
   @ 10, 10 TO 14, 20 DOUBLE
   IF (nChoice := ACHOICE(11, 11, 13, 19, aList)) != 0
      KEYBOARD CHR(K_CTL_Y) + aList[nChoice] + ;
            CHR(nKey)
   ENDIF
   RESTSCREEN(10, 10, 14, 20, cScreen)
   RETURN .T.
```

Files

Library is CLIPPER.LIB, header file is Inkey.ch.

See Also

CHR(), INKEY(), LASTKEY(), NEXTKEY(), SET KEY, SET TYPEAHEAD

L2BIN() function

Convert a CA-Clipper numeric value to a 32-bit binary integer

Syntax

`L2BIN(<nExp>) → cBinaryInteger`

Arguments

<nExp> is the numeric value to be converted. Decimal digits are truncated.

Returns

L2BIN() returns a four-byte character string formatted as a 32-bit binary integer.

Description

L2BIN() is a low-level file function used with FWRITE() to write CA-Clipper numeric values to a binary file. This function is like I2BIN() which formats a CA-Clipper numeric to a 16-bit binary value.

L2BIN() is the inverse function of BIN2L().

Examples

- This example creates a new binary file, and then writes a series of numbers to the files using L2BIN() to convert the numeric value to 32-bit binary form:

```
#include "Fileio.ch"
//
LOCAL nNumber, nHandle
nHandle := FCREATE("MyFile", FC_NORMAL)
FOR nNumber := 1 TO 100
    FWRITE(nHandle, L2BIN(nNumber) + CHR(0))
NEXT
FCLOSE(nHandle)
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also

BIN2I(), BIN2L(), BIN2W(), CHR(), FWRITE(), I2BIN()

LABEL FORM command

Display labels to the console

Syntax

```
LABEL FORM <xcLabel>
   [TO PRINTER] [TO FILE <xcFile>] [NOCONSOLE]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SAMPLE]
```

Arguments

<xcLabel> is the name of the label (.lbl) file that contains the FORM definition of the LABEL and can be specified either as a literal file name or as a character expression enclosed in parentheses. If an extension is not specified .lbl is assumed.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output to *<xcFile>*. Specify *<xcFile>* as a literal file name or as a character expression enclosed in parentheses. If an extension is not specified, .txt is added.

NOCONSOLE suppresses all LABEL FORM output to the console. If not specified, output automatically displays to the console unless SET CONSOLE is OFF.

<scope> is the portion of the current database file to display labels. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies to LABEL FORM, the conditional set of records within the given scope.

SAMPLE displays test labels as rows of asterisks. Each test label has the same number of columns and rows as the label definition. Then, following each test label display, is the query, "Do you want more samples?" Answering "Y" forces another test label display. Answering "N" causes LABEL FORM to display the actual labels for the specified scope and condition.

Description

LABEL FORM is a console command that sequentially accesses records in the current work area, displaying labels using a definition stored in a .lbl file. Create the .lbl using RL.EXE or by dBASE III PLUS. Refer to the "Report and Label Utility" chapter in the *Programming and Utilities Guide* for more information about creating label definitions.

When invoked, output is sent to the screen and, optionally, to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before invocation of LABEL FORM or use the NOCONSOLE keyword.

When invoked, LABEL FORM searches the current SET PATH drive and directory, if the <*xcLabel*> file is not found in the current directory and the path is not specified.

Notes

- ***Interrupting LABEL FORM:*** To interrupt a LABEL FORM, use INKEY() as a part of the FOR condition to test for an interrupt key press. See the example below.
- ***Printer margin:*** LABEL FORM obeys the current SET MARGIN for output echoed to the printer.

Examples

- This example prints a set of labels and writes them to a file with a single command. Two forms of the command are shown:

```
LOCAL cLabel := "Sales", cFile := "Sales"
USE Sales INDEX Sales NEW
LABEL FORM Sales TO PRINTER TO FILE Sales
LABEL FORM (cLabel) TO PRINTER TO FILE (cFile)
```

- This example interrupts LABEL FORM using INKEY() to test if the user pressed the Esc key:

```
#define K_ESC  27
USE Sales INDEX Sales NEW
LABEL FORM Sales WHILE INKEY() != K_ESC
```

Files

Library is CLIPPER.LIB.

See Also

REPORT FORM, SET PRINTER

LASTKEY() function

Return the INKEY() value of the last key extracted from the keyboard buffer

Syntax

```
LASTKEY() → nInkeyCode
```

Returns

LASTKEY() returns an integer value from -39 to 386 for keyboard events and integer values from 1001 to 1007 for mouse events. This value identifies either the key extracted from the keyboard buffer or the mouse event that last occurred. If the keyboard buffer is empty, and no mouse events are taking place, LASTKEY() returns 0. LASTKEY() returns values for all ASCII characters, function, Alt+Function, Alt+Letter, and Ctrl+Letter key combinations.

Description

LASTKEY() is a function that reports the INKEY() value of the last key fetched from the keyboard buffer by the INKEY() function, or the next mouse event, or a wait state such as ACCEPT, INPUT, READ, WAIT, ACHOICE(), DBEDIT(), or MEMOEDIT(). The time LASTKEY() waits is based on the operating system clock and is not related to the microprocessor speed. LASTKEY() retains its current value until another key is fetched from the keyboard buffer.

LASTKEY() has a number of uses which include:

- Determining the key that terminates a READ
- Determining the key that exits the current Get object within a user-defined function, invoked by a VALID clause
- Identifying an exception key in the user function of ACHOICE(), DBEDIT(), or MEMOEDIT()

LASTKEY() is also used with UPDATED() to determine if any Get object's buffer was changed during a READ.

LASTKEY() is related to NEXTKEY() and READKEY(). NEXTKEY() reads the current key pending in the keyboard buffer without removing it. Use NEXTKEY() instead of INKEY() when polling for a key.

For a complete list of INKEY() codes and Inkey.ch constants for each key, refer to the *Error Messages and Appendices Guide.*

Examples

- This example illustrates a typical application of LASTKEY() to test the key that exits a READ. If the user exits with any key other than Esc and a GET was changed, the specified database file is updated:

```
#include "Inkey.ch"
//
USE Customer NEW
MEMVAR->balance = Customer->Balance
@ 10, 10 SAY "Current Balance" GET MEMVAR->balance
READ
//
IF (LASTKEY() != K_ESC) .AND. UPDATED()
   REPLACE Customer->Balance WITH MEMVAR->balance
ENDIF
```

Files Library is CLIPPER.LIB, header file is Inkey.ch.

See Also CHR(), INKEY(), KEYBOARD, NEXTKEY()

LASTREC() function

Determine the number of records in the current .dbf file

Syntax

```
LASTREC() | RECCOUNT()* → nRecords
```

Returns

LASTREC() returns the number of physical records in the current database file as an integer numeric value. Filtering commands such as SET FILTER or SET DELETED have no effect on the return value. LASTREC() returns zero if there is no database file in USE in the current work area.

Description

LASTREC() is a database function that determines the number of physical records in the current database file. LASTREC() is identical to RECCOUNT() which is supplied as a compatibility function.

By default, LASTREC() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Note: Although the functionality of RECNO() has been expanded to encompass the concept of "identity," the LASTREC() function continues to return only record numbers—not identities. LASTREC() has no expanded functionality, so it is not "identity-aware."

Examples

- This example illustrates the relationship between LASTREC(), RECCOUNT(), and COUNT:

```
USE Sales NEW
? LASTREC(), RECCOUNT()          // Result: 84 84
//
SET FILTER TO Salesman = "1001"
COUNT TO nRecords
? nRecords, LASTREC()            // Result: 14 84
```

- This example uses an aliased expression to access the number of records in a open database file in an unselected work area:

```
USE Sales NEW
USE Customer NEW
? LASTREC(), Sales->(LASTREC())
```

Files Library is CLIPPER.LIB.

See Also COUNT, EOF()

LEFT() function

Extract a substring beginning with the first character in a string

Syntax

```
LEFT(<cString>, <nCount>) → cSubString
```

Arguments

<cString> is a character string from which to extract characters. The maximum size of <*cString*> is 65,535 (64K) bytes.

<nCount> is the number of characters to extract.

Returns

LEFT() returns the leftmost <*nCount*> characters of <*cString*> as a character string. If <*nCount*> is negative or zero, LEFT() returns a null string (""). If <*nCount*> is larger than the length of the character string, LEFT() returns the entire string.

Description

LEFT() is a character function that returns a substring of a specified character string. It is the same as SUBSTR(<*cString*>, 1, <*nCount*>). LEFT() is also like RIGHT(), which returns a substring beginning with the last character in a string.

LEFT(), RIGHT(), and SUBSTR() are often used with both the AT() and RAT() functions to locate the first and/or the last position of a substring before extracting it.

Examples

- This example extracts the first three characters from the left of the target string:

```
? LEFT("ABCDEF", 3)                    // Result: ABC
```

- This example extracts a substring from the beginning of a string up to the first occurrence of a comma:

```
LOCAL cName := "James, William"
? LEFT(cName, AT(",", cName) - 1)   // Result: James
```

Files

Library is CLIPPER.LIB.

See Also

AT(), LTRIM(), RAT(), RIGHT(), RTRIM(), STUFF(), SUBSTR()

LEN() function

Return the length of a character string or the number of elements in an array

Syntax

```
LEN(<cString> | <aTarget>) → nCount
```

Arguments

<cString> is the character string to count.

<aTarget> is the array to count.

Returns

LEN() returns the length of a character string or the number of elements in an array as an integer numeric value. If the character string is a null string ("") or the array is empty, LEN() returns zero.

Description

LEN() is a character and array function that returns the length of a character string or the number of elements in an array. With a character string, each byte counts as one, including an embedded null byte (CHR(0)). By contrast, a null string ("") counts as zero.

For an array, LEN() returns the number of elements. If the array is multidimensional, subarrays count as one element. This means that the LEN() of a nested or multidimensional array simply returns the length of the first dimension. To determine the number of elements in other dimensions, use LEN() on the subarrays as shown in the example below. Note that nested arrays in CA-Clipper need not have uniform dimensions.

Examples

- These examples demonstrate LEN() with various arguments:

```
? LEN("string of characters")        // Result: 20
? LEN("")                            // Result: 0
? LEN(CHR(0))                        // Result: 1
//
LOCAL aTest[10]
? LEN(aTest)                         // Result: 10
```

- This example creates a literal two-dimensional array, and then returns the number of elements in the subarray contained in the first element of the original array:

```
LOCAL aArray := { {1, 2}, {1, 2}, {1, 2} }
? LEN(aArray)                        // Result: 3
? LEN(aArray[1])                     // Result: 2
```

- This example navigates a multidimensional array using LEN():

```
LOCAL aArray := { {1, 2}, {1, 2}, {1, 2} }
LOCAL nRow, nColumn, nRowCount, nColumnCount
//
nRowCount = LEN(aArray)
FOR nRow = 1 TO nRowCount
    nColumnCount = LEN(aArray[nRow])
    FOR nColumn = 1 TO nColumnCount
       ? nRow, nColumn, aArray[nRow][nColumn]
    NEXT
NEXT
```

- In this example a function returns an array of numeric values that describe the dimensions of a nested or multidimensional array. The function assumes that the array has uniform dimensions:

```
FUNCTION Dimensions( aArray )
    LOCAL aDims := {}
    DO WHILE ( VALTYPE(aArray) == "A" )
       AADD( aDims, LEN(aArray) )
       aArray := aArray[1]
    ENDDO
    RETURN (aDims)
```

Files Library is CLIPPER.LIB.

See Also AADD(), ASIZE(), LTRIM(), RTRIM()

LIST command

List records to the console

Syntax

```
LIST <exp list>
   [TO PRINTER] [TO FILE <xcFile>]
   [<scope>] [WHILE <lCondition>]
   [FOR <lCondition>] [OFF]
```

Arguments

<exp list> is the list of expressions to be evaluated and displayed for each record processed.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output to the specified file name and can be specified either as a literal file name or as a character expression enclosed in parentheses. If an extension is not specified, .txt is added.

<scope> is the portion of the current database file to LIST. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to LIST within the given scope.

OFF suppresses the display of record numbers.

Description

LIST is a console command that sequentially accesses records in the current work area, displaying the results of one or more expressions for each record accessed. The output is in tabular format with each column separated by a space. LIST is identical to DISPLAY with the exception that its default scope is ALL rather than NEXT 1.

When invoked, output is sent to the screen and, optionally, to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the LIST invocation.

Notes

- ***Interrupting LIST:*** So the user may interrupt a LIST, use INKEY() as part of the FOR condition to test for an interrupt key press. See the example below.
- ***Printer margin:*** LIST honors the current SET MARGIN for output echoed to the printer.

Examples

- In this example, a simple list is followed by a conditional list to the printer:

```
USE Sales
LIST DATE(), TIME(), Branch
LIST Branch, Salesman FOR Amount > 500 TO PRINTER
```

- This example interrupts LIST using INKEY() to test whether the user pressed the Esc key:

```
#define K_ESC    27
USE Sales INDEX Salesman NEW
LIST Branch, Salesman, Amount WHILE INKEY() != K_ESC
```

Files Library is CLIPPER.LIB.

See Also ?|??, DISPLAY

ListBox class

Create a list box

Description

A list box displays a list of strings (or *items*) to the user. You can use the methods of ListBox to add, arrange, remove, and interrogate the items in a list box.

Class Function

```
ListBox( <nTop>, <nLeft>, <nBottom>, <nRight>
   [, <lDropDown>] ) → oListBox
```

Arguments

<nTop> is a numeric value that indicates the top screen row of the list box.

<nLeft> is a numeric value that indicates the left screen column of the list box.

<nBottom> is a numeric value that indicates the bottom screen row of the list box.

<nRight> is a numeric value that indicates the right screen column of the list box.

<lDropDown> is an optional logical value that indicates whether the list box is a drop-down list box. A value of true (.T.) indicates that it is a drop-down list box; otherwise, a value of false (.F.) indicates that it is not. The default is false (.F.).

Returns

Returns a ListBox object when all of the required arguments are present; otherwise, ListBox() returns NIL.

Exported Instance Variables

`bitmap` (Assignable)

Contains a character string that indicates a bitmap file to be displayed on the button. Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and, if it is not found, search in the bitmap library second. If no file is found either on disk or in the library, no bitmap will be displayed. This instance variable only affects applications running in graphic mode and is ignored in text mode.

`bottom` (Assignable)

Contains a numeric value that indicates the bottommost screen row where the list box is displayed.

`buffer`

Contains a numeric value that indicates the position in the list of the selected item.

`capCol` (Assignable)

Contains a numeric value that indicates the screen column where the list box's caption is displayed.

`capRow` (Assignable)

Contains a numeric value that indicates the screen row where the list box's caption is displayed.

`caption` (Assignable)

Contains a character string that concisely describes the list box on the screen.

When present, the & character specifies that the character immediately following it in the caption is the list box's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to a list box. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

`cargo` (Assignable)

Contains a value of any type that is ignored by the ListBox object. ListBox:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a ListBox object and retrieved later.

`coldBox` (Assignable)

Contains an optional string that specifies the characters to use when drawing a box around the list box when it does not have input focus. Its default value is a single line box.

Standard Box Types

Constant	Description
B_SINGLE	Single line box
B_DOUBLE	Double line box
B_SINGLE_DOUBLE	Single line top/bottom, double line sides
B_DOUBLE_SINGLE	Double line top/bottom, single line sides

Box.ch contains manifest constants for the ListBox:coldBox value.

`colorSpec` (Assignable)

Contains a character string that indicates the color attributes that are used by the list box's Display() method. If the list box is a drop-down list, the string must contain eight color specifiers, otherwise it must contain seven color specifiers.

Note: The background colors of the ListBox Color Attributes are ignored in graphic mode.

ListBox Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	List box items that are not selected when the list does not have input focus	Unselected
2	The selected list box item when the list does not have input focus	Unselected
3	List box items that are not selected when the list has input focus	Unselected
4	The selected list box item when the list has input focus	Enhanced
5	The list box's border	Border
6	The list box's caption	Standard
7	The list box caption's accelerator key	Background
8	The list box's drop-down button	Standard

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box in the item properties.

`dropDown`

Contains an optional logical value that indicate whether the object is a drop-down list. A value of true (.T.) indicates that it is a drop-down list; otherwise, a value of false (.F.) indicates that it is not. The default is false.

`fBlock` `(Assignable)`

Contains an optional code block that, when present, is evaluated each time the ListBox object receives or loses input focus. The code block takes no implicit arguments. Use the ListBox:hasFocus variable to determine if the list box is receiving or losing input focus. A value of true (.T.) indicates that it is receiving input focus; otherwise, a value of false (.F) indicates that it is losing input focus.

This code block is included in the ListBox class to provide a method of indicating when an input focus change event has occurred. The name "fBlock" refers to focus block.

`hasFocus`

Contains a logical value that indicates whether the ListBox object has input focus. ListBox:hasFocus contains true (.T.) if it has input focus; otherwise, it contains false (.F.).

`hotBox` `(Assignable)`

Contains an optional string that specifies the characters to use when drawing a box around the list box when it has input focus. Its default value is a double-line box.

Standard Box Types

Constant	Description
B_SINGLE	Single line box
B_DOUBLE	Double line box
B_SINGLE_DOUBLE	Single line top/bottom, double line sides
B_DOUBLE_SINGLE	Double line top/bottom, single line sides

Box.ch contains manifest constants for the ListBox:hotBox value.

`isOpen`

Contains a logical value that indicates whether the list is visible. A value of true (.T.) indicates that the list is visible; otherwise a value of false (.F.) indicates that it is not visible. When ListBox:dropDown is false (.F.), ListBox:isOpen is always true (.T.); otherwise, ListBox:isOpen is true (.T.) during the period of time between calling ListBox:open() and ListBox:close().

`itemCount`

Contains a numeric value that indicates the total number of items contained within the ListBox object.

`left` (Assignable)

Contains a numeric value that indicates the leftmost screen column where the list box is displayed.

`message` (Assignable)

Contains a character string that describes the list box. It is displayed on the screen's status bar line.

`right` (Assignable)

Contains a numeric value that indicates the rightmost screen column where the list box is displayed.

`sBlock` (Assignable)

Contains an optional code block that, when present, is evaluated immediately after the ListBox object's selection changes. The code block takes no implicit arguments. Use the ListBox:buffer variable to determine the current selection.

This code block is included in the ListBox class to provide a method of indicating when a state change event has occurred. The name "sBlock" refers to state block.

`top` (Assignable)

Contains a numeric value that indicates the topmost screen row where the list box is displayed.

`topItem` (Assignable)

Contains a numeric value that indicates the position in the list box of the first visible item.

`typeOut`

Contains a logical value that indicates whether the list contains any items. A value of true (.T.) indicates that the list contains selectable items; otherwise, a value of false (.F.) indicates that the list is empty.

`vScroll` (Assignable)

Contains an optional ScrollBar object whose orientation must be vertical. The scroll bar thumb position reflects the relationship between the current first visible item on the screen and the total number of possible top items (total number of items - number of visible items - 1).

When present, the scroll bar is automatically integrated within the behaviors of the following ListBox object methods: addItem(), display(), delItem(), hitTest(), insItem(), nextItem(), prevItem(), and select().

Exported Methods

`<oListBox>:addItem( <cText>[, <expValue>] ) → self`

<cText> is a character string that indicates the item's text to display in the list.

<expValue> is a value that is associated with the list item. If omitted, the item's associated data will be NIL.

addItem() is a method of the ListBox class that is used for appending a new item to a list. When adding an item, an optional value may be included. This enables you to associate pertinent data with the text displayed in the list.

Note: When present, the scroll bar is automatically updated to reflect the addition of the new item.

`<oListBox>:close() → self`

close() is a method of the ListBox class that is used for restoring the screen under a drop-down list box.

`<oListBox>:delItem( nPosition> ) → self`

*<nPosition>*is a numeric value that indicates the position in the list of the item to delete.

delItem() is a method of the ListBox class that is used for removing an item from a list. ListBox:buffer is automatically adjusted when an item is deleted while the last item in the list is selected.

Note: When present, the scroll bar is automatically updated to reflect the deletion of the item.

```
<oListBox>:display() → self
```

display() is a method of the ListBox class that is used for showing a list and its caption on the screen. display() uses the values of the following instance variables to correctly show the list in its current context, in addition to providing maximum flexibility in the manner a list box appears on the screen: bottom, capCol, capRow, caption, coldBox, colorSpec, hasFocus, hotBox, itemCount, left, right, style, top, topItem, and vScroll.

Note: When present, the scroll bar is automatically displayed when the ListBox:display() method is called.

```
<oListBox>:findText( <cText> [,<nPosition>]
   [,<lCaseSensitive>] [,<lExact>] ) → nPosition
```

<cText>is a character string that indicates the text that is being searched for.

<nPosition> is an optional numeric value that indicates the starting position in the list of the search. The default is 1.

<lCaseSensitive> is an optional logical value that indicates whether the search should be case sensitive. Set *<lCaseSensitive>* to true (.T.) to indicate that the search should be case sensitive; otherwise, set *<lCaseSensitive>* to false (.F.). The default is true (.T.).

<lExact> is an optional logical value that indicates whether the search enforces an exact comparison including length and trailing characters. A value of true (.T.) indicates that findText() searches for an exact match; otherwise, a value of false (.F.) indicates that it should not. The default is the current setExact() setting.

Returns a numeric value that indicates the position in the list of the first item from *<nPosition>* whose text matches *<cText>*, or 0 if ListBox:findText() is unsuccessful.

findText() is a method of the ListBox class that is used for determining whether an item is a member of a list and its position within the list. findText() always searches from *<nPosition>* to the end of the list and, when necessary, continues from the beginning of the list to *<nPosition>* - 1.

```
<oListBox>:getData( <nPosition> ) → expValue
```

<nPosition> is a numeric value that indicates the position within the list of the item whose data is being retrieved.

Returns the data associated with the item in the list specified by *<nPosition>*.

getData() is a method of the ListBox class that is used for retrieving the data portion of a list box item.

```
<oListBox>:getItem( <nPosition> ) → aItem
```

<nPosition> is a numeric value that indicates the position in the list of the item that is being retrieved.

Returns the list box item specified by *<nPosition>* expressed as a two-element array consisting of the item's text and data respectively. The text is a character string that indicates the item's text to display in the list. The data is a value that is associated with the list item.

getItem() is a method of the ListBox class that is used for retrieving a list box item.

```
<oListBox>:getText( <nPosition> ) → cText
```

<nPosition> is a numeric value that indicates the position within the list of the item whose text is being retrieved.

Returns the text associated with the item in the list specified by *<nPosition>*.

getText() is a method of the ListBox class that is used for retrieving the text portion of a list box item.

```
<oListBox>:hitTest( <nMouseRow>, <nMouseCol> )
    → nhitStatus
```

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the list box.

Applicable Hit Test Return Values

Value	Constant	Description
> 0	Not Applicable	The position in the list of the item whose region the mouse is within
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the list box occupies
-1	HTTOPLEFT	The mouse cursor is on the top left corner of the list box's border
-2	HTTOP	The mouse cursor is on the list box's top border
-3	HTTOPRIGHT	The mouse cursor is on the top right corner of the list box's border
-4	HTRIGHT	The mouse cursor is on the list box's right border
-5	HTBOTTOMRIGHT	The mouse cursor is on the bottom right corner of the list box's border
-6	HTBOTTOM	The mouse cursor is on the list box's bottom border
-7	HTBOTTOMLEFT	The mouse cursor is on the bottom left corner of the list box's border
-8	HTLEFT	The mouse cursor is on the list box's left border
-1025	HTCAPTION	The mouse cursor is on the list box's caption
-4097	HTDROPBUTTON	The mouse cursor is on the list box's drop down button.

Button.ch contains manifest constants for the ListBox:hitTest() return value.

hitTest() is a method of the ListBox class that is used for determining if the mouse cursor is within the region of the screen that the list box occupies.

Note: When a scroll bar is present, its hitTest() method is automatically called to determine if the mouse cursor is within its region. If ScrollBar:hitTest() succeeds in determining that a "hit" has been achieved, ListBox:hitTest() returns the appropriate scroll bar hit test return code.

```
<oListBox>:insItem( <nPosition>, <cText>,
   [<expValue>]) → self
```

<nPosition> is a numeric value that indicates the position at which the new item is inserted.

<c**Text**> is the item's text to display in the list.

<expValue> is a value that is associated with the list item. If omitted, the item's associated data will be NIL.

insItem() is a method of the ListBox class that is used for inserting a new item to a list. When inserting an item, an optional value may be included. This enables you to associate pertinent data with the text displayed in the list.

Note: When present, the scroll bar is automatically updated to reflect the insertion of the new item.

```
<oListBox>:killFocus() → self
```

killFocus() is a method of the ListBox class that is used for taking input focus away from a ListBox object. Upon receiving this message, the ListBox object redisplays itself and, if present, evaluates the code block within its fBlock variable.

This message is meaningful only when the ListBox object has input focus.

```
<oListBox>:nextItem() → self
```

nextItem() is a method of the ListBox class that is used for changing the selected item from the current item to the one immediately following it. If necessary, nextItem() will call its display() method to ensure that the newly selected item is visible.

This message is meaningful only when the ListBox object has input focus.

Note: When present, the scroll bar is automatically updated to reflect the new first visible item in the list when the call to ListBox:nextItem() causes the list to be scrolled.

```
<oListBox>:open() → self
```

open() is a method of the ListBox class that is used for saving the screen under a drop-down list box and displaying the list.

```
<oListBox>:prevItem() → self
```

prevItem() is a method of the ListBox class that is used for changing the selected item from the current item to the one immediately before it. If necessary, prevItem() will call its display() method to ensure that the newly selected item is visible.

This message is meaningful only when the ListBox object has input focus.

Note: When present, the scroll bar is automatically updated to reflect the new first visible item in the list when the call to ListBox:prevItem() causes the list to be scrolled.

```
<oListBox>:scroll( <nMethod> ) → self
```

<nMethod> indicates the manner in which the scroll operation is carried out. A call to ListBox:scroll() is typically issued in response to ScrollBar:hitTest returning that a hit has been established.

ListBox:Scroll Argument Values

Value	Constant	Result
-3074	HTSCROLLUNITDEC	Scroll down one line
-3075	HTSCROLLUNITINC	Scroll up one line
-3076	HTSCROLLBLOCKDEC	Scroll down one window
-3077	HTSCROLLBLOCKINC	Scroll up one window

scroll() is a method of the ListBox class that is used for scrolling the contents of a list box up or down.

```
<oListBox>:select( <nPosition> ) → self
```

<nPosition> is a numeric value that indicates the position in the list of the item to select.

select() is a method of the ListBox class that is used for changing the selected item in a list. Its state is typically changed when one of the arrow keys is pressed or the mouse's left button is pressed when its cursor is within the ListBox object's screen region. If necessary, select() will call its display() method to ensure that the newly selected item is visible.

Note: When present, the scroll bar is automatically updated to reflect the new first visible item in the list when the call to ListBox:select() causes the list to be scrolled.

`<oListBox>:setData( <nPosition>, <expValue> ) → self`

<nPosition> is a numeric value that indicates the position of the item within the list whose data is being replaced.

<expValue> is the data that is being associated with the item specified by *<nPosition>*.

setData() is a method of the ListBox class that is used for changing the data that is associated with a list item.

`<oListBox>:setFocus() → self`

setFocus() is a method of the ListBox class that is used for giving focus to a ListBox object. Upon receiving this message, the ListBox object redisplays itself and, if present, evaluates the code block within its fBlock variable.

This message is meaningful only when the ListBox object does not have input focus.

`<oListBox>:setItem( <nPosition>, <aItem> ) → self`

<nPosition> is a numeric value that indicates the position of the item within the list which is being replaced.

<aItem> is the new list box item expressed as a two-element array consisting of the item's text and data respectively.

setItem() is a method of the ListBox class that is used for replacing an item in a list.

`<oListBox>:setText( <nPosition>, <cText> ) → self`

<nPosition> is a numeric value that indicates the position of the item within the list whose text is being replaced.

<cText> is a character string that indicates the new text for the item specified by *<nPosition>*.

setText() is a method of the ListBox class that is used for changing the text that is associated with a list item.

Examples

- This example shows how the user can select the screen's foreground color:

```
function SelectFore()
local cOldWindow, oList, nChoice
memvar GetList
    cOldWindow := SaveScreen( 5, 10, 12, 20 )
    oList := ListBox( 5, 10, 12, 20, { | x | iif( x == NIL,;
                     nChoice,    nChoice  := x } )
    oList:AddItem( "Black", "N" )
    oList:AddItem( "Blue", "B" )
    oList:AddItem( "Green", "G" )
    oList:AddItem( "Cyan", "BG" )
    oList:AddItem( "Red", "R" )
    oList:AddItem( "Magenta", "RB" )
    oList:AddItem( "Brown", "GR" )
    oList:AddItem( "White", "W" )
    aAdd( GetList, oList )
    read
    RestScreen( 5, 10, 12, 20, cOldWindow )
return ( oList:GetData( nChoice ) )
```

See Also SETEXACT()

LOCAL statement

Declare and initialize local variables and arrays

Syntax

```
LOCAL <identifier> [[:= <initializer>], ... ]
```

Arguments

<identifier> is the name of a variable or array to declare local. If the *<identifier>* is followed by square brackets ([ ]), it is created as an array. If the *<identifier>* is an array, the syntax for specifying the number of elements for each dimension can be array[*<nElements>*, *<nElements2>*,...] or array[*<nElements>*][*<nElements2>*]... The maximum number of elements per dimension is 4096. The maximum number of dimensions per array is limited only by available memory.

<initializer> is the optional assignment of a value to a new local variable. Array identifiers, however, cannot be given values with an *<initializer>*. An *<initializer>* for a local variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression including a literal array. If no explicit *<initializer>* is specified, the variable is given an initial value of NIL. In the case of an array, each element is NIL.

Note: The macro operator (&) cannot be used in a LOCAL declaration statement.

Description

LOCAL is a declaration statement that declares one or more variables or arrays local to the current procedure or user-defined function, and must occur before any executable statement including PRIVATE, PUBLIC, and PARAMETERS. Local variable declarations hide all inherited private variables and visible public variables with the same name. A LOCAL statement, however, that declares a variable name which is already declared causes a fatal compiler error and no object file (.OBJ) is generated. This error can happen as a result of two declarations for the same variable name in the same routine, or as the result of redeclaring a variable with filewide scope. Declaration statements include FIELD, MEMVAR, and STATIC.

Local variables are visible only within the current procedure or user-defined function and, unlike private variables, are not visible within invoked routines. Local variables are created automatically each time the procedure in which they were declared begins executing. They continue to exist and retain their values until the declaring procedure or user-defined function returns control to the code that invoked it. If a procedure or user-defined function is invoked recursively (calls itself), each recursive activation creates a new set of local variables.

The initial value of local variables and array elements is NIL if not explicitly initialized, either in the initializer list or by assignment. The initializer expression can be any valid CA-Clipper expression, including function calls. Note that an array declaration cannot have an initializer.

The maximum number of local variables in a program is limited only by available memory. Arrays, however, assigned to a local variable are still limited to 4096 elements per dimension.

For more information on variable declarations and scoping, refer to the Variables section in the "Basic Concepts" chapter of the *Programming and Utilities Guide*.

Notes

- ***Inspecting local variables within the debugger:*** To access local variable names within the CA-Clipper DOS-level debugger, you must compile program (.prg) files using the /B option so that local variable information is included in the object file.

- ***Local parameters:*** Declare a list of local parameters as a part of a FUNCTION or PROCEDURE declaration by enclosing the list of parameters in parentheses following the <*idFunction*>:

  ```
  FUNCTION <idFunction>(<idParam list>)
  ```

  Declaration of local parameters supersedes creation of private parameters with the PARAMETERS statement.

- ***Macro expressions:*** You cannot refer to local variables within macro variables and expressions. If you refer to a local variable within a macro variable, a private or public variable with the same name will be referenced instead. If no such variable exists, a runtime error will be generated.

- ***Memory files:*** Local variables cannot be SAVED to or RESTOREd from memory (.mem) files.
- ***Type of a local variable:*** Since TYPE() uses the macro operator (&) to evaluate its argument, it cannot be used to determine the type of a local or static variable or an expression containing a local or static variable reference. The VALTYPE() function provides this facility. VALTYPE() evaluates its argument and returns the type of the return value.

Examples

- This example declares two local arrays and two local variables:

```
LOCAL aArray1[20, 10], aArray2[20][10], var1, var2
```

- This example declares two local variables with initializers. The first is initialized to a date value and the second to a literal array:

```
LOCAL dWhen := DATE()
LOCAL aVegies := {"Tomato", "Chickadee", "Butterbean"}
```

See Also

FUNCTION, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, STATIC

LOCATE command

Search sequentially for a record matching a condition

Syntax

```
LOCATE [<scope>] FOR <lCondition>
   [WHILE <lCondition>]
```

Arguments

<scope> is the portion of the current database file in which to perform the LOCATE. The default scope is ALL records.

FOR <lCondition> specifies the next record to LOCATE within the given scope.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

Description

LOCATE is a database command that searches for the first record in the current work area that matches the specified conditions and scope. When you first execute a LOCATE, it searches from the beginning record of the scope for the first matching record in the current work area. It terminates when a match is found or the end of the LOCATE scope is reached. If it is successful, the matching record becomes the current record and FOUND() returns true (.T.). If it is unsuccessful, FOUND() returns false (.F.) and the positioning of the record pointer depends on the controlling scope of the LOCATE.

Each work area can have its own LOCATE condition. The condition remains active until you execute another LOCATE command in that work area or the application terminates.

LOCATE works with CONTINUE. Once a LOCATE has been issued, you can resume the search from the current record pointer position with CONTINUE. There are, however, some exceptions. See the following note.

Notes

- ***CONTINUE:*** Both the <*scope*> and the WHILE condition apply only to the initial LOCATE and are not operational for any subsequent CONTINUE commands. To continue a pending LOCATE with a scope or WHILE condition, use SKIP then LOCATE REST WHILE <*lCondition*> instead of CONTINUE.

Examples

- These examples show typical LOCATEs:

```
USE Sales INDEX Salesman
LOCATE FOR Branch = "200"
? FOUND(), EOF(), RECNO()        // Result: .T. .F. 5
LOCATE FOR Branch = "5000"
? FOUND(), EOF(), RECNO()        // Result: .F. .T. 85
```

- This example shows a LOCATE with a WHILE condition that is continued by using LOCATE REST:

```
SEEK "Bill"
LOCATE FOR Branch = "200" WHILE Salesman = "Bill"
DO WHILE FOUND()
    ? Branch, Salesman
    SKIP
    LOCATE REST FOR Branch = "200" WHILE ;
              Salesman = "Bill"
ENDDO
```

Files

Library is CLIPPER.LIB.

See Also

CONTINUE, EOF(), FOUND(), SEEK, SET FILTER

LOG() function

Calculate the natural logarithm of a numeric value

Syntax

```
LOG(<nExp>) → nNaturalLog
```

Arguments

<nExp> is a numeric value greater than zero to be converted to its natural logarithm.

Returns

LOG() returns the natural logarithm as a numeric value. If *<nExp>* is less than or equal to zero, LOG() returns a numeric overflow (displayed as a row of asterisks).

Description

LOG() is a numeric function that calculates the natural logarithm of a number and is the inverse of EXP(). The natural logarithm has a base of e which is approximately 2.7183. The LOG() function returns x in the following equation,

```
e**x = y
```

where y is the numeric expression used as the LOG() argument (i.e., LOG(y) = x). Due to mathematical rounding, the values returned by LOG() and EXP() may not agree exactly (i.e., EXP(LOG(x)) may not always equal x).

Examples

- These examples demonstrate various results of LOG():

```
? LOG(10)                  // Result: 2.30
? LOG(10 * 2)              // Result: 3.00
? EXP(LOG(1))              // Result: 1.00
? LOG(2.71)                // Result: 1.00
```

- This example is a user-defined function that returns the base 10 logarithm:

```
FUNCTION Log10( nNumber )
IF nNumber > 0
    RETURN LOG(nNumber)/LOG(10)
ELSE
    RETURN NIL
ENDIF
```

Files Library is CLIPPER.LIB.

See Also EXP(), SET DECIMALS, SET FIXED

LOWER() function

Convert uppercase characters to lowercase

Syntax

LOWER(<*cString*>) → *cLowerString*

Arguments

<*cString*> is a character string to be converted to lowercase.

Returns

LOWER() returns a copy of <*cString*> with all alphabetic characters converted to lowercase. All other characters remain the same as in the original string.

Description

LOWER() is a character function that converts uppercase and mixed case strings to lowercase. It is related to UPPER() which converts lowercase and mixed case strings to uppercase. LOWER() is related to the ISLOWER() and ISUPPER() functions which determine whether a string begins with a lowercase or uppercase letter.

LOWER() is generally used to format character strings for display purposes. It can, however, be used to normalize strings for case-independent comparison or INDEXing purposes.

Examples

- These examples demonstrate various results of LOWER():

```
? LOWER("STRING")              // Result: string
? LOWER("1234 CHARS = ")       // Result: 1234 chars =
```

Files

Library is CLIPPER.LIB.

See Also

ISLOWER(), ISUPPER(), UPPER()

LTRIM() function

Remove leading spaces from a character string

Syntax

`LTRIM(<cString>) → cTrimString`

Arguments

<cString> is the character string to copy without leading spaces.

Returns

LTRIM() returns a copy of *<cString>* with the leading spaces removed. If *<cString>* is a null string ("") or all spaces, LTRIM() returns a null string ("").

Description

LTRIM() is a character function that formats character strings with leading spaces. These can be, for example, numbers converted to character strings using STR().

LTRIM() is related to RTRIM(), which removes trailing spaces, and ALLTRIM(), which removes both leading and trailing spaces. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADR(), and PADL() functions which center, right-justify, or left-justify character strings by padding them with fill characters.

Notes

- ***Space characters:*** The LTRIM() function treats carriage returns, line feeds, and tabs as space characters and removes these as well.

Examples

- These examples illustrate LTRIM() used with several other functions:

```
nNumber = 18
? STR(nNumber)                  // Result: 18
? LEN(STR(nNumber))             // Result: 10
? LTRIM(STR(nNumber))           // Result: 18
? LEN(LTRIM(STR(nNumber)))      // Result:  2
```

Files Library is CLIPPER.LIB.

See Also ALLTRIM(), PAD(), RTRIM(), STR(), SUBSTR(), TRIM()

LUPDATE() function

Return the last modification date of a database (.dbf) file

Syntax

```
LUPDATE() → dModification
```

Returns

LUPDATE() returns the date of the last change to the open database file in the current work area. If there is no database file in USE, LUPDATE() returns a blank date.

Description

LUPDATE() is a database function that determines the date the database file in the current work area was last modified and CLOSEd. By default, LUPDATE() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression, as shown in the example below.

Examples

- This example demonstrates that the modification date of the database file is not changed until the database file is closed:

```
? DATE()                    // Result: 09/01/90
USE Sales NEW
? LUPDATE()                 // Result: 08/31/90
//
APPEND BLANK
? LUPDATE()                 // Result: 08/31/90
CLOSE DATABASES
//
USE Sales NEW
? LUPDATE()                 // Result: 09/01/90
```

- This example uses an aliased expression to access LUPDATE() for a database file opened in an unselected work area:

```
USE Sales NEW
USE Customer NEW
? LUPDATE(), Sales->(LUPDATE())
```

Files Library is EXTEND.LIB.

See Also FIELDNAME(), LASTREC(), RECSIZE()

MAX() function

Return the larger of two numeric or date values

Syntax

```
MAX(<nExp1>, <nExp2>) → nLarger
MAX(<dExp1>, <dExp2>) → dLarger
```

Arguments

<nExp1> and ***<nExp2>*** are the numeric values to be compared.

<dExp1> and ***<dExp2>*** are the date values to be compared.

Returns

MAX() returns the larger of the two arguments. The value returned is the same type as the arguments.

Description

MAX() is a numeric and date function that ensures the value of an expression is larger than a specified minimum. The inverse of MAX() is MIN(), which returns the lesser of two numeric or date values.

Examples

- In these examples MAX() returns the greater of two numeric values:

```
? MAX(1, 2)                    // Result: 2
? MAX(2, 1)                    // Result: 2
```

- In these examples MAX() compares date values:

```
? DATE()                              // Result: 09/01/90
? MAX(DATE(), DATE() + 30)            // Result: 10/01/90
? MAX(DATE(), CTOD(""))               // Result: 09/01/90
```

Files

Library is CLIPPER.LIB.

See Also

MIN()

MAXCOL() function

Determine the maximum visible screen column

Syntax

```
MAXCOL() → nColumn
```

Returns

MAXCOL() returns the column number of the rightmost visible column for display purposes.

Description

MAXCOL() is a screen function that determines the maximum visible column of the screen. Row and column numbers start at zero in CA-Clipper.

If you use a C or other extended function to set the video mode, use the SETMODE() function so your CA-Clipper application returns the correct value for MAXCOL().

Examples

- This example uses MAXROW() and MAXCOL() to determine the area in which to draw a box, and then executes DBEDIT() within the box region:

```
CLS
@ 0, 0 TO MAXROW(), MAXCOL() DOUBLE
DBEDIT(1, 1, MAXROW() + 1, MAXCOL()  1)
```

Files

Library is CLIPPER.LIB.

See Also

COL(), MAXROW(), ROW()

MAXROW() function

Determine the maximum visible screen row

Syntax

```
MAXROW() → nRow
```

Returns

MAXROW() returns the row number of the bottommost visible row for display purposes.

Description

MAXROW() is a screen function that determines the maximum visible row of the screen. Row and column numbers start at zero in CA-Clipper.

If you use a C or other extended function to set the video mode, use the SETMODE() function so your CA-Clipper application returns the correct value for MAXCOL().

Examples

- This user-defined function, SCREENSIZE(), uses MAXROW() and MAXCOL() to return an array containing the current screen size:

```
FUNCTION ScreenSize
   RETURN { MAXROW(), MAXCOL() }
```

Files

Library is CLIPPER.LIB.

See Also

COL(), MAXCOL(), ROW()

MCOL() function

Determine the mouse cursor's screen column position

Syntax

`MCOL() → nCurrentMouseColumn`

Returns

MCOL() returns the mouse cursor's current screen column position.

Description

MCOL() is a function that is used for determining the mouse cursor's screen column position. This is useful when implementing a hit testing routine whose purpose is to determine if the mouse cursor is on pertinent information when the left mouse button is pressed.

See Also

MROW()

MDBLCLK() function

Determine the double-click speed threshold of the mouse

Syntax

```
MDBLCLK( [<nNewSpeed>] ) → nSpeed
```

Arguments

<nNewSpeed> is the maximum allowable amount of time between mouse key presses for a double-click to be detected. This is measured in milliseconds.

Returns

MDBLCLK() returns the current double-click speed threshold.

Description

MDBLCLK() is a function used for determining and, optionally, changing the mouse's double-click speed threshold. This is useful when the mouse's double-click sensitivity needs to be adjusted.

MEMOEDIT() function

Display or edit character strings and memo fields

Syntax

```
MEMOEDIT([<cString>],
    [<nTop>], [<nLeft>],
    [<nBottom>], [<nRight>],
    [<lEditMode>],
    [<cUserFunction>],
    [<nLineLength>],
    [<nTabSize>],
    [<nTextBufferRow>],
    [<nTextBufferColumn>],
    [<nWindowRow>],
    [<nWindowColumn>]) → cTextBuffer
```

Arguments

<cString> is the character string or memo field to copy to the MEMOEDIT() text buffer. If not specified, the text buffer is empty.

<nTop>, ***<nLeft>***, ***<nBottom>***, and ***<nRight>*** are the upper-left and lower-right window coordinates. Row values can range from zero to MAXROW(), and column positions can range from zero to MAXCOL(). If not specified, the default coordinates are 0, 0, MAXROW(), and MAXCOL().

<lEditMode> determines whether the text buffer can be edited or merely displayed. Specifying true (.T.) allows the user to make changes to the text buffer, while specifying false (.F.) only allows the user to browse the text buffer. If not specified, the default value is true (.T.).

<cUserFunction> is the name of a user-defined function that executes when the user presses a key not recognized by MEMOEDIT() and when no keys are pending in the keyboard buffer. *<cUserFunction>* is specified as a character value without parentheses or arguments. Specifying false (.F.) for this argument displays *<cString>* and causes MEMOEDIT() to immediately terminate. If this argument is specified, the automatic behavior of MEMOEDIT() changes. Refer to the discussion below.

<nLineLength> determines the length of lines displayed in the MEMOEDIT() window. If a line is greater than *<nLineLength>*, it is word wrapped to the next line in the MEMOEDIT() window. If *<nLineLength>* is greater than the number of columns in the MEMOEDIT() window, the window will scroll if the cursor moves past the window border. If *<nLineLength>* is not specified, the default line length is (*<nRight>* *<nLeft>*).

<nTabSize> determines the tab stops that will be used when the user presses Tab. If *<nTabSize>* is not specified, tab stops will be placed at every four characters.

<nTextBufferRow> and ***<nTextBufferColumn>*** define the display position of the cursor within the text buffer when MEMOEDIT() is invoked. *<nTextBufferRow>* begins with one (1) and *<nTextBufferColumn>* begins with zero (0). If these arguments are not specified, the cursor is placed at row one (1) and column zero (0) of the MEMOEDIT() window.

<nWindowRow> and ***<nWindowColumn>*** define the initial position of the cursor within the MEMOEDIT() window. Row and column positions begin with zero (0). If these arguments are not specified, the initial window position is row zero (0) and the current cursor column position.

Returns

MEMOEDIT() returns the text buffer if the user terminates editing with Ctrl+W or a copy of *<cString>* if user terminates with Esc.

Description

MEMOEDIT() is a user interface and general purpose text editing function that edits memo fields and long character strings. Editing occurs within a specified window region placed anywhere on the screen. Like the other user interface functions (ACHOICE() and DBEDIT()), MEMOEDIT() supports a number of different modes and includes a user function that allows key reconfiguration and other activities relevant to programming the current text editing task.

- ***The text buffer:*** When you invoke MEMOEDIT() and specify *<cString>*, it is copied to the text buffer. The user actually edits the text buffer. If the *<cString>* is not specified, the user is presented with an empty text buffer to edit.

  When the user exits MEMOEDIT() by pressing Ctrl+W, the contents of the text buffer are returned. If the user exits by pressing Esc, the text buffer is discarded and the original *<cString>* value is returned. In either case, the return value can then be assigned to a variable or memo field, or passed as an argument to another function.

- ***Editing modes:*** MEMOEDIT() supports two editing modes depending on the value of *<lEditMode>*. When *<lEditMode>* is true (.T.), MEMOEDIT() enters edit mode and the user can change the contents of the MEMOEDIT() text buffer. When *<lEditMode>* is false (.F.), MEMOEDIT() enters browse mode and the user can only navigate about the text buffer but cannot edit or insert new text. To make browsing easier for the user, the scrolling is disabled so Up arrow and Down arrow scroll the text buffer up or down one line within the MEMOEDIT() window.

- ***Entering and editing text:*** Within MEMOEDIT(), the user can enter and edit text by positioning the cursor, adding, or deleting characters. To facilitate editing the text, there are a number of different navigation and editing keys:

MEMOEDIT() Navigation and Editing Keys

Key	Action
Up arrow/Ctrl+E	Move up one line
Down arrow/Ctrl+X	Move down one line
Left arrow/Ctrl+S	Move left one character
Right arrow/Ctrl+D	Move right one character
Ctrl+Left arrow/Ctrl+A	Move left one word
Ctrl+Right arrow/Ctrl+F	Move right one word
Home	Move to beginning of current line
End	Move to end of current line
Ctrl+Home	Move to beginning of current window
Ctrl+End	Move to end of current window
PgUp	Move to previous edit window
PgDn	Move to next edit window
Ctrl+PgUp	Move to beginning of memo
Ctrl+PgDn	Move to end of memo
Return	Move to beginning of next line
Delete	Delete character at cursor
Backspace	Delete character to left of cursor
Tab	Insert tab character or spaces
Printable characters	Insert character
Ctrl+Y	Delete the current line
Ctrl+T	Delete word right
Ctrl+B	Reform paragraph
Ctrl+V/Ins	Toggle insert mode
Ctrl+W	Finish editing with save
Esc	Abort edit and return original

When the user is entering text, there are two text entry modes, insert and overstrike. When MEMOEDIT() is invoked, the default mode is overstrike. Edit mode changes in MEMOEDIT() when the user presses Ins which toggles between the insert and overstrike. It also changes in a user function using READINSERT() or RETURNing 22. In insert mode, characters are entered into the text buffer at the current cursor position and the remainder of the text moves to the right. Insert mode is indicated in the scoreboard area. In overstrike mode, characters are entered at the current cursor position overwriting existing characters while the rest of the text buffer remains in its current position.

As the user enters text and the cursor reaches the edge of the MEMOEDIT() window, the current line wraps to the next line in the text buffer and a soft carriage return (CHR(141)) is inserted into the text. If the *<nLineLength>* argument is specified, text wraps when the cursor position is the same as *<nLineLength>*. If *<nLineLength>* is greater than the width of the MEMOEDIT() window, the window scrolls. To explicitly start a new line or paragraph, the user must press Return.

- ***The edit screen:*** When MEMOEDIT() displays, it overwrites the specified area of the screen and does not save the underlying screen. Additionally, it does not display a border or a title. To provide these facilities, you must create a procedure or user-defined function that performs these actions, and then calls MEMOEDIT(). See the example below.

- ***The user function:*** *<cUserFunction>*, a user-defined function specified as an argument, handles key exceptions and reconfigures special keys. The user function is called at various times by MEMOEDIT(), most often in response to keys it does not recognize. Keys that instigate a key exception are all available control keys, function keys, and Alt keys. Since these keys are not processed by MEMOEDIT(), they can be reconfigured. Some of these keys have a default action assigned to them. In the user function, you perform various actions depending on the current MEMOEDIT() mode, and then RETURN a value telling MEMOEDIT() what to do next.

  When the user function argument is specified, MEMOEDIT() defines two classes of keys: nonconfigurable and key exceptions. When a nonconfigurable key is pressed, MEMOEDIT() executes it; otherwise, a key exception is generated and the user function is called. When there are no keys left in the keyboard buffer for MEMOEDIT() to process, the user function is called once again.

When MEMOEDIT() calls the user function, it automatically passes three parameters indicating the MEMOEDIT() mode, the current text buffer line, and the current text buffer column. The mode indicates the current state of MEMOEDIT() depending on the last key pressed or the last action taken prior to executing the user function. The following modes are possible:

MEMOEDIT() Modes

Mode	Memoedit.ch	Description
0	ME_IDLE	Idle, all keys processed
1	ME_UNKEY	Unknown key, memo unaltered
2	ME_UNKEYX	Unknown key, memo altered
3	ME_INIT	Initialization mode

A mode value of 3 indicates that MEMOEDIT() is in initialization mode. When you specify *<cUserFunction>*, MEMOEDIT() makes a call to the user function immediately after being invoked. At this point, you RETURN a request to set MEMOEDIT()'s various text formatting modes: word wrap, scroll, or insert. MEMOEDIT() calls the user function repeatedly, remaining in the initialization mode until you RETURN 0. The text buffer is then displayed, and the user enters the edit mode set by *<lEditMode>*. Note that if word wrap is on when MEMOEDIT() changes from initialization to edit mode, the entire text buffer is formatted with *<nLineLength>*. To prevent this initial formatting, toggle word wrap off during initialization. Note also that the toggles for scroll and word wrap are not assigned to any key, but can be assigned to a key from the user function.

Modes 1 and 2 indicate that MEMOEDIT() has fetched an unrecognizable or configurable key from the keyboard buffer. Configurable keys are processed by RETURNing 0 to execute the MEMOEDIT() default action. RETURNing a different value executes another key action, thereby redefining the key. If the key is an unrecognizable key, you can define an action for it by RETURNing a value requesting a key action or perform an action of your own definition.

Mode 0 indicates that MEMOEDIT() is now idle with no more keys to process. Whenever MEMOEDIT() becomes idle, it always make a call to the user function. At this point, you generally update line and column number displays.

The other two parameters, current line and column, indicate the current cursor position in the text buffer when the user function is called. The line parameter begins with position one (1), and column begins with position zero (0).

When the mode is either 1, 2, or 3, you can return a value instructing MEMOEDIT() what action to perform next. The following table summarizes the possible return values and their consequences:

MEMOEDIT() User Function Return Values

Value	Memoedit.ch	Action
0	ME_DEFAULT	Perform default action
1-31	ME_UNKEY	Process requested action corresponding to key value
32	ME_IGNORE	Ignore unknown key
33	ME_DATA	Treat unknown key as data
34	ME_TOGGLEWRAP	Toggle word wrap mode
35	ME_TOGGLESCROLL	Toggle scroll mode
100	ME_WORDRIGHT	Perform word-right operation
101	ME_BOTTOMRIGHT	Perform bottom-right operation

- ***Header files:*** To make the mode and request values easier to remember and use, the header file Memoedit.ch is supplied in \CLIP53\INCLUDE. Additionally, Inkey.ch, which contains manifest constants for all the INKEY() values, is also located in the same directory.

Notes

- ***Configuring keys:*** If the *<cUserFunction>* is specified, the keys in the table below are configurable.

MEMOEDIT() Configurable Keys

Key	Default Action
Ctrl+Y	Delete the current line
Ctrl+T	Delete word right
Ctrl+B	Reform Paragraph
Ctrl+V/Ins	Toggle insert mode
Ctrl+W	Finish editing with save
Esc	Abort edit and return original

If the key is configurable, RETURNing 0 executes the MEMOEDIT() default action. RETURNing a different value, however, executes another key action thereby redefining the key. If the key is not a configurable key recognized by MEMOEDIT(), you can define an action for it also by RETURNing a value requesting a key action from the table above.

- ***Word wrap:*** Word wrap is a formatting mode you can toggle by RETURNing 34 from the user function. When word wrap is on (the default setting), MEMOEDIT() inserts a soft carriage return/line feed at the closest word break to the window border or line length, whichever occurs first. When word wrap is off, MEMOEDIT() scrolls text buffer beyond the edge of the window until the cursor reaches the end of line. At this point, the user must press Return (inserting a hard carriage return/line feed) to advance to the next line.

- ***Reforming paragraphs:*** Pressing Ctrl+B or RETURNing a 2 from a user function reformats the text buffer until a hard carriage return (end of paragraph) or the end of the text buffer is reached. This happens regardless of whether word wrap is on or off.

- ***Soft carriage returns:*** In CA-Clipper, the insertion of soft carriage return/linefeed characters is never allowed to change the significant content of the text. That is, when a soft carriage return/linefeed is inserted between two words, the space characters between the two words are preserved. When text is reformatted, any soft carriage return/linefeed characters are removed. This leaves the text in its original form and properly handles the case where a soft carriage return/linefeed has been inserted in the middle of a word.

  In the Summer '87 version of MEMOEDIT(), when a soft carriage return/line feed is inserted, a single space character is removed from the text at that point. If the text is later reformatted using a different line width, each soft carriage return/linefeed is replaced by a single space. However, if the text string is reformatted using any of the CA-Clipper text handling functions, words that were separated by a soft carriage return/linefeed will be run together because the soft carriage return/linefeed is not replaced with a space.

  To prevent this, text that was formatted using Summer '87 MEMOEDIT() should be processed to change any soft carriage return/linefeed pairs into space characters. This can be accomplished using the STRTRAN() function as follows:

```
STRTRAN( <text>, CHR(141)+CHR(10), " " )
```

  To convert memo values in an existing database, the following two line program can be used:

```
USE <xcDatabase>
REPLACE ALL <idMemo> WITH ;
    STRTRAN( <idMemo>, CHR(141)+CHR(10), " " )
```

  Because of the .dbt file format, replacing all occurrences of a memo field can cause the .dbt file to grow significantly. The .dbt file can be reduced by copying the .dbf to a new file.

For very large .dbt files, it may not be feasible to perform the above procedure. The supplied utility program, DBT50.EXE located in \CLIP53\BIN, may be useful in these cases. DBT50 scans an entire .dbt file, replacing any soft carriage return/line feed pairs with two spaces. Although this has the undesirable effect of causing certain words to be separated by two spaces instead of one, it allows the file to be processed in place without using additional disk space. DBT50 modifies only soft carriage return/linefeed pairs in the target file. Other text is unaffected.

- ***Editing text files:*** MEMOEDIT() edits text files if the text file can be read into a CA-Clipper character variable. This can be done with the MEMOREAD() function. After editing the contents of the text file held in the character variable, write it back to the file using MEMOWRIT().

Examples

- This example lets you browse a memo field but prevents any changes to the text buffer:

```
USE Customer NEW
SET CURSOR OFF
MEMOEDIT(CustNotes, 5, 10, 20, 69, .F.)
SET CURSOR ON
```

- This example allows editing of a memo field, assigning the changes back to the memo field:

```
USE Customer NEW
REPLACE CustNotes WITH ;
      MEMOEDIT(CustNotes, 5, 10, 20, 69)
```

- This example creates a character string using MEMOEDIT():

```
LOCAL cNotes
cNotes = MEMOEDIT()
```

- This example is a user-defined function that edits a character string in a boxed window displayed with a title:

```
FUNCTION EditMemo( cString, cTitle,;
         nTop, nLeft, nBottom, nRight )
   LOCAL cScreen := SAVESCREEN(nTop, nLeft,;
         nBottom, nRight)
   @ nTop - 1, nLeft - 2 CLEAR TO nBottom + 1,;
         nRight + 2
   @ nTop - 1, nLeft - 2 TO nBottom + 1, nRight + 2
   @ nTop - 1, nLeft SAY "[" + cTitle + "]"
   cString = MEMOEDIT(cString, nTop, nLeft,;
         nBottom, nRight)
   RESTSCREEN(nTop, nLeft, nBottom, nRight, cScreen)
   RETURN (cString)
```

- This example reads the contents of a text file into a character variable, edits it, and then writes it back to disk:

```
LOCAL cString := MEMOREAD("Text.txt")
cString := MEMOEDIT(cString)
IF !MEMOWRIT("Text.txt", cString)
    ? "Write error"
    BREAK
ENDIF
RETURN
```

- This example contains a user-defined function that displays a message describing the current MEMOEDIT() mode. Additionally, while in ME_UNKEY mode, the function will perform either a ME_WORDRIGHT or ME_BOTTOMRIGHT action depending on which associated function key is pressed:

```
#include "Memoedit.ch"
#include "Inkey.ch"

PROCEDURE Main()
    USE Customer NEW
    REPLACE CustNotes WITH;
    MEMOEDIT( CustNotes, 5, 5, 15, 75, .T., "MemoUDF" )
RETURN

FUNCTION MemoUDF( nMode, nLine, nCol )
    LOCAL nKey := LASTKEY()
    LOCAL nRetVal := ME_DEFAULT        // Default return action

    DO CASE
    CASE nMode == ME_IDLE
       @ 20, 5 SAY "MemoMode is ME_IDLE   "
    CASE nMode == ME_UNKEY
       @ 20, 5 SAY "MemoMode is ME_UNKEY "
       DO CASE
       CASE nKey == K_F2
          nRetVal := ME_WORDRIGHT
       CASE nKey == K_F3
          nRetVal := ME_BOTTOMRIGHT
       ENDCASE
       CASE nMode == ME_UNKEYX
       @ 20, 5 SAY "MemoMode is ME_UNKEYX"
    OTHERWISE
       @ 20, 5 SAY "MemoMode is ME_INIT   "
    ENDCASE

RETURN nRetVal
```

Files Library is EXTEND.LIB, header files are Memoedit.ch and Inkey.ch.

See Also HARDCR(), LASTKEY(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT(), SET SCOREBOARD

MEMOLINE() function

Extract a line of text from a character string or memo field

Syntax

```
MEMOLINE(<cString>,
    [<nLineLength>],
    [<nLineNumber>],
    [<nTabSize>],
    [<lWrap>]) → cLine
```

Arguments

<cString> is the memo field or character string from which a line of text is to be extracted.

<nLineLength> specifies the number of characters per line and can be between four and 254. If not specified, the default line length is 79.

<nLineNumber> is the line number to be extracted. If not specified, the default value is one.

<nTabSize> defines the tab size. If not specified, the default value is four. If *<nTabSize>* is greater than or equal to *<nLineLength>*, then the tab size is automatically converted to *<nLineLength>* - 1.

<lWrap> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on; false (.F.) toggles it off. If not specified, the default value is true (.T.).

Returns

MEMOLINE() returns the line of text specified by *<nLineNumber>* in *<cString>* as a character string. If the line has fewer characters than the indicated length, the return value is padded with blanks. If the line number is greater than the total number of lines in *<cString>*, MEMOLINE() returns a null string ("").

If *<lWrap>* is true (.T.) and the indicated line length breaks the line in the middle of a word, that word is not included as part of the return value but shows up at the beginning of the next line extracted with MEMOLINE().

If *<lWrap>* is false (.F.), MEMOLINE() returns only the number of characters specified by the line length. The next line extracted by MEMOLINE() begins with the character following the next hard carriage return, and all intervening characters are not processed.

Description

MEMOLINE() is a memo function used with MLCOUNT() to extract lines of text from character strings and memo fields based on the number of characters per line. It is the most basic facility provided by CA-Clipper to display memo fields and long strings.

The basic method of operation is to determine the number of lines in the memo field or character string using MLCOUNT() with the same number of characters per line, tab size, and wrapping behavior as you intend to use with MEMOLINE(). Using this value as the upper boundary of a FOR...NEXT, each line of the memo field or character string can be extracted with MEMOLINE() and processed with any combination of output commands and functions required.

Examples

- This example demonstrates the general method for displaying memo fields and long character strings using the combination of MLCOUNT() and MEMOLINE():

```
LOCAL nLineLength := 40, nTabSize := 3, lWrap := .T.
LOCAL nLines, nCurrentLine
USE Customer INDEX CustName NEW
//
nLines := MLCOUNT(CustNotes, nLineLength,;
       nTabSize, lWrap)
//
SET PRINTER ON
FOR nCurrentLine := 1 TO nLines
    ? MEMOLINE(CustNotes, nLineLength, nCurrentLine,;
          nTabSize, lWrap)
NEXT
SET PRINTER OFF
```

Files

Library is EXTEND.LIB.

See Also

MEMOEDIT(), MLCOUNT(), MLPOS()

MEMOREAD() function

Return the contents of a disk file as a character string

Syntax

```
MEMOREAD(<cFile>) → cString
```

Arguments

<cFile> is the name of the file to read from disk. It must include an extension, if there is one, and can optionally include a path.

Returns

MEMOREAD() returns the contents of a text file as a character string. The maximum file size that can be read is 65,535 characters (64K)—the maximum size of a character string. If <*cFile*> cannot be found, MEMOREAD() returns a null string ("").

Description

MEMOREAD() is a memo function that reads a disk file into memory where it can be manipulated as a character string or assigned to a memo field. MEMOREAD() is used with MEMOEDIT() and MEMOWRIT() to edit an imported disk file, and then write it back to disk. MEMOREAD() searches for <*cFile*> beginning with the current DOS directory. If the file is not found, MEMOREAD() searches the DOS path. MEMOREAD() does not use the CA-Clipper DEFAULT or PATH to search for <*cFile*>.

In a network environment, MEMOREAD() attempts to open the specified file shared and read-only. If the file is opened exclusive by another process, MEMOREAD() returns a null string ("").

Examples

- This example uses MEMOREAD() to assign the contents of a text file to the Notes memo field and to a character variable:

```
REPLACE Notes WITH MEMOREAD("Temp.txt")
cString = MEMOREAD("Temp.txt")
```

- This example defines a function that edits a disk file:

```
FUNCTION Editor( cFile )
   LOCAL cString
   IF (cString := MEMOREAD(cFile)) == ""
      ? "Error reading " + cFile
      RETURN .F.
   ELSE
      MEMOWRIT(cFile, MEMOEDIT(cString))
      RETURN .T.
   ENDIF
```

Files Library is EXTEND.LIB.

See Also MEMOEDIT(), MEMOWRIT(), REPLACE

MEMORY() function

Determine the amount of available free pool memory

Syntax

MEMORY(<*nExp*>) → *nKbytes*

Arguments

<*nExp*> is a numeric value that determines the type of value MEMORY() returns as follows:

MEMORY() Argument Values

Value	Meaning
0	Estimated total space available for character values
1	Largest contiguous block available for character values
2	Area available for RUN commands

Returns

MEMORY() returns an integer numeric value representing the amount of memory available, in one kilobyte increments.

Description

MEMORY() is an environment function that reports various states of free pool memory. (Free pool is the dynamic region of memory that stores character strings and executes RUN commands.)

Examples

- This example uses MEMORY() before a RUN command to determine if there is enough memory available to execute the external program:

```
#define MEM_CHAR  0
#define MEM_BLOCK 1
#define MEM_RUN   2
//
IF MEMORY(MEM_RUN) >= 128
   RUN MYPROG
ELSE
   ? "Not enough memory to RUN"
   BREAK
ENDIF
```

Files

Library is CLIPPER.LIB.

MEMOSETSUPER() function

Set a RDD inheritance chain for the DBFMEMO database driver

Syntax

```
MEMOSETSUPER([<cSuperRDD>]) → cOldSuperName
```

Arguments

<cSuperRDD> is a character string representing the name of the RDD from which DBFMEMO will inherit.

Returns

MEMOSETSUPER() always returns NIL.

Description

The DBFMEMO driver is only capable of handling commands and functions that relate to memo fields. Therefore, it *must* inherit the non-memo database characteristics (such as SKIP, REPLACE, SEEK, etc.) from a "super" driver (such as DBFNTX).

For example, the DBFCDX RDD is nothing more than the DBFMEMO RDD hardwired to inherit database characteristics from the _DBFCDX RDD. That is, when you use the DBFCDX RDD, you are using the DBFMEMO driver to handle the memo field operations and you are using the _DBFCDX driver to handle the other database and index operations as the "super" driver.

At times, you may want to use a database driver such as DBFNTX instead of DBFCDX but, at the same time, have the efficient and enhanced memo field capabilities of the DBFMEMO RDD instead of the DBT memo file format. In order to do this, you would use MEMOSETSUPER().

Warning! *The MEMOSETSUPER() function can only set one inheritance chain per application instance. That is, once you have set MEMOSETSUPER() in an application, MEMOSETSUPER() cannot be changed. Doing so will have unpredictable results.*

Examples

- This example uses MEMOSETSUPER() to create a database (.dbf) file with an .fpt/.dbv memo field:

```
REQUEST DBFMEMO
// Set the default to DBFMEMO
RDDSETDEFAULT( "DBFMEMO" )
// Make DBFMEMO inherit from DBFNTX
MEMOSETSUPER( "DBFNTX" )

// Create a DBF with an FPT/DBV not DBT
DBCREATE( "test.dbf", {{"notes", "M", 10, 0}} )

// Open DBF with FPT/DBV
// Indexes created would be NTX if created
USE test NEW
```

MEMOTRAN() function

Replace carriage return/linefeeds in character strings

Syntax

```
MEMOTRAN(<cString>,
   [<cReplaceHardCR>],
   [<cReplaceSoftCR>]) → cNewString
```

Arguments

<cString> is the character string or memo field to be searched.

<cReplaceHardCR> is the character with which to replace a hard carriage return/line feed pair. If not specified, the default value is a semicolon (;).

<cReplaceSoftCR> is the character with which to replace a soft carriage return/line feed pair. If not specified, the default value is a space.

Returns

MEMOTRAN() returns a copy of *<cString>* with the specified carriage return/line feed pairs replaced.

Description

MEMOTRAN() is a memo function that converts a memo field or long character string containing hard and soft carriage return/line feed characters into a form that can be displayed. These two character combinations are end of line formatting indicators placed in the string by MEMOEDIT(). Soft carriage returns (CHR(141)) are inserted when a line longer than the width of the MEMOEDIT() window wraps. Hard carriage returns (CHR(13)) are inserted when the user explicitly presses Return.

MEMOTRAN() is particularly useful when displaying a memo field in a REPORT FORM which does not wrap when a soft carriage return is encountered. MEMOTRAN() resolves this by converting soft carriage returns to spaces. Note, however, that you must declare MEMOTRAN() as external using the REQUEST statement if it is used in a REPORT FORM and not specified anywhere else in the current program.

Examples

- This example strips all end of line characters from a memo field:

```
REPLACE Notes WITH MEMOTRAN(Notes)
```

Files Library is EXTEND.LIB.

See Also EXTERNAL, HARDCR(), REPORT FORM, REQUEST, STRTRAN()

MEMOWRIT() function

Write a character string or memo field to a disk file

Syntax

```
MEMOWRIT(<cFile>, <cString>) → lSuccess
```

Arguments

<cFile> is the name of the target disk file including the file extension and optional path and drive designator.

<cString> is the character string or memo field to write to *<cFile>*.

Returns

MEMOWRIT() returns true (.T.) if the writing operation is successful; otherwise, it returns false (.F.).

Description

MEMOWRIT() is a memo function that writes a character string or memo field to a disk file. If a path is not specified, MEMOWRIT() writes *<cFile>* to the current DOS directory and not the current DEFAULT directory. If *<cFile>* already exists, it is overwritten.

MEMOWRIT() is generally used with MEMOREAD() to load text files into memory where they can be edited, displayed, and written back to disk. You can also use MEMOWRIT() as a quick way of exporting a memo field to a text file.

Examples

- This example uses MEMOWRIT() with MEMOREAD() to allow editing of memo fields with an external editor:

```
LOCAL cEditor := "MYEDIT.EXE"
USE Sales NEW
IF MEMOWRIT("Cliptmp.txt", Notes)
    RUN (cEditor + " Cliptmp.txt")
    REPLACE Notes WITH MEMOREAD("Cliptmp.txt")
ELSE
    ? "Error while writing Cliptmp.txt"
    BREAK
ENDIF
```

Files

Library is EXTEND.LIB.

See Also

MEMOEDIT(), MEMOREAD()

MEMVAR statement

Declare private and public variable names

Syntax

```
MEMVAR <idMemvar list>
```

Arguments

<idMemvar list> is a list of public and private variable names to declare to the compiler.

Description

MEMVAR is a declaration statement that causes the compiler to resolve references to variables specified without an explicit alias by implicitly assuming the memory variable alias (MEMVAR>). Only explicit, unaliased references to the specified variables are affected. MEMVAR, like all declaration statements, has no effect on references made within macro expressions or variables.

The MEMVAR statement neither creates the variables nor verifies their existence. Its primary effect is to ensure correct references to variables whose existence is known to be guaranteed at runtime. At runtime, the specified variables must be created using the PRIVATE, PARAMETERS or PUBLIC statements. This can occur in the procedure containing the MEMVAR declaration or in a higher level procedure. Attempting to access the variables before they are created will cause an error.

The scope of the MEMVAR declaration is the procedure or function in which it occurs, or the entire source file if it precedes any PROCEDURE or FUNCTION statements and the /N compiler option is used. The /N option suppresses automatic definition of a procedure with the same name as the program (.prg) file.

Like other declaration statements, MEMVAR must precede any executable statements, including PARAMETERS, PUBLIC, and PRIVATE statements in a procedure or function definition, or the program (.prg) file if the declaration has filewide scope.

MEMVAR can be used in conjunction with the /W compiler option—which generates warning messages for ambiguous variable references—to perform compile-time checking for undeclared variables.

For more information on variable declarations and scoping, refer to the Variables section in the "Basic Concepts" chapter of the *Programming and Utilities Guide*.

Examples

- This example demonstrates the relationship between a private and field variable with the same name. The private variable is declared with the MEMVAR statement:

```
FUNCTION Example
   MEMVAR amount, address
   PRIVATE amount := 100
   USE Customer NEW
   //
   ? amount          // Refers to amount private variable
   ? Customer>Amount
                     // Refers to Amount field variable
   //
   RETURN NIL
```

See Also

FIELD, LOCAL, PRIVATE, PUBLIC, STATIC

MEMVARBLOCK() function

Return a SET-GET code block for a given memory variable

Syntax

```
MEMVARBLOCK(<cMemvarName>) → bMemvarBlock
```

Arguments

<cMemvarName> is the name of the variable referred to by the SET-GET block, specified as a character string.

Returns

MEMVARBLOCK() returns a code block that when evaluated sets (assigns) or gets (retrieves) the value of the given memory variable. If *<cMemvarName>* does not exist, MEMVARBLOCK() returns NIL.

Description

The code block created by MEMVARBLOCK() has two operations depending on whether an argument is passed to the code block when it is evaluated. If evaluated with an argument, it assigns the value of the argument to *<cMemvarName>*. If evaluated without an argument, the code block retrieves the value of *<cMemvarName>*.

Notes

- MEMVARBLOCK() creates SET-GET blocks only for variables whose names are known at runtime. MEMVARBLOCK(), therefore, cannot be used to create SET-GET blocks for local or static variables. The same restriction applies to creating blocks using the macro operator (&).

Examples

- This example compares MEMVARBLOCK() to a code block created using the macro operator (&). Note that using MEMVARBLOCK() allows you to avoid the speed and size overhead of the macro operator:

```
PRIVATE var := "This is a string"
//
// Set-Get block defined using macro operator
bSetGet := &( "{ |setVal|;
       IF( setVal == NIL, var, var := setVal ) }" )
// Set-Get block defined using MEMVARBLOCK()

// bSetGet created here is the functional
// equivalent of bSetGet above
bSetGet := MEMVARBLOCK("var")
```

Files Library is CLIPPER.LIB.

See Also FIELDBLOCK(), FIELDWBLOCK()

MenuItem class

Create a menu item

Description

MenuItem objects are the basis for which both top bar and pop-up menus are built upon.

Class Function

```
MenuItem(<cCaption>, <expData>, [<nShortcut>],
   [<cMessage>], [<nID>]) → oMenuItem
```

Arguments

<cCaption> is a character string that contains either a text string that concisely describes the menu option or a menu separator specifier. This value is assigned to the MenuItem:caption instance variable.

<expData> is a value that contains either a code block or a PopUpMenu object. This argument is ignored when *<cCaption>* contains a menu separator specifier. This value is assigned to the MenuItem:data instance variable.

<nShortcut> is an optional numeric inkey value that indicates the shortcut key combination that selects and launches the menu selection. The default is 0. This value is assigned to the MenuItem:shortcut instance variable. Constant values for various key combinations are defined in Inkey.ch.

<cMessage> is an optional character string that indicates the text to display on the status bar when the menu item is selected. The default is an empty string. This value is assigned to the MenuItem:message instance variable.

<nID> is an optional numeric value that uniquely identifies the menu item. The default is 0. This value is assigned to the MenuItem:id instance variable.

Returns

Returns a MenuItem object when all of the required arguments are present; otherwise, MenuItem() returns NIL.

Exported Instance Variables

`caption` (Assignable)

Contains either a text string that concisely describes the menu option or a menu separator specifier. MenuItem:caption is the text that appears in the actual menu.

A menu separator is a horizontal line in a pop-up menu that separates menu items into logical groups. Use the constant MENU_SEPARATOR in Button.ch to assign the menu separator specifier to MenuItem:caption.

When present, the & character specifies that the character immediately following it in the caption is the menu item's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to select a menu item when the menu that it is contained within has input focus. When the menu is a member of a TopBarMenu object, the user selects the menu item by pressing the Alt key in combination with the accelerator key. When the menu is a member of a PopUpMenu object, the user selects the menu item by simply pressing the accelerator key. The accelerator key is not case sensitive.

`cargo` (Assignable)

Contains a value of any type that is ignored by the MenuItem object. MenuItem:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a MenuItem object and retrieved later.

`checked` (Assignable)

Contains a logical value that indicates whether a check mark appears to the left of the menu item's caption. A value of true (.T.) indicates that a check mark should show; otherwise, a value of false (.F.) indicates that it should not.

`data` (Assignable)

Contains either a code block or a PopUpMenu object or, when the menu item's Caption property contains a menu separator specifier, MenuItem:data contains NIL. When the menu item is selected, its code block, if present, is evaluated; otherwise, its PopUpMenu object is opened.

`enabled` (Assignable)

Contains a logical value that indicates whether the menu item can be selected or not. MenuItem:enabled contains true (.T.) to permit user access; otherwise, it contains false (.F.) to deny user access. When disabled, the item will be shown in its disabled color.

`id` (Assignable)

Contains an optional numeric value that uniquely identifies the menu item. The default is 0. This value is returned by MENUMODAL() to indicate the selected menu item.

`message` (Assignable)

Contains an optional string that describes the menu item. This is the text that appears on the screen's status bar line. The default is an empty string. The MENUMODAL() function defines the row and width for the area on the screen where messages are displayed.

`shortcut` (Assignable)

Contains an optional numeric inkey value indicating the key that activates the menu selection. The default is 0. Shortcut keys are available only for menu items on a PopUpMenu. TopBarMenu items cannot have shortcut keys associated with them.

The shortcut key name automatically appears to the right of the MenuItem:caption. Unlike with an accelerator key, the menu need not be open for the shortcut key to be active.

`style` (Assignable)

Contains a character string that indicates the delimiter characters that are used by the PopUpMenu:display() method. The string must contain two characters. The first is the character associated with the MenuItem:checked property. Its default value is the square root (√) character. The second is the submenu indicator. Its default is the right arrow (▶) character.

Exported Methods

`<oMenuItem>:isPopUp() → lPopUpStatus`

Returns a logical value that indicates whether the menu item's data property contains a pop-up menu. A value of true (.T.) indicates that MenuItem:data contains a PopUpMenu object; otherwise, a value of false (.F.) indicates that it does not.

isPopUp() is a method of the MenuItem class that is used for determining whether a menu item is a branch in a menu tree. When a menu item is selected, typically one of two results will occur. If it is a menu tree branch, its pop-up menu is opened; otherwise, its code block will be evaluated.

Examples

See the Menu.prg sample file in the \CLIP53\SOURCE\SAMPLE directory. This example demonstrates combining TopBarMenu, PopUpMenu, and MenuItem objects to create a menu with a number of available choices. See "Introduction to the Menu System" in the *Programming and Utilities Guide* for more information about using this class.

See Also

MENUMODAL(), PopUpMenu class, TopBarMenu class

MENUMODAL() function

Activate a top bar menu

Syntax

```
MENUMODAL(<oTopBar>, <nSelection>, <nMsgRow>,
   <nMsgLeft>, <nMsgRight>, <cMsgColor>) → MenuID
```

Arguments

<oTopBar> is a TopBarMenu object created from the TopBarMenu class.

<nSelection> is the TopBarMenu item selected by default.

<nMsgRow> is the row number where menu item messages will appear.

<nMsgLeft> specifies the left border for menu item messages.

<nMsgRight> specifies the right border for menu item messages.

<cMsgColor> defines the color string for the menu item messages. It consists of a single foreground/background pair.

Returns

MENUMODAL() returns the menu ID of the chosen menu item. Menu IDs are assigned using the MenuItem class.

Description

MENUMODAL() is a user-interface function that implements the pull-down menu system in CA-Clipper. It is part of the open architecture Get system of CA-Clipper. MENUMODAL() is similar to the READ command in that it waits for the user to perform an action. However, the MENUMODAL() function will only respond to menu actions.

To implement a menu object at the same time as other objects, use the READMODAL() function which has one of its arguments as TopBarMenu object.

When the user chooses a menu item, control is passed to the code block associated with that particular menu item. Code blocks are defined using the MenuItem class.

The menu items can be selected by using either the keyboard or the mouse. To select a menu item with the mouse, simply select its TopBarMenu item with the mouse and then choose the appropriate PopUp menu item.

Note: The MENUMODAL() function will take one menu event from the user and then terminate. To avoid this, the following can be used, and the same will allow the program to continuously accept menu events:

```
DO WHILE (MENUMODAL(themenu,...) <> ExitMenu)
ENDDO
```

The following table lists the active keys that can be used during MENUMODAL():

MENUMODAL() Navigation Keys

Key	Action
Left arrow, Ctrl+S	Move to the next TopBarMenu item to the left. If there are no more items to the left, the rightmost TopBarMenu item will be selected.
Right arrow, Ctrl+D	Move to the next TopBarMenu item to the right. If there are no more items to the right, the leftmost TopBarMenu will be selected.
Up arrow, Ctrl+E	Move to the previous PopUp menu item. If there are no more items above the current item, the menu item on the bottom will be selected.
Down arrow, Ctrl+X	Move to the next PopUp menu item. If there are no more items below the current item, the menu item on the top will be selected.

Examples

See the Menu.prg sample file in the \CLIP53\SOURCE\SAMPLE directory. This example demonstrates combining TopBarMenu, PopUpMenu, and MenuItem objects to create a menu with a number of available choices. See "Introduction to the Menu System" in the *Programming and Utilities Guide* for more information about using this function.

Files

Library is CLIPPER.LIB, source file is SOURCE\SYS\MENUSYS.PRG

See Also

MenuItem class, PopUpMenu class, READMODAL(), READ(), TopBarMenu class

MENU TO command

Execute a lightbar menu for defined PROMPTs

Syntax

```
MENU TO <idVar>
```

Arguments

<idVar> is the name of the variable to be assigned the result of the menu selection. If the specified variable is not visible or does not exist, a private variable is created and assigned the result.

Description

MENU TO is the selection mechanism for the CA-Clipper lightbar menu system. Before invoking MENU TO, first define the menu items and associated MESSAGEs with a series of @...PROMPT commands. Then, activate the menu with MENU TO *<idVar>*. If *<idVar>* does not exist or is not visible, MENU TO creates it as a private variable and places the highlight on the first menu item. If *<idVar>* does exist, its initial value determines the first menu item highlighted.

Notes

- ***Color:*** Menu items are painted in the current standard color with the highlighted menu item appearing in the current enhanced color.
- ***Navigation and selection:*** Pressing the arrow keys moves the highlight to the next or previous menu item. As each menu item is highlighted the associated MESSAGE displays on the row specified with SET MESSAGE. If WRAP is ON, an Up arrow from the first menu item moves the highlight to the last menu item. Also, a Down arrow from the last menu item moves the highlight to the first.

  To make a selection, press Return or the first character of a menu item. MENU TO then returns the position of the selected menu item as a numeric value into the specified memory variable. Pressing Esc aborts the menu selection and returns zero. The table below summarizes the active keys within MENU TO.

- ***SET KEY procedures:*** A MENU TO command can be nested within a SET KEY procedure invoked within a menu without clearing the pending PROMPTs.

MENU TO Active Keys

Key	Action
Up arrow	Move to previous menu item
Down arrow	Move to next menu item
Home	Move to first menu item
End	Move to last item menu item
Left arrow	Move to previous menu item
Right arrow	Move to next menu item
PgUp	Select menu item, returning position
PgDn	Select menu item, returning position
Return	Select menu item, returning position
Esc	Abort selection, returning zero
First letter	Select first menu item with same first letter, returning position

Examples

- This example creates a simple vertical lightbar menu with messages appearing centered on line 23. When invoked, the highlight defaults to the second menu item based on the initial value of *nChoice*:

```
LOCAL nChoice := 2
SET WRAP ON
SET MESSAGE TO 23 CENTER
@ 6, 10 PROMPT "Add"  MESSAGE "New Acct"
@ 7, 10 PROMPT "Edit" MESSAGE "Change Acct"
@ 9, 10 PROMPT "Quit" MESSAGE "Return to DOS"
MENU TO nChoice
//
DO CASE
CASE nChoice = 0
   QUIT
CASE nChoice = 1
   NewAccount()
CASE nChoice = 2
   ChangeAccount()
CASE nChoice = 3
   QUIT
ENDCASE
RETURN
```

Files

Library is CLIPPER.LIB.

See Also

@...PROMPT, ACHOICE(), SET MESSAGE, SET WRAP

MHIDE() function

Hide the mouse pointer

Syntax

```
MHIDE() → NIL
```

Returns

MHIDE() always returns NIL.

Description

MHIDE() hides the mouse pointer. This function should be used together with MSHOW() when updating the screen. It is important to hide the mouse pointer before changing the screen display and then show it again after the change.

Note: The MSETCURSOR() function should be used in place of MSHOW() and MHIDE(). It is kept here for compatibility.

Examples

- This example uses the mouse pointer:

```
MHIDE()
@ 10, 20 say "Hi there, folks!!!"
MSHOW()
```

- You can automate calls to MHIDE()/MSHOW() by modifying parts of your header files (*.ch). For example:

```
#command @ <row>, <col> SAY <xpr>;[PICTURE <pic>];
    [COLOR <color>];
    => DEVPOS(<row>, <col>);
    DEVOUTPict(<xpr>, <pic> [, <color>])
    // Can be changed to
#command @ <row>, <col> SAY <xpr>;
    [PICTURE <pic>];
    [COLOR <color>];

    => MHIDE();
    DEVPOS(<row>, <col>);
    DEVOUTPict(<xpr>, <pic> [, <color>]);
    MSHOW()
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

MSHOW(), MSETCLIP(), MSETCURSOR()

MIN() function

Return the smaller of two numeric or date values

Syntax

```
MIN(<nExp1>, <nExp2>) → nSmaller
MIN(<dExp1>, <dExp2>) → dSmaller
```

Arguments

<nExp1> and ***<nExp2>*** are the numeric values to be compared.

<dExp1> and ***<dExp2>*** are the date values to be compared.

Returns

MIN() returns the smaller of the two arguments. The value returned is the same data type as the arguments.

Description

MIN() is a numeric and date function that ensures the value of an expression is smaller than a specified minimum. The inverse of MIN() is MAX() which returns the greater of two numeric or date values.

Examples

- In these examples MIN() returns the smaller of two numeric values:

```
? MIN(99, 100)                  // Result: 99
? MIN(100, 99)                  // Result: 99
```

- In these examples MIN() compares date values:

```
? DATE()                        // Result: 09/01/90
? MIN(DATE(), DATE() + 30)      // Result: 09/01/90
```

Files

Library is CLIPPER.LIB.

See Also

MAX()

MLCOUNT() function

Count the number of lines in a character string or memo field

Syntax

```
MLCOUNT(<cString>, [<nLineLength>],
   [<nTabSize>], [<lWrap>]) → nLines
```

Arguments

<cString> is the character string or memo field to be counted.

<nLineLength> specifies the number of characters per line and can range from four to 254. If not specified, the default line length is 79.

<nTabSize> defines the tab size. If not specified, the default value is four. If *<nTabSize>* is greater than or equal to *<nLineLength>*, then the tab size is automatically converted to *<nLineLength>* - 1.

<lWrap> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on; false (.F.) toggles it off. If not specified, the default value is true (.T.).

Returns

MLCOUNT() returns the number of lines in *<cString>* depending on the *<nLineLength>*, the *<nTabSize>*, and whether word wrapping is on or off.

Description

MLCOUNT() is a memo function used with MEMOLINE() to print character strings and memo fields based on the number of characters per line. In the basic operation, use MLCOUNT() to return the number of lines in the character string or memo field. Then, using MEMOLINE() to extract each line, loop through the memo field until there are no lines left.

If *<lWrap>* is true (.T.) and an end of line position breaks a word, it is word wrapped to the next line and the next line begins with that word. If *<lWrap>* is false (.F.), MLCOUNT() counts the number of characters specified by *<nLineLength>* as the current line. The next line begins with the character following the next hard or soft carriage return. Intervening characters are ignored.

Examples

- This example displays the contents of each Notes memo field in the Sales database file, one line at a time:

```
USE Sales NEW
nLineLength = 65
//
DO WHILE !EOF()
    nLines = MLCOUNT(Sales->Notes, nLineLength)
    FOR nCurrLine = 1 TO nLines
       ? MEMOLINE(Sales->Notes, nLineLength, nCurrLine)
    NEXT
    SKIP
    ?
ENDDO
```

Files Library is EXTEND.LIB.

See Also MEMOLINE(), MEMOTRAN(), MLPOS()

MLCTOPOS() function

Return the byte position of a formatted string based on line and column position

Syntax

```
MLCTOPOS(<cText>, <nWidth>, <nLine>,
   <nCol>, [<nTabSize>], [<lWrap>]) → nPosition
```

Arguments

<cText> is the text string to be scanned.

<nWidth> is the line length formatting width.

<nLine> is the line number counting from 1.

<nCol> is the column number counting from 0.

<nTabSize> is the number of columns between tab stops. If not specified, the default is 4.

<lWrap> is the word wrap flag. If not specified, the default is true (.T.).

Returns

MLCTOPOS() returns the byte position within *<cText>* counting from 1.

Description

MLCTOPOS() is a memo function that determines the byte position that corresponds to a particular line and column within the formatted text. Note that the line number is one-relative and the column number is zero-relative. This is compatible with MEMOEDIT(). The return value is one-relative, making it suitable for use in SUBSTR() or other string functions.

MLCTOPOS() is used with MPOSTOLC() to create search routines or other text processing for MEMOEDIT(). Refer to the source code for the program editor (PE.EXE) found in the \CLIP53\SOURCE\PE directory.

Examples

- This example determines the byte position of line 5, column 3 in the *cText* string:

```
cText := "Note the side on which the bread ;
            is buttered."
//
? MLCTOPOS(cText, 5, 3, 0)       // Result: 10
```

Files Library is CLIPPER.LIB.

See Also MEMOEDIT(), MLPOS(), MPOSTOLC()

MLEFTDOWN() function

Determine the press status of the left mouse button

Syntax

```
MLEFTDOWN() → lIsPressed
```

Returns

MLEFTDOWN() returns true (.T.) if the left mouse button is currently pressed; otherwise, it returns false (.F.).

Description

MLEFTDOWN() determines the button press status of the left mouse button. This is particularly useful for mouse-oriented routines when the status of its left button is critical, such as dragging and dropping or floating menu bars.

See Also

MRIGHTDOWN()

MLPOS() function

Determine the position of a line in a character string or memo field

Syntax

```
MLPOS(<cString>, <nLineLength>,
   <nLine>, [<nTabSize>], [<lWrap>]) → nPosition
```

Arguments

<cString> is a character string or memo field.

<nLineLength> specifies the number of characters per line.

<nLine> specifies the line number.

<nTabSize> defines the tab size. The default is four. If *<nTabSize>* is greater than or equal to *<nLineLength>*, then the tab size is adjusted to *<nLineLength>* - 1.

<lWrap> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on, and false (.F.) toggles it off. The default is true (.T.).

Returns

MLPOS() returns the character position of *<nLine>* in *<cString>* as an integer numeric value. If *<nLine>* is greater than the number of lines in *<cString>*, MLPOS() returns the length of *<cString>*.

Examples

- This example uses MLPOS() to find the position of a specific line, given a line length:

```
cString = MEMOREAD("Temp.txt")
nLineLength = 40
nLine = 5
nPosition = MLPOS(cString, nLineLength, nLine)
? SUBSTR(cString, nPosition, 12)
```

Files

Library is EXTEND.LIB.

See Also

MEMOLINE(), MEMOTRAN(), MLCOUNT()

*MOD() function

Return the dBASE III PLUS modulus of two numbers

Syntax

```
MOD(<nDividend>, <nDivisor>) → nRemainder
```

Arguments

<nDividend> is the dividend of the division operation.

<nDivisor> is the divisor of the division operation.

Returns

MOD() returns a number representing the remainder of *<nDividend>* divided by *<nDivisor>*.

Description

MOD() is a numeric function that emulates the dBASE III PLUS MOD() function. It is implemented using the CA-Clipper modulus operator (%). Note that there are differences between the dBASE III PLUS MOD() function and the CA-Clipper modulus operator which are described in the following table:

Differences Between dBASE III PLUS MOD() Function and the CA-Clipper Modulus Operator

Dividend	Divisor	Modulus Operator	MOD()	dBASE III PLUS MOD() function
3	0	Error	Error	3
3	-2	1	-1	-1
-3	2	-1	1	1
-3	0	Error	Error	-3
-1	3	-1	2	2
-2	3	-2	1	1
2	-3	2	-1	-1
1	-3	1	-2	-2

MOD() is supplied as a compatibility function and therefore not recommended. It is superseded entirely by the modulus operator (%).

Notes

- ***Zero divisor in dBASE III PLUS:*** In dBASE III PLUS, a zero divisor returns the dividend for every value of the dividend. In CA-Clipper, by contrast, the modulus of any dividend using a zero divisor causes a runtime error.

- ***Zero divisor in earlier versions:*** In versions of CA-Clipper prior to Summer '87, a modulus operation with a zero divisor returned zero for all dividends. In Summer '87 and later versions, it returns a runtime error.

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\MOD.PRG.

MONTH() function

Convert a date value to the number of the month

Syntax

```
MONTH(<dDate>) → nMonth
```

Arguments

<dDate> is the date value to be converted.

Returns

MONTH() returns an integer numeric value in the range of zero to 12. Specifying a null date (CTOD("")) returns zero.

Description

MONTH() is a date conversion function that is useful when you require a numeric month value during calculations for such things as periodic reports. MONTH() is a member of a group of functions that return components of a date value as numeric values. The group includes DAY() and YEAR() to return the day and year values as numerics. CMONTH() is a related function that allows you to return the name of the month from a date value.

Examples

- These examples return the month of the system date:

```
? DATE()                          // Result: 09/01/90
? MONTH(DATE())                   // Result: 9
? MONTH(DATE()) + 1               // Result: 10
```

- This example demonstrates MONTH() acting on a null date:

```
#define NULL_DATE    (CTOD(""))
? MONTH(NULL_DATE)                // Result: 0
```

Files Library is CLIPPER.LIB.

See Also CMONTH(), DAY(), DOW(), YEAR()

MPOSTOLC() function

Return line and column position of a formatted string based on a specified byte position

Syntax

```
MPOSTOLC(<cText>, <nWidth>, <nPos>,
   [<nTabSize>], [<lWrap>]) → aLineColumn
```

Arguments

<cText> is a text string.

<nWidth> is the length of the formatted line.

<nPos> is the byte position within text counting from one (1).

<nTabSize> is the number of columns between tab stops. If not specified, the default is four (4).

<lWrap> is the word wrap flag. If not specified, the default is true (.T.).

Returns

MPOSTOLC() returns an array containing the line and the column values for the specified byte position, *<nPos>*.

Description

MPOSTOLC() is a memo function that determines the formatted line and column corresponding to a particular byte position within *<cText>*. Note that the line number returned is one-relative and the column number is zero-relative. This is compatible with MEMOEDIT(). *<nPos>* is one-relative, compatible with AT(), RAT(), and other string functions.

MPOSTOLC(), used with MLCTOPOS(), can create search routines or other text processing for MEMOEDIT(). Refer to the source code for the program editor (PE.EXE) found in \CLIP53\SOURCE\PE directory.

Examples

- This example determines, for the text string shown, the line and column corresponding to the tenth character of the text, assuming a formatting width of five columns. A formatting width of five would cause each of the first three words to be placed on a line by itself. The tenth character of the text is the "s" in "side". The word "side" would be at the leftmost column of the third line of the formatted text, so the return value is {3, 0}.

```
cText := "Note the side on which the bread ;
      is buttered."
//
aLC := MPOSTOLC(cText, 5, 10)       // Result: {3, 0}
```

Files Library is CLIPPER.LIB.

See Also MEMOEDIT(), MLCTOPOS(), MLPOS()

MPRESENT() function

Determine if a mouse is present

Syntax

```
MPRESENT() → lIsPresent
```

Returns

MPRESENT() returns true (.T.) if a mouse is present; otherwise, it returns false (.F.).

Description

MPRESENT() determines whether a mouse is available or not. For a mouse to be available, it must be physically installed in addition to loading the appropriate mouse device driver.

MRESTSTATE() function

Re-establish the previous state of a mouse

Syntax

`MRESTSTATE( <cSaveState> ) → NIL`

Returns

MRESTSTATE() always returns NIL.

Description

MRESTSTATE() is a function that is used for re-establishing a previously saved mouse state. This includes the mouse's screen position, visibility, and boundaries.

See Also

MSAVESTATE()

MRIGHTDOWN() function

Determine the status of the right mouse button

Syntax

```
MRIGHTDOWN() → lIsPressed
```

Returns

MRIGHTDOWN() returns true (.T.) if the mouse's right button is currently pressed; otherwise, it returns false (.F.) .

Description

MRIGHTDOWN() determines the button press status of the right mouse button. This function is provided in the interest of completeness since, on the PC, the right mouse button is typically reserved for single-click oriented activities such as activating a pop-up menu.

See Also

MLEFTDOWN()

MROW() function

Determine a mouse cursor's screen row position

Syntax

```
MROW() → nCurrentMouseRow
```

Returns

MROW() returns the mouse cursor's current screen row position.

Description

MROW() is a function that is used for determining the mouse cursor's screen row position. This is useful when implementing a hit testing routine whose purpose is to determine if the mouse cursor is on pertinent information when the left mouse button is pressed.

See Also

MCOL()

MSAVESTATE() function

Save the current state of a mouse

Syntax

```
MSAVESTATE() → cSaveState
```

Returns

MSAVESTATE() returns a character string that describes the mouse's current state.

Description

MSAVESTATE() is a function that is used for storing the mouse's current state. This includes the mouse's screen position, visibility, and boundaries.

See Also

MRESTSTATE()

MSETBOUNDS() function

Set screen boundaries for the mouse cursor

Syntax

```
MSETBOUNDS( [<nTop>], [<nLeft>], [<nBottom>],
   [<nRight>] ) → NIL
```

Arguments

<nTop> defines the uppermost allowable screen row for the mouse cursor or 0 if omitted. This value may range from 0 to the value of *<nBottom>*.

<nLeft> defines the leftmost allowable screen column for the mouse or 0 if omitted. This value may range from 0 to the value of *<nRight>*.

<nBottom> defines the bottommost screen row for the mouse cursor or MAXROW() if omitted. This value may range from the value of *<nTop>* to MAXROW().

<nRight> defines the rightmost allowable screen column for the mouse or MAXCOL() if omitted. This value may range from the value of *<nLeft>* to MAXCOL().

Returns

MSETBOUNDS() always returns NIL.

Description

MSETBOUNDS() is a function that is used for setting the region of the screen that the mouse cursor is restricted to. The default at startup is the entire screen. This setting is automatically maintained by the runtime's mouse subsystem when the screen mode is changed, for example, from the 50-line mode to the 25-line mode.

MSETCLIP() function

Define an inclusion region

Syntax

```
MSETCLIP([<nCoord list>], [<nMode>])
```

Arguments

<nCoord List> contains the coordinates of the inclusion region which is represented as a comma-separated list of four coordinates whose interpretation differs depending on mode. If *<nMode>* is LLM_COOR_TEXT, the coordinates look like this: *<nTop>*, *<nLeft>*, *<nBottom>*, *<nRight>*. If *<nMode>* is LLM_COOR_GRAPH, the coordinates are *<nX1>*, *<nY1>*, *<nX2>*, *<nY2>*.

<nMode> is one of the following two constants:

Constant	Description
LLM_COOR_TEXT	Specifies that the coordinates are given in lines and columns of text
LLM_COOR_GRAPH	Specifies that the coordinates are given in pixels

Returns

MSETCLIP() returns an array of coordinate information. The coordinates of the inclusion area are given in pixels and then in row/col format:

```
aRegions := {nIncX1, nIncY1, nIncX2, nIncY2,;
   nIncR1, nIncC1, nIncR2, nIncC2,;
   nCoorType}
```

Description

MSETCLIP() controls mouse pointer movements. It allows you to restrict movement to a region. When an inclusion is defined and the user tries to move the mouse pointer out of the rectangle, it remains stuck at the edge of the area, but is still visible. This mode should be used to restrict the range of user choices for moving or clicking the mouse.

Notes

The inclusion area is initialized to be the entire screen. To eliminate an inclusion region you just pass the value LLM_INCLUDE as the first parameter or pass the four parameters for maximum screen values 0, 0, MAXCOL(), MAXROW().

Examples

- This example shows how the MSETCLIP() function works:

```
// Define an inclusion region and save the current
// inclusion region in the array aOldInc
    aOldInc := MSETCLIP(5, 5, 20, 75,LLM_COOR_TEXT)
// Code for selecting objects and buttons
// Restore old inclusion region
MSETCLIP(aOldInc[5],;
    aOldInc[6],;
    aOldInc[7],;
    aOldInc[8],;
    LLG_VIDEO_TEXT)
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also MHIDE(), MSETCURSOR(), MSHOW()

MSETCURSOR() function

Determine a mouse's visibility

Syntax

```
MSETCURSOR( [<lVisible>] ) → lIsVisible
```

Arguments

<lVisible> determines if the mouse should be visible. Set to true (.T.) to show the mouse cursor and set to false (.F.) to hide it.

Returns

MSETCURSOR() returns the mouse cursor's previous visibility state if *<lVisible>* is passed; otherwise, it returns its current visibility state.

Description

MSETCURSOR() is a function that determines whether the mouse cursor is visible or not.

MSETPOS() function

Set a new position for the mouse cursor

Syntax

```
MSETPOS( <nRow>, <nCol> ) → NIL
```

Returns

MSETPOS() always returns NIL.

Description

MSETPOS () is a function that moves the mouse cursor to a new position on the screen. After the mouse cursor is positioned, MROW() and MCOL() are updated accordingly. To control the visibility of the mouse cursor, use MSETCURSOR().

See Also

MSETCURSOR(), MROW(), MCOL()

MSHOW() function

Display the mouse pointer

Syntax

```
MSHOW([<nCol>, <nRow>, <nStyle>])
    → nOldCursorShape

MSHOW([<nCursorShape>]) → nOldCursorShape

MSHOW([<nCursorShape>] | [<nCol>, <nRow>, <nMode>])
    → nOldCursorShape
```

Arguments

<nCol> and ***<nRow>*** define mouse pointer coordinates.

<nStyle> defines the style of mouse pointer using one of the constants listed in the table below:

Text and Graph Constants

Constant	Description
LLM_COOR_TEXT	Specifies that the coordinates are passed in rows and columns of text
LLM_COOR_GRAPH	Specifies that the coordinates are passed in pixels

<nCursorShape> is a numeric value representing the mouse cursor shape. The following are the possible values predefined for this parameter:

Cursor Shape Constants

Constant	Description
LLM_CURSOR_ARROW	Standard pointer
LLM_CURSOR_SIZE_NS	North South arrow
LLM_CURSOR_SIZE_WE	West East arrow
LLM_CURSOR_SIZE_NW_SE	North-West South-East arrow
LLM_CURSOR_SIZE_NE_SW	North-East South-West arrow
LLM_CURSOR_HAND	Hand
LLM_CURSOR_FINGER	Hand with one pointing finger
LLM_CURSOR_CROSS	Cross
LLM_CURSOR_WAIT	Hourglass

Returns

MSHOW() returns the previously used cursor shape. See *<nCurorShape>* above for further information.

Description

MSHOW() displays the mouse pointer. It is generally used without parameters to simply redisplay the mouse pointer at the position where MHIDE() hid it (assuming the user has not moved the mouse).

It is possible to use two sets of parameters with this function.

- Specify the coordinates where the pointer should appear. In this case, three parameters must be passed: the mode and its coordinates. In text mode, coordinates are passed as row and column. In graphic mode, you can pass either text or graphic coordinates. Conversion is done automatically based on the font size of the current characters.
- You can also specify the mouse cursor shape to be displayed when the mouse is visible. This feature is available in graphic mode only.

It is important to hide the mouse pointer before any new screen display, and then show it again. See MHIDE() for further information on how to do this.

Note: The MSETCURSOR() function should be used in place of MSHOW() and MHIDE(). It is kept here for compatibility.

Examples

- The following example hides the mouse pointer before using an @...SAY command and then redisplays it. Next, the mouse pointer is repositioned, hidden, changed to an hour-glass, and then restored to its previous shape:

```
LOCAL nOldShape := 0
MHIDE()
   @ 10, 20 say "Hello world!!!"
MSHOW()
// Position the pointer at the center of the screen
MSHOW(MAXCOL() / 2, MAXROW() / 2, LLM_COOR_TEXT)
MHIDE()
   @ 10, 20 say "Please wait ..."
// Display an hour glass cursor
nOldShape := MSHOW(LLM_CURSOR_WAIT)
// Your code
// Restore previously used cursor
MSHOW(nOldShape)
```

Files

Library is LLIBG.LIB, header file is Llibg.ch.

See Also

MHIDE(), MSETCLIP(), MSHOW()

MSTATE() function

Return the current mouse state

Syntax

```
MSTATE() → aState | 0
```

Returns

MSTATE() Return Array

Position	Description
LLM_STATE_X	State of X position.
LLM_STATE_Y	State of Y position.
LLM_STATE_ROW	State of column position.
LLM_STATE_COL	State of line position.
LLM_STATE_LEFT	State of left mouse button. LLM_BUTTON_DOWN means down and LLM_BUTTON_UP means up.
LLM_STATE_RIGHT	State of right mouse button. LLM_BUTTON_DOWN means down and LLM_BUTTON_UP means up.
LLM_STATE_VISIBLE	State of mouse pointer. True (.T) means visible and false (.F) means invisible. (See MSHOW() and MHIDE() for more information.)
LLM_STATE_DRIVER	Indicates version of mouse driver.
LLM_STATE_SHAPE	Mouse cursor shape. (See note below.)
LLM_STATE_CLICKS_LEFT	Number of left clicks since last MSTATE() call.
LLM_STATE_CLICKS_RIGHT	Number of right clicks since last MSTATE() call.

Note: The following are the possible values predefined for this return array position: LLM_CURSOR_ARROW, LLM_CURSOR_SIZE_NS, LLM_CURSOR_SIZE_WE, LLM_CURSOR_SIZE_NW_SE, LLM_CURSOR_SIZE_NE_SW, LLM_CURSOR_HAND, LLM_CURSOR_FINGER, LLM_CURSOR_CROSS, LLM_CURSOR_WAIT. For a description of these values see the MSHOW() table of Cursor Shape Constants.

If the mouse is missing, 0 is returned.

The number of clicks (i.e., aState[LLM_STATE_CLICKS_LEFT] and aState[LLM_STATE_CLICKS_RIGHT]) is reset each time MSTATE() is called. Use MSTATE() to reset the mouse settings when needed.

Description

MSTATE() returns information on the mouse state, i.e., the current screen position of the pointer, the state of the left and right mouse buttons, the visibility status of the mouse pointer, and the version of the mouse driver.

Notes

If the version of the mouse driver (i.e., aState[LLM_STATE_DRIVER]) is NULL (""), this indicates that the mouse is not connected or is configured incorrectly. Do not hesitate to insert a version check such as the following:

```
IF MSTATE()[LLM_STATE_DRIVER]<=500
    // Consider the mouse to be absent
ELSE
    // Consider the mouse to be present
ENDIF
```

Warning! *Old mouse drivers are usually unreliable and can cause odd behavior in your application, such as a poorly drawn pointer and unsatisfactory saves and restores of the pointer and underlying pixels.*

Examples

- This example shows how to use MSTATE() function:

```
// Show the mouse pointer
MSHOW()
DO WHILE INKEY() != K_ESC
                // Retrieve mouse state
   aState := MSTATE()
                // Line position
   @ 24,  0 SAY aState[LLM_STATE_ROW]
                // Column position
   @ 24, 10 SAY aState[LLM_STATE_COL]
                // State of left button
   @ 24, 20 SAY "Left" +
If(aState[LLM_STATE_LEFT]==LLM_BUTTON_DOWN,;
   "Down"  ,;
   "Up")
                // State of right button
   @ 24, 40 SAY "Right " + If(aState[LLM_STATE_RIGHT]==    ;
   LLM_BUTTON_DOWN ,;
   "Down"  ,;
   "Up")
ENDDO
   // Hide the mouse pointer
MHIDE()
```

Files Library is LLIBG.LIB, header file is Llibg.ch.

See Also MHIDE(), MSETCURSOR(), MSHOW()

NETERR() function

Determine if a network command has failed

Syntax

```
NETERR([<lNewError>]) → lError
```

Arguments

<lNewError>, if specified, sets the value returned by NETERR() to the specified status. *<lNewError>* can be either true (.T.) or false (.F.). Setting NETERR() to a specified value allows the runtime error handler to control the way certain file errors are handled. For more information, refer to Errorsys.prg.

Returns

NETERR() returns true (.T.) if a USE or APPEND BLANK fails. The initial value of NETERR() is false (.F.). If the current process is not running under a network operating system, NETERR() always returns false (.F.).

Description

NETERR() is a network function. It is a global flag set by USE, USE...EXCLUSIVE, and APPEND BLANK in a network environment. It is used to test whether any of these commands have failed by returning true (.T.) in the following situations:

NETERR() Causes

Command	Cause
USE	USE EXCLUSIVE by another process
USE...EXCLUSIVE	USE EXCLUSIVE or USE by another process
APPEND BLANK	FLOCK() or RLOCK() of LASTREC() + 1 by another user

NETERR() is generally applied in a program by testing it following a USE or APPEND BLANK command. If it returns false (.F.), you can perform the next operation. If the command is USE, you can open index files. If it is APPEND BLANK, you can assign values to the new record with a REPLACE or @...GET command. Otherwise, you must handle the error by either retrying the USE or APPEND BLANK, or terminating the current operation with a BREAK or RETURN.

Examples

- This example demonstrates typical usage of NETERR(). If the USE succeeds, the index files are opened and processing continues. If the USE fails, a message displays and control returns to the nearest BEGIN SEQUENCE construct:

```
USE Customer SHARED NEW
IF !NETERR()
   SET INDEX TO CustNum, CustOrders, CustZip
ELSE
   ? "File is in use by another"
   BREAK
ENDIF
```

Files Library is CLIPPER.LIB.

See Also APPEND BLANK, FLOCK(), RLOCK(), USE

NETNAME() function

Return the current workstation identification

Syntax

```
NETNAME() → cWorkstationName
```

Returns

NETNAME() returns the workstation identification as a character string up to 15 characters in length. If the workstation identification was never set or the application is not operating under the IBM PC Network, it returns a null string ("").

Examples

- This example demonstrates the NETNAME() result when a workstation is started as a network node with a station identifier of "STATION 1":

```
? NETNAME()              // Result: STATION 1
```

- This example demonstrates the NETNAME() result when a workstation is started as a stand-alone unit:

```
? NETNAME()              // Result: ""
```

Files

Library is CLIPPER.LIB.

NEXTKEY() function

Read the pending key in the keyboard buffer

Syntax

```
NEXTKEY() → nInkeyCode
```

Returns

NEXTKEY() returns an integer numeric value ranging from -39 to 386 for keyboard events and integer values from 1001 to 1007 for mouse events. This value identifies either the key extracted from the keyboard buffer or the mouse event that last occurred. If the keyboard buffer is empty and no mouse events are taking place, NEXTKEY() returns zero. If SET TYPEAHEAD is zero, NEXTKEY() always returns zero.

Description

NEXTKEY() is a function that reads the keystroke pending in the keyboard buffer or the next mouse event without removing it. The value returned is the INKEY() code of the key pressed—the same value as returned by INKEY() and LASTKEY(). NEXTKEY() returns values for all ASCII characters as well as function, Alt+function, Ctrl+function, Alt+letter, and Ctrl+letter key combinations.

NEXTKEY() is like the INKEY() function but differs in one fundamental respect. INKEY() removes the pending key from the keyboard buffer and updates LASTKEY() with the value of the key. By contrast NEXTKEY() reads, but does not remove the key from the keyboard buffer, and does not update LASTKEY().

Since NEXTKEY() does not remove the key from the keyboard buffer, it can be used to poll the keyboard, and then pass control to a routine that uses a wait state or INKEY() to actually fetch the key from the buffer.

For a complete list of INKEY() codes and Inkey.ch constants, refer to the "CA-Clipper Inkey Codes Appendix" in the *Error Messages and Appendices Guide*.

Examples

- This example places an Esc key in the keyboard buffer, and then shows the differences between INKEY(), LASTKEY(), and NEXTKEY():

```
#include "Inkey.ch"
//
CLEAR TYPEAHEAD
KEYBOARD CHR(K_ESC)
//
? NEXTKEY(), LASTKEY()          // Result: 27 0
? INKEY(), LASTKEY()            // Result: 27 27
? NEXTKEY()                     // Result: 0
```

Files Library is EXTEND.LIB, header file is Inkey.ch.

See Also INKEY(), KEYBOARD, LASTKEY(), SET TYPEAHEAD

NOSNOW() function

Toggle snow suppression

Syntax

NOSNOW(<*lToggle*>) → *NIL*

Arguments

<lToggle> is a logical value that toggles the current state of snow suppression. A value of true (.T.) enables the snow suppression on, while a value of false (.F.) disables snow suppression.

Description

NOSNOW() is used to suppress snow on CGA monitors. Typically, use NOSNOW() in the configuration section of your application to give the user the option to suppress snow.

Files

Library is CLIPPER.LIB.

NOTE* command

Place a single-line comment in a program file

Syntax

```
NOTE [<commentText>]
```

Arguments

<commentText> is a string of characters placed after the comment indicator.

Description

NOTE is command synonym for the single-line comment indicator (*). All characters after NOTE are ignored until the CA-Clipper compiler encounters an end of line (carriage return/line feed). This means a single-line comment cannot be continued with the semicolon (;) onto a new line.

If you need a multi-line or inline comment, begin the comment block with a slash-asterisk (/*) symbol and end the comment block with an asterisk-slash (*/) symbol. If you need to comment out a block of code, use the conditional compilation directives #ifdef...#endif instead of multiline comments. This is important since nested comments are illegal.

NOTE is a compatibility command and therefore not recommended. It is superseded by the C-style comment symbols slash-asterisk (/*) and asterisk-slash (*/), as well as the double-slash (//). It is also superseded by the dBASE-style comment symbols, asterisk (*) and the double-ampersand (&&).

For a complete discussion on comment indicators, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide.*

Examples

- These examples show the various comment symbols supported by CA-Clipper:

```
// This is a comment
/* This is a comment */
* This is a comment
&& This is a comment
NOTE This is a comment
```

ORDBAGEXT() function

Return the default order bag RDD extension

Syntax

```
ORDBAGEXT() → cBagExt
```

Returns

ORDBAGEXT() returns a character expression.

Description

ORDBAGEXT() is an order management function that returns a character expression that is the default order bag extension of the current or aliased work area. *cBagExt* is determined by the RDD active in the current work area.

Notes

- ORDBAGEXT() supersedes the INDEXEXT() function, which is not recommended.
- ORDBAGEXT() returns the default index extension of the driver loaded, not the actual index file extension.

Examples

```
USE sample VIA "DBFNTX"
? ORDBAGEXT()      //  Returns .ntx
```

See Also

INDEXEXT(), ORDBAGNAME()

ORDBAGNAME() function

Return the order bag name of a specific order

Syntax

ORDBAGNAME(<*nOrder*> | <*cOrderName*>) → *cOrderBagName*

Arguments

<*nOrder*> is an integer that identifies the position in the order list of the target order whose order bag name is sought.

<*cOrderName*> is a character string that represents the name of the target order whose order bag name is sought.

Returns

ORDBAGNAME() returns a character string, the order bag name of the specific order.

Description

ORDBAGNAME() is an order management function that lets you access the name of the order bag in which <*cOrderName*> resides. You may identify the order as a character string or with an integer that represents its position in the order list. In case of duplicate names, ORDBAGNAME() only recognizes the first matching name.

Note: ORDBAGNAME(0) works as ORDBAGNAME(INDEXORD())

Examples

- The following example uses ORDBAGNAME() with the default DBFNTX driver:

```
USE Customer VIA "DBFNTX" NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDBAGNAME( 2 )             // Returns: CuName
ORDBAGNAME( 1 )             // Returns: CuAcct
ORDBAGNAME( 3 )             // Returns: CuZip
```

- In this example, Customer.cdx contains three orders named CuAcct, CuName, CuZip:

```
USE Customer VIA "DBFCDX" NEW
SET INDEX TO Customer
ORDBAGNAME( "CuAcct" )      // Returns: Customer
ORDBAGNAME( "CuName" )      // Returns: Customer
ORDBAGNAME( "CuZip" )       // Returns: Customer
```

See Also ORDBAGEXT(), INDEXORD()

ORDCOND() function

Specify conditions for ordering

Syntax

```
ORDCOND ([ FOR < lCondition > ]
         [ ALL ] [ WHILE <lCondition > ]
         [ EVAL < bBlock > [ EVERY < nInterval > ] ]
         [ RECORD < nRecord > ] [ NEXT < nNumber > ]
         [ REST ] [ DESCENDING ])
```

Arguments

FOR < *lCondition* > specifies the conditional set of records on which to create the order. Only those records that meet the condition are included in the resulting order. < *lCondition* > is an expression that may be no longer than 250 characters under the DBFNTX and DBFNDX drivers. The maximum value for these expressions is determined by the RDD. The FOR condition is stored as part of the order bag and used when updating or recreating the index using the ORDCREATE() or ORDREBUILD() functions. Duplicate key values are not added to the order bag.

Drivers that do not support the FOR condition produce an "unsupported" error.

The FOR clause provides the only scoping that is maintained for all database changes. All other scope conditions create orders that do not reflect database updates.

ALL specifies all orders in the current or specified work area. ALL is the default scope for ORDCOND().

WHILE < *lCondition* > specifies another condition that must be met by each record as it is processed. As soon as the record is encountered that causes the condition to fail, the ORDCREATE() function terminates. If a WHILE clause is specified, the data is processed in the controlling order. The WHILE condition is transient (i.e., it is not stored in the file, not used for index updates, and not used for index updates and ORDREBUILD() purposes). The WHILE clause creates *temporary* orders, but these orders are not updated.

Drivers that do not support the WHILE condition produce an "unsupported" error.

Using the WHILE clause is more efficient and faster than using the FOR clause. The WHILE clause only processes data for which <*lCondition* > is true from the current position. The FOR clause, however, processes all data in the data source.

EVAL < *bBlock* > evaluates a code block every < *nInterval* >, where < *nInterval* > is a value specified by the EVERY clause. The default value is 1. This is useful in producing a status bar or odometer that monitors the indexing progress. The return value of < *bBlock* > must be a logical data type. If < *bBlock* > returns false (.F.), indexing halts.

EVERY < *nInterval* > is a clause containing a numeric expression that modifies how often < *bBlock* > is EVALuated. The EVERY option of the EVAL clause offers a performance enhancement by evaluating the condition for every *n*th record instead of evaluating *every* record ordered. The EVERY keyword is ignored if you specify no EVAL conditions.

RECORD < *nRecord* > specifies the processing of the specified record.

NEXT < *nNumber* > specifies the portion of the database to process. If you specify NEXT, the database is processed in the controlling order for the < *nNumber* > number of identities. The scope is transient (i.e., it is not stored in the order and not used for ORDREBUILDing purposes).

REST specifies the processing of all records from the current position of the record pointer to the end of file (EOF).

DESCENDING specifies that the keyed pairs be sorted in decreasing order of value. If you create a DESCENDING index, you will not need to use the DESCEND() function during a SEEK. DESCENDING is an attribute of the file, where it is stored and used for ORDREBUILDing purposes.

Description

ORDCOND() is designed to set up the conditions for creating a new order (using the ORDCREATE() function) or rebuilding an existing order (using the ORDREBUILD() function). Do not use the ORDCOND() function if you wish to create or rebuild an entire index file; it is only used for setting particular conditions for the order.

ORDCREATE() or ORDREBUILD() should be used immediately following the ORDCOND() function.

If the DESCENDING clause is not specified, the order is then assumed to be ascending.

The EVAL clause lets you specify a code block to be evaluated as each record is placed in the order. The EVERY clause lets you modify how often < *bBlock* > is called. Instead of evaluating each record as it is placed in the order, evaluation only occurs as every < *nInterval* > records are placed in the order. This can be used, for example, to create a gauge that displays how far the ORDCREATE() or ORDREBUILD() has progressed so far.

The FOR clause provides the only order scoping that is permanent and that can be maintained across the life of the application. The string passed as the FOR condition is stored within the order for later use in maintaining the order. Though only accessing part of the database, orders created using this clause exist as long as the database is active. The FOR clause lets you create *maintainable scoped* orders.

The WHILE, NEXT, REST, and RECORD clauses process data from the current position of the database cursor in the default or specified work area. If you specify these clauses, the order list remains open and the active order is used to organize the database while it is being created. These clauses let you create temporary (*non-maintainable*) orders. Orders created using these clauses contain records in which < *lCondition* > is true(.T.) at the location of the record pointer.

Examples

- The following example creates a conditional order based on a FOR clause. This index contains only records whose field TransDate contains a date greater than or equal to January 1, 1992:

```
USE Invoice NEW
ORDCOND ( FOR ( Invoice->TransDate >= CTOD
( "01/01/92" ) ) )
ORDCREATE( "InvDate" , , "Invoice->TransDate" )
```

- The following example creates an order that calls a routine, "MyMeter," during its creation:

```
USE Invoice NEW
ORDCOND ( EVAL { | | MyMeter() } EVERY MTR_INCREMENT )
ORDCREATE("Invoice" , , "Invoice->Customer")
```

See Also

DESCEND(), INDEX, INDEXORD(), ORDCREATE(), ORDREBUILD(), REINDEX, SEEK, SET INDEX, SET ORDER, SORT, USE

ORDCONDSET() function

Set the condition and scope for an order

Syntax

```
ORDCONDSET([<cForCondition>],
    [<bForCondition>],
    [<lAll>],
    [<bWhileCondition>],
    [<bEval>],
    [<nInterval>],
    [<nStart>],
    [<nNext>],
    [<nRecord>],
    [<lRest>],
    [<lDescend>],
    [<lAdditive>],
    [<lCurrent>],
    [<lCustom>],
    [<lNoOptimize>]) → lSuccess
```

Arguments

<cForCondition> is a string that specifies the FOR condition for the order. This string is returned by DBORDERINFO(DBOI_CONDITION, [*<cIndexFile>*], *<cOrder>*). If you do not need this information, you can specify a null string ("").

<bForCondition> is a code block that defines a FOR condition that each record within the scope must meet in order to be processed. If a record does not meet the specified condition, it is ignored and the next record is processed. Duplicate key values are not added to the index file when a FOR condition is used. The default is NIL.

This condition (not *<cForCondition>*) is the one that is actually used to create the order. Unlike the WHILE condition and other scoping information, the FOR condition is stored as part of the index file and is used when updating or rebuilding the order with DBREINDEX(). Any limitations on the FOR condition are determined by the RDD (see the "Replaceable Database Driver Architecture" chapter in the *Drivers Guide* for information about RDD limitations).

<lAll> is specified as true (.T.) to define a scope of all records. Use false (.F.) if you want to indicate other record scoping conditions (i.e., *<nNext>*, *<nRecord>*, or *<lRest>*). The default is false (.F.).

<bWhileCondition> is a code block that defines a condition that each record must meet in order to be processed. If no scope is specified, using a *<bWhileCondition>* changes the default scope to *<lRest>*. As soon as a record is encountered that causes the condition to fail, the operation terminates.

The WHILE condition is used only to create the order. It is not stored in the index file or used for updating or reindexing purposes. The default is NIL.

<bEval> is a code block that is evaluated at intervals specified by *<nInterval>*. This is useful in producing a status bar or odometer that monitors the ordering progress. The return value of *<bEval>* must be a logical value. If *<bEval>* returns false (.F.), indexing halts. The default is NIL.

<nInterval> is a numeric expression that determines the number of times *<bEval>* is evaluated. This argument offers a performance enhancement by evaluating the condition at intervals instead of evaluating every record processed. To step through every record, you can specify a value of 0. The default is 0.

<nStart> is the starting record number. To start at the beginning of the file, specify a value of 0. The default is 0.

You define the scope using one of these three, mutually exclusive arguments (use 0 or false (.F.) for the others). The default is all records. Record scoping information is used only to create the order. It is not stored in the index file or used for index updates and reindexing purposes.

<nNext> is the number of records to process, starting at *<nStart>*. Specify 0 to ignore this argument.

<nRecord> is a single record number to process. Specify 0 to ignore this argument.

<lRest> is specified as true (.T.) to process only records from *<nStart>* to the end of the file. False (.F.) processes all records.

<lDescend> specifies whether the keyed pairs should be sorted in decreasing or increasing order of value. True (.T.) results in descending order and false (.F.) results in ascending order. The default is false (.F.).

<lAdditive> specifies whether open orders should remain open while the new order is being created. True (.T.) specifies that they should remain open. False (.F.) is the default and it specifies that all open orders should be closed.

<lCurrent> specifies whether only records in the controlling order—and within the current range as specified by ORDSETSCOPE()—will be included in this order. True (.T.) specifies that the controlling order and range should be used to limit the scope of the newly created order. False (.F.) is the default and it specifies that all records in the database file are included in the order.

<lCustom> specifies whether the new order will be a custom built order (for RDDs that use this feature). True (.T.) specifies that a custom built order will be created. A custom built order is initially empty, giving you complete control over order maintenance. The system does not automatically add and delete keys from a custom built order. Instead, you explicitly add and delete keys using ORDKEYADD() and ORDKEYDEL(). False (.F.) specifies a standard, system-maintained order. The default is false (.F.).

<lNoOptimize> specifies whether the FOR condition will be optimized (for RDDs that support this feature). True (.T.) optimizes the FOR condition, and false (.F.) does not. The default is false (.F.).

Returns

ORDCONDSET() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

By default ORDCONDSET() operates on the currently selected work area. This function can be made to operate on an unselected work area by specifying it within an aliased expression.

Unless you specify otherwise with ORDCONDSET(), new orders that you create will use default scoping rules, processing all records in the work area. ORDCONDSET() allows you to specify conditions and scoping rules that records must meet in order to be included in the next order created. Creating a new order automatically resets the work area to use the default scoping rules. Thus, if scoping is required, you must reset ORDCONDSET() each time you create a new order.

This function is essential if you want to create conditional orders using DBCREATEINDEX() because this function does not support arguments to do this.

Examples

- The following example sets the condition for the creation of orders:

```
LOCAL cFor AS STRING
LOCAL lAll, lRest, lDescend AS LOGIC
LOCAL bForCondition, bWhileCondition, ;
    bEval AS USUAL
LOCAL nStep, nStart, nNext, nRecord AS SHORTINT
// For condition string format
cFor := 'UPPER(Name) = "MELISSA"'
// Actual for condition
bForCondition := {|| UPPER(Name) = "MELISSA"}
lAll := .T.
bWhileCondition := {|| .T.}          // While all
bEval := {|| Name + City}            // Indexing code
                                     // block
nStep := 0                           // Step through all
nStart := 0                          // From top
nNext := 0                           // All
nRecord := 0                         // All records
lRest := .F.                         // All
lDescend :=  .F.                     // Ascending
ORDCONDSET(cFor, bForCondition, lAll, ;
    bWhileCondition, bEval, nStep, nStart, ;
    nNext, nRecord, lRest, lDescend)
```

Files

Library is CLIPPER.LIB.

See Also

DBCREATEINDEX(), DBORDERINFO(), INDEX, ORDKEYADD(), ORDKEYDEL(), ORDSCOPE()

ORDCREATE() function

Create an order in an order bag

Syntax

```
ORDCREATE(<cOrderBagName>,[<cOrderName>],
   <cExpKey>, <bExpKey>, [<lUnique>]) → NIL
```

Arguments

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or extension. Without the extension, CA-Clipper uses the default extension of the current RDD.

<cOrderName> is the name of the order to be created.

Note: Although both *<cOrderBagName>* and *<cOrderName>* are both optional, at least one of them must be specified.

<cExpKey> is an expression that returns the key value to place in the order for each record in the current work area. *<cExpKey>* can represent a character, date, logical, or numeric data type. The database driver determines the maximum length of the index key expression.

<bExpKey> is a code block that evaluates to a key value that is placed in the order for each record in the current work area. If you do not supply *<bExpKey>*, it is macro-compiled from *<cExpKey>*.

<lUnique> is an optional logical value that specifies whether a unique order is to be created. If *<lUnique>* is omitted, the current global _SET_UNIQUE setting is used.

Returns

ORDCREATE() always returns NIL.

Description

ORDCREATE() is an order management function that creates an order in the current work area. It works like DBCREATEINDEX() except that it lets you create orders in RDDs that recognize multiple-order bags. ORDCREATE() supersedes the DBCREATEINDEX() function because of this capability, and it is the preferred function.

The active RDD determines the order capacity of an order bag. The default DBFNTX and DBFNDX drivers only support single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBFMDX drivers).

In RDDs that support production or structural indices (e.g., DBFCDX, DBPX), if you specify a tag but do not specify an order bag, the tag is created and added to the index. If no production or structural index exists, it will be created and the tag will be added to it. When using RDDs that support multiple-order bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling order. If you do not specify a controlling order, the data file will be viewed in natural order.

If <*cOrderBagName*> does not exist, it is created in accordance with the RDD in the current or specified work area.

If <*cOrderBagName*> exists and the RDD specifies that order bags can only contain a single order, <*cOrderBagName*> is erased and the new order is added to the order list in the current or specified work area.

If <*cOrderBagName*> exists and the RDD specifies that order bags can contain multiple tags, <*cOrderName*> is created if it does not already exist; otherwise <*cOrderName*> is replaced in <*cOrderBagName*> and the order is added to the order list in the current or specified work area.

Examples

- The following example demonstrates ORDCREATE() with the DBFNDX driver:

```
USE Customer VIA "DBFNDX" NEW
ORDCREATE( "CuAcct",, "Customer->Acct" )
```

- The following example demonstrates ORDCREATE() with the default DBFNTX driver:

```
USE Customer VIA "DBFNTX" NEW
ORDCREATE( "CuAcct", "CuAcct", "Customer->Acct", ;
      {|| Customer->Acct } )
```

- The following example demonstrates ORDCREATE() with the FoxPro driver, DBFCDX:

```
USE Customer VIA "DBFCDX" NEW
ORDCREATE( "Customer", "CuAcct", "Customer->Acct" )
```

- This example creates the order "CuAcct" and adds it to the production index (order bag) "Customer." The production index will be created if it does not exist:

```
USE Customer VIA "DBFMDX" NEW
ORDCREATE( , "CuAcct", "Customer->Acct" )
```

See Also

DBCREATEIND(), INDEX, SET UNIQUE

ORDDESCEND() function

Return and optionally change the descending flag of an order

Syntax

```
ORDDESCEND([<cOrder> | <nPosition>],[<cIndexFile>],
   [<lNewDescend>]) → lCurrentDescend
```

Arguments

<cOrder>* | *<nPosition> is the name of the order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. If omitted or NIL, the controlling order is assumed.

Specifying an invalid value will raise a runtime error.

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

If *<cIndexFile>* is not open by the current process, a runtime error is raised.

<lNewDescend> is a logical value that if true (.T.) dynamically turns on the descending flag for the order, resulting in descending order. False (.F.) dynamically turns the flag off, resulting in ascending order.

Returns

If *<lNewDescend>* is not specified, ORDDESCEND() returns the current setting. If *<lNewDescend>* is specified, the previous setting is returned.

Description

ORDDESCEND() changes the ascending/descending flag at runtime only—it does not change the descending flag stored in the actual index file. To change the descending flag in the index file, see the INDEX command in the *Reference Guide, Volume 1*.

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- The following example illustrates ORDDESCEND(). Every order can be both ascending and descending:

```
USE Customer VIA "DBFCDX"
INDEX ON LastName TAG Last
INDEX ON FirstName TAG First DESCENDING

SET ORDER TO TAG Last
// Last was originally created in ascending order

// Swap it to descending
ORDDESCEND(,, .T.)
// Last will now be processed in descending order

SET ORDER TO TAG First
// First was originally created in descending order

// Swap it to ascending
ORDDESCEND(,, .F.)
// First will now be processed in ascending order
```

Files Library is CLIPPER.LIB.

See Also INDEX

ORDDESTROY() function

Remove a specified order from an order bag

Syntax

```
ORDDESTROY(<cOrderName> [, <cOrderBagName> ]) → NIL
```

Arguments

<cOrderName> is the name of the order to be removed from the current or specified work area.

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

ORDDESTROY() always returns NIL.

Description

ORDDESTROY() is an order management function that removes a specified order from multiple-order bags.

The active RDD determines the order capacity of an order bag. The default DBFNTX and the DBFNDX drivers only support single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBPX drivers).

Note: RDD suppliers may define specific behaviors for this command.

Warning! *ORDDESTROY() is not supported for DBFNDX and DBFNTX.*

Examples

- This example demonstrates ORDDESTROY() with the FoxPro driver, DBFCDX:

```
USE Customer VIA "DBFCDX" NEW
SET INDEX TO Customer, CustTemp
ORDDESTROY( "CuAcct", "Customer" )
```

See Also

DELETE TAG, ORDCREATE()

ORDFOR() function

Return the FOR expression of an order

Syntax

```
ORDFOR(<cOrderName> | <nOrder> [, <cOrderBagName>])
    → cForExp
```

Arguments

<cOrderName> is the name of the target order whose *cForExp* is sought.

<nOrder> is an integer that identifies the position in the order list of the target order whose *cForExp* is sought.

<cOrderBagName> is the name of an order bag containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

ORDFOR() returns a character expression, *cForExp*, that represents the FOR condition of the specified order. If the order was not created using the FOR clause, the return value will be an empty string (""). If the database driver does not support the FOR condition, it may either return an empty string ("") or raise an "unsupported function" error, depending on the driver.

Description

ORDFOR() is an order management function that returns the character string, *cForExp*, that represents the logical FOR condition of *<cOrderName>* or *<nOrder>*.

Note: ORDFOR(0) works as ORDFOR(INDEXORD()).

Examples

- This example retrieves the FOR condition from an order:

```
USE Customer NEW
INDEX ON  Customer->Acct   ;
    TO  Customer      ;
    FOR Customer->Acct > "AZZZZZ"

ORDFOR( "Customer" )    // Returns: Customer->Acct > "AZZZZZ"
```

See Also INDEX, ORDKEY(), ORDCREATE(), ORDNAME(), ORDNUMBER()

ORDISUNIQUE() function

Return the status of the unique flag for a given order

Syntax

```
ORDISUNIQUE([<cOrder> | <nPosition>],
   [<cIndexFile>]) → lUnique
```

Arguments

<cOrder>* | *<nPosition> is the name of the order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. If omitted or NIL, the controlling order is assumed.

Specifying an invalid order will raise a runtime error.

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

If *<cIndexFile>* is not open by the current process, a runtime error is raised.

Returns

ORDISUNIQUE() returns the status of the indicated order's unique flag as a logical value.

Description

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example shows the return value of ORDISUNIQUE() using various orders:

```
USE Customer VIA "DBFCDX"
INDEX ON LastName TAG Last UNIQUE
INDEX ON FirstName TAG First
INDEX ON Age TO j:\test\tmp\age UNIQUE

SET ORDER TO TAG Last

? ORDISUNIQUE()                 // Result: .T. for Last
? ORDISUNIQUE("First")          // Result: .F.
? ORDISUNIQUE("Age")            // Result: .T.
```

Files Library is CLIPPER.LIB.

See Also ORDDESCEND(), ORDFOR(), ORDKEY()

ORDKEY() function

Return the key expression of an order

Syntax

```
ORDKEY(<cOrderName> | <nOrder>
   [, <cOrderBagName>]) → cExpKey
```

Arguments

<cOrderName> is the name of an order, a logical ordering of a database.

<nOrder> is an integer that identifies the position in the order list of the target order whose *cExpKey* is sought.

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

Returns a character string, *cExpKey*.

Description

ORDKEY() is an order management function that returns a character expression, *cExpKey*, that represents the key expression of the specified order.

You may specify the order by name or with a number that represents its position in the order list. Using the order name is the preferred method.

The active RDD determines the order capacity of an order bag. The default DBFNTX and the DBFNDX drivers only support single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBFMDX drivers).

Note: ORDKEY(0) works as ORDKEY(INDEXORD()).

Examples

- This example retrieves the index expression from an order:

```
USE Customer NEW
INDEX ON  Customer->Acct   ;
    TO  Customer      ;
    FOR Customer->Acct > "AZZZZZ"

ORDKEY( "Customer" )     // Returns: Customer->Acct
```

See Also ORDFOR(), ORDKEY(), ORDNAME(), ORDNUMBER()

ORDKEYADD() function

Add a key to a custom built order

Syntax

```
ORDKEYADD([<cOrder> | <nPosition>],
   [<cIndexFile>],[<expKeyValue>]) → lSuccess
```

Arguments

<cOrder> | <nPosition> is the name of the order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. If omitted or NIL, the controlling order is assumed.

Specifying an invalid order, such as one that is not custom built, will raise a runtime error.

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

If *<cIndexFile>* is not open by the current process, a runtime error is raised.

<expKeyValue> is a specific key value that you want to add for the current record. The data type must match that of the order. If not specified, the order's key expression is evaluated for the current record and added to the order.

Returns

ORDKEYADD() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

ORDKEYADD() adds keys to a custom built order which is an order that is *not* automatically maintained by the DBFCDX driver. You can determine if an order is custom built using DBORDERINFO(DBOI_CUSTOM, ...). When you create such an order, it is initially empty. You must then manually add and delete keys using ORDKEYADD() and ORDKEYDEL().

Note: An existing order can be changed to a custom built order by using the DBORDERINFO() function.

ORDKEYADD() evaluates the key expression (or *<expKeyValue>*, if specified), and then adds the key for the current record to the order. If the order has a FOR condition, the key will be added only if that condition is met, and then only if it falls within the current scoping range.

Note: You can add several keys for the same record with consecutive calls to ORDKEYADD().

ORDKEYADD() will fail if:

- The record pointer is positioned on an invalid record (i.e., at EOF())
- The specified order is not custom built
- The specified order does not exist
- No order was specified and there is no controlling order

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example creates a custom index and adds every fiftieth record to it:

```
USE Customer VIA "DBFCDX"
// Create custom-built order that is initially empty
INDEX ON LastName TO Last CUSTOM

// Add every 50th record
FOR n := 1 TO RECCOUNT() STEP 50
   GOTO n
   ORDKEYADD()
NEXT
```

Files

Library is CLIPPER.LIB.

See Also

DBORDERINFO(), ORDFOR(), ORDKEYDEL(), ORDSCOPE()

ORDKEYCOUNT() function

Return the number of keys in an order

Syntax

```
ORDKEYCOUNT([<cOrder> | <nPosition>],
   [<cIndexFile>]) → nKeys
```

Arguments

<cOrder>* | *<nPosition> is the name of the order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. If omitted or NIL, the controlling order is assumed.

Specifying an invalid order will raise a runtime error.

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

If *<cIndexFile>* is not open by the current process, a runtime error is raised.

Returns

ORDKEYCOUNT() returns the number of keys in the specified order.

Description

ORDKEYCOUNT() counts the keys in the specified order and returns the result as a numeric value. If the order is not conditional and no scope has been set for it, ORDKEYCOUNT() is identical to RECCOUNT(), returning the number of records in the database file. However, for a conditional order, there may be fewer keys than there are records, since some records may not meet the order's FOR condition or may not fall inside the scope specified by ORDSCOPE()—in counting the keys, ORDKEYCOUNT() respects the currently defined scope and for condition.

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example demonstrates using ORDKEYCOUNT() with various orders:

```
USE customer
// Assume 1000 total records,
// 500 less than thirty years old, and
// 895 making less than 50,000

INDEX ON Age TO Age
INDEX ON First TO First FOR Age < 30
INDEX ON Last TO Last FOR Salary < 50000

// Age is the controlling order
SET INDEX TO Age, First, Last

? RECCOUNT()                    // Result: 1000
? ORDKEYCOUNT()                 // Result: 1000

? ORDKEYCOUNT("First")          // Result: 500
? ORDKEYCOUNT(3)                // Result: 895
```

Files Library is CLIPPER.LIB.

See Also ORDKEYGOTO(), ORDKEYNO(), ORDSCOPE()

ORDKEYDEL() function

Delete a key from a custom built order

Syntax

```
ORDKEYDEL([<cOrder> | <nPosition>],
   [<cIndexFile>],
   [<expKeyValue>]) → lSuccess
```

Arguments

<cOrder>* | *<nPosition> is the name of the order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. If omitted or NIL, the controlling order is assumed.

Specifying an invalid order, such as one that is not custom built, will raise a runtime error.

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

If *<cIndexFile>* is not open by the current process, a runtime error is raised.

<expKeyValue> is a specific key value that you want to delete for the current record. The data type must match that of the order. If not specified, the order's key expression is evaluated for the current record and deleted from the order.

Returns

ORDKEYDEL() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

ORDKEYDEL() deletes a key from a custom built order which is an order that is *not* automatically maintained by the DBFCDX driver. You can determine if an order is custom built using DBORDERINFO(DBOI_CUSTOM, ...). When you create such an order, it is initially empty. You must then manually add and delete keys using ORDKEYADD() and ORDKEYDEL().

Note: An existing order can be changed to a custom built order by using the DBORDERINFO() function.

ORDKEYDEL() evaluates the key expression (or *<expKeyValue>*, if specified), and then deletes the key for the current record from the order.

ORDKEYDEL() will fail if:

- The record pointer is positioned on an invalid record (for example, EOF() returns true (.T.) or the record pointer is positioned on a record that falls outside the order's scope or for condition)
- The specified order is not custom built
- The specified order does not exist
- No order was specified and there is no controlling order

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example creates a custom index, adds every fiftieth record to it, and deletes every hundredth record:

```
USE Customer VIA "DBFCDX"
// Create custom-built order that is initially empty
INDEX ON LastName TO Last CUSTOM

// Add every 50th record
FOR n := 1 TO RECCOUNT() STEP 50
    GOTO n
    ORDKEYADD()
NEXT

// Remove every 100th record
FOR n := 1 TO RECCOUNT() STEP 100
    GOTO n
    ORDKEYDEL()
NEXT
```

Files Library is CLIPPER.LIB.

See Also DBORDERINFO(), ORDFOR(), ORDKEYADD(), ORDSCOPE()

ORDKEYGOTO() function

Move to a record specified by its logical record number in the controlling order

Syntax

```
ORDKEYGOTO(<nKeyNo>) → lSuccess
```

Arguments

<nKeyNo> is the logical record number. If the value specified does not satisfy the scope or FOR condition for the order, the record pointer is positioned at the end of file.

Returns

ORDKEYGOTO() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

ORDKEYGOTO() is the complement to ORDKEYNO(). ORDKEYNO() returns the logical record number (i.e., its position in the controlling order) of the current record, and ORDKEYGOTO() moves the record pointer to the specified logical record.

Tip: This function can be useful when displaying scroll bars. If the user clicks on a certain position on the scroll bar, you can move to the corresponding record by calling ORDKEYGOTO().

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example shows the difference between physical and logical record number:

```
USE Customer
SET INDEX TO First    // Make records in first name
                      // order
ORDKEYGOTO(100)       // Go to the 100th logical record
? RECNO()             // Returns the physical record
                      // number
? ORDKEYNO()          // Returns 100, the logical
                      // record no
```

Files Library is CLIPPER.LIB.

See Also DBGOTO(), ORDKEYCOUNT(), ORDKEYNO()

ORDKEYNO() function

Get the logical record number of the current record

Syntax

```
ORDKEYNO([<cOrder> | <nPosition>],
   [<cIndexFile>]) → nKeyNo
```

Arguments

<cOrder>* | *<nPosition> is the name of the order or a number representing its position in the order list. Using the order name is the preferred method since the position may be difficult to determine using multiple-order index files. If omitted or NIL, the controlling order is assumed.

Specifying an invalid order will raise a runtime error.

<cIndexFile> is the name of an index file, including an optional drive and directory (no extension should be specified). Use this argument with *<cOrder>* to remove ambiguity when there are two or more orders with the same name in different index files.

If *<cIndexFile>* is not open by the current process, a runtime error is raised.

Returns

ORDKEYNO() returns the relative position of the current record in the specified order as a numeric value. ORDKEYNO() respects the scope and FOR condition of the order by returning zero if the record pointer is positioned on an invalid record or if EOF() is true (.T.).

Description

ORDKEYNO() returns the *logical* record number of a key in an order. This is in contrast to the *physical* record number (returned using the RECNO() function), which is the relative position of the record in the physical database file.

Tip: This function can be useful for displaying scroll bars and messages, such as "Record 9 of 123," when viewing records in a browser.

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example shows the difference between physical and logical record number:

```
USE Customer              // Assuming 1000 records
SET INDEX TO First        // Make records in first order

GO TOP                    // Position the data pointer at
                          // the first record

? ORDKEYNO()              // Result: 1

DBSKIP(10)
? ORDKEYNO()              // Result: 11
? RECNO()                 // Result: Physical record number

DBGOBOTTOM()
? ORDKEYNO()              // Result: 1000
? RECNO()                 // Result: Physical record number
```

Files

Library is CLIPPER.LIB.

See Also

ORDKEYCOUNT(), ORDKEYGOTO(), ORDSCOPE(), RECNO()

ORDKEYVAL() function

Get the key value of the current record from the controlling order

Syntax

```
ORDKEYVAL() → uKeyValue
```

Returns

ORDKEYVAL() returns the current record's key value. The data type of the return value is the same as that of the key expression used to create the order. Use VALTYPE() to determine the data type.

ORDKEYVAL() returns NIL if:

- There is no controlling order
- The record pointer is at the end of file (EOF() returns true (.T.))
- There is no key defined for this record (for example, you have positioned the record pointer to a record that does not meet the order's FOR condition or that lies outside of its specified scope)

Description

The key value is retrieved from the controlling order, *not* the database file. This makes the retrieval faster because no time is spent reading in the actual record.

Tip: ORDKEYVAL() is fast, but if you are going to use the value more than once, it is faster to store the result in a local variable. Then use the local variable rather than calling ORDKEYVAL() repeatedly for the same record.

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example displays the values for all keys in an order without ever reading the individual records into memory:

```
FUNCTION DISPLAYKEYS()
   LOCAL cKey, cFirst, cLast

   USE Customer
   // Assuming both LastName and FirstName are
   // 20 characters
   INDEX ON LastName + FirstName TO LastFir

   DO WHILE !Customer->EOF()
      cKey := Customer->ORDKEYVAL()          // Get key
                                             // value
      cLast := LEFT(cKey, 20)                // Get last
                                             // name
      cFirst := RIGHT(cKey, 20)              // Get first
                                             // name
      ? cLast, cFirst
      Customer->DBSKIP()
   ENDDO

   CLOSE
```

Files Library is CLIPPER.LIB.

See Also ORDSCOPE()

ORDLISTADD() function

Add orders to the order list

Syntax

```
ORDLISTADD(<cOrderBagName> [, <cOrderName>]) → NIL
```

Arguments

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

<cOrderName> the name of the specific order from the order bag to be added to the order list of the current work area. If you do not specify *<cOrderName>*, all orders in the order bag are added to the order list of the current work area.

Returns

ORDLISTADD() always returns NIL.

Description

ORDLISTADD() is an order management function that adds the contents of an order bag, or a single order in an order bag, to the order list. This function lets you extend the order list without issuing a SET INDEX command that, first, clears all the active orders from the order list.

Any orders already associated with the work area continue to be active. If the newly opened order bag contains the only order associated with the work area, it becomes the controlling order; otherwise, the controlling order remains unchanged.

After the new orders are opened, the work area is positioned to the first logical record in the controlling order.

ORDLISTADD() is similar to the SET INDEX command or the INDEX clause of the USE command, except that it does not clear the order list prior to adding the new order(s).

ORDLISTADD() supersedes the DBSETINDEX() function.

The active RDD determines the order capacity of an order bag. The default DBFNTX and the DBFNDX drivers only support single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX driver). When using RDDs that support multiple-order bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling order. If you do not specify a controlling order, the data file will be viewed in first order.

Examples

- In this example Customer.cdx contains three orders, CuAcct, CuName, and CuZip. ORDLISTADD() opens Customer.cdx but only uses the order named CuAcct:

```
USE Customer VIA "DBFCDX" NEW
ORDLISTADD( "Customer", "CuAcct" )
```

See Also

DBSETINDEX(), INDEX, SET INDEX, USE

ORDLISTCLEAR() function

Clear the current order list

Syntax

```
ORDLISTCLEAR() → NIL
```

Returns

ORDLISTCLEAR() always returns NIL.

Description

ORDLISTCLEAR() is an order management function that removes all orders from the order list for the current or aliased work area. When you are done, the order list is empty.

This function supersedes the function DBCLEARINDEX().

Examples

```
USE Sales NEW
SET INDEX TO SaRegion, SaRep, SaCode
.
. < statements >
.
ORDLISTCLEAR()    // Closes all the current indexes
```

See Also

SET INDEX , DBCLEARINDEX()

ORDLISTREBUILD() function

Rebuild all orders in the order list of the current work area

Syntax

```
ORDLISTREBUILD() → NIL
```

Returns

ORDLISTREBUILD() always returns NIL.

Description

ORDLISTREBUILD() is an order management function that rebuilds all the orders in the current or aliased order list.

To only rebuild a single order use the function ORDCREATE().

Unlike ORDCREATE(), this function rebuilds all orders in the order list. It is equivalent to REINDEX.

Examples

```
USE Customer NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDLISTREBUILD()     // Causes CuAcct, CuName, CuZip to
                     // be rebuilt
```

See Also

REINDEX, INDEX ON, ORDCREATE()

ORDNAME() function

Return the name of an order in the order list

Syntax

```
ORDNAME(<nOrder>[,<cOrderBagName>]) → cOrderName
```

Arguments

<nOrder> is an integer that identifies the position in the order list of the target order whose database name is sought.

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<xcOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

ORDNAME() returns the name of the specified order in the current order list or the specified order bag if opened in the current order list.

Description

ORDNAME() is an order management function that returns the name of the specified order in the current order list.

If *<cOrderBagName>* is an order bag that has been emptied into the current order list, only those orders in the order list that correspond to *<cOrderBagName>* order bag are searched.

The active RDD determines the order capacity of an order bag. The default DBFNTX and the DBFNDX drivers only support single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBPX drivers).

Note: ORDNAME(0) works as ORDNAME(INDEXORD()).

Examples

- This example retrieves the name of an order using its position in the order list:

```
USE Customer NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDNAME( 2 )                    // Returns: CuName
```

- This example retrieves the name of an order given its position within a specific order bag in the order list:

```
USE Customer NEW
SET INDEX TO Temp, Customer
// Assume Customer contains CuAcct, CuName, CuZip
ORDNAME( 2, "Customer" )        // Returns: CuName
```

See Also

ORDFOR(), ORDKEY(), ORDNUMBER()

ORDNUMBER() function

Return the position of an order in the current order list

Syntax

```
ORDNUMBER(<cOrderName>[, <cOrderBagName>]) → nOrderNo
```

Arguments

<cOrderName> the name of the specific order whose position in the order list is sought.

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

ORDNUMBER() returns an integer that represents the position of the specified order in the order list.

Description

ORDNUMBER() is an order management function that lets you determine the position in the current order list of the specified order. ORDNUMBER() searches the order list in the current work area and returns the position of the first order that matches *<cOrderName>*. If *<cOrderBagName>* is the name of an order bag newly emptied into the current order list, only those orders in the order list that have been emptied from *<cOrderBagName>* are searched.

If *<cOrderName>* is not found, ORDNUMBER() raises a recoverable runtime error.

The active RDD determines the order capacity of an order bag. The default DBFNTX driver only supports single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBPX drivers).

Examples

```
USE Customer VIA "DBFNTX" NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDNUMBER( "CuName" )           // Returns: 2
```

See Also

INDEXORD()

ORDSCOPE() function

Set or clear the boundaries for scoping key values in the controlling order

Syntax

```
ORDSCOPE(<nScope>, [<expNewValue>]) → uCurrentValue
```

Arguments

<nScope> is a number specifying the top (TOPSCOPE) or bottom (BOTTOMSCOPE) boundary.

Note: To use the TOPSCOPE and BOTTOMSCOPE constants, you must include (#include) the Ord.ch header file in your application.

<expNewValue> is the top or bottom range of key values that will be included in the controlling order's current scope. *<expNewValue>* can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.

Omitting *<expNewValue>* or specifying it as NIL has the special effect of resetting the specified scope to its original default. The default top range is the first logical record in the controlling order, and the default bottom range is the last logical record.

Returns

If *<expNewValue>* is not specified, ORDSCOPE() returns and clears the current setting. If *<expNewValue>* is specified, the function sets it and the previous setting is returned.

Description

The range of values specified using ORDSCOPE() is inclusive. In other words, the keys included in the scope will be greater than or equal to the top boundary and less than or equal to the bottom boundary.

Note: To return current settings without changing them, call the DBORDERINFO() function using the DBOI_SCOPETOP and DBOI_SCOPEBOTTOM constants.

Examples

- This example illustrates using ORDSCOPE() to set various scoping limitations on an order:

```
USE Friends
SET INDEX TO Age

// Make 25 the lowest age in range
ORDSCOPE(TOPSCOPE, 25)

// Make 30 the highest age in range
ORDSCOPE(BOTTOMSCOPE, 30)
LIST Age                          // Shows records with
                                  // 25 <= Age <= 30

// Change highest age to 35
ORDSCOPE(BOTTOMSCOPE, 35)
LIST Age                          // Shows records with
                                  // 25 <= Age <= 35

// Reset top boundary
ORDSCOPE(TOPSCOPE, NIL)
LIST Age                          // Shows records with
                                  // Age <= 35

// Reset bottom boundary
ORDSCOPE(BOTTOMSCOPE, NIL)
LIST Age                          // Shows all records
```

Files Library is CLIPPER.LIB, header file is Ord.ch.

See Also DBORDERINFO(), SET SCOPE, SET SCOPEBOTTOM, SET SCOPETOP

ORDSETFOCUS() function

Set focus to an order in an order list

Syntax

```
ORDSETFOCUS([<cOrderName> | <nOrder>]
   [,<cOrderBagName>]) → cPrevOrderNameInFocus
```

Arguments

<cOrderName> is the name of the selected order, a logical ordering of a database. ORDSETFOCUS() ignores any invalid values of *<cOrderName>*.

<nOrder> is a number representing the position in the order list of the selected order.

<cOrderBagName> is the name of a disk file containing one or more orders. You may specify *<cOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<cOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Returns

ORDSETFOCUS() returns the order name of the previous controlling order.

Description

ORDSETFOCUS() is an order management function that returns the order name of the previous controlling order and, optionally, sets the focus to an new order.

If you do not specify *<cOrderName>* or *<nOrder>*, the name of the currently controlling order is returned and the controlling order remains unchanged.

All orders in an order list are properly updated no matter what *<cOrderName>* is the controlling order. After a change of controlling orders, the record pointer still points to the same record.

The active RDD determines the order capacity of an order bag. The default DBFNTX driver only supports single-order bags, while other RDDs may support multiple-order bags (e.g., the DBFCDX and DBPX drivers).

Note: ORDSETFOCUS() supersedes INDEXORD().

Examples

```
USE Customer VIA "DBFNTX" NEW
SET INDEX TO CuAcct, CuName, CuZip
? ORDSETFOCUS( "CuName" )     // Displays: "CuAcct"
? ORDSETFOCUS()                  // Displays: "CuName"
```

See Also

SET INDEX, SET ORDER, INDEXORD()

ORDSETRELATION() function

Relate a specified work area to the current work area

Syntax

```
ORDSETRELATION(<nArea> | <cAlias>,<bKey>, [<cKey>])
    → NIL
```

Arguments

<nArea> is the number of the child work area.

<cAlias> is the alias of the child work area.

<bKey> is a code block that expresses the relational expression in executable form.

<cKey> is an optional string value that expresses the relational expression in textual form. If *<cKey>* is supplied, it must be equivalent to *<bKey>*. If *<cKey>* is omitted, ORDSETRELATION() returns a null string ("") for the relation.

Returns

ORDSETRELATION() always returns NIL.

Description

ORDSETRELATION() relates the work area specified by *<nArea>* or *<cAlias>* (the child work area) to the current work area (the parent work area). Any existing relations remain active.

Relating work areas synchronizes the child work area with the parent work area. This is achieved by automatically repositioning the child work area whenever the parent work area moves to a new record. If there is a controlling order in the child work area, moving the parent work area causes an automatic seek operation in the child work area; the seek key is based on the expression specified by *<bKey>* and/or *<cKey>*. If the child work area has no controlling order, moving the parent work area causes an automatic "go to" in the child work area; the record number for the "go to" is based on the expression specified by *<bKey>* and/or *<cKey>*.

ORDSETRELATION() is identical to DBSETRELATION() (and the SET RELATION command), but it also sets up a scope on the order in the child work area. This means that whenever you select the child work area, only the records related to the current parent record will be visible. This allows straightforward handling of one-to-many relationships. Refer to DBSETRELATION() for more information.

Examples

- This example displays each invoice with its related line items:

```
USE LineTtem NEW VIA "DBFCDX"
SET ORDER TO TAG InvNo

USE Invoice NEW VIA "DBFCDX"

// Set a selective relation from Invoice into
// LineItem
ORDSETRELATION("LineItem", {|| Invoice->InvNo}, ;
    "Invoice->InvNo")

GO TOP
DO WHILE !EOF()
    ? InvNo, InvDate          // Display invoice fields

    SELECT LineItem
    // Only records for current invoice # are visible
    LIST "   ", PartNo, Qty, Price
    SELECT Invoice            // On to next invoice
    SKIP
ENDDO
```

Files Library is CLIPPER.LIB.

See Also DBCLEARRELATION(), DBRELATION(), DBSELECT(), DBSETRELATION(), FOUND(), SET RELATION

ORDSKIPUNIQUE() function

Move the record pointer to the next or previous unique key in the controlling order

Syntax

```
ORDSKIPUNIQUE([<nDirection>]) → lSuccess
```

Arguments

<nDirection> specifies whether the function will skip to the next or previous key. Omitting this value or specifying it as 1 causes the record pointer to skip to the next unique key. Specifying a negative value makes it skip to the previous key.

Returns

ORDSKIPUNIQUE() returns true (.T.) if successful; otherwise, it returns false (.F.).

Description

ORDSKIPUNIQUE() allows you to make a non-unique order look like a unique order. Each time you use ORDSKIPUNIQUE(), you are moved to the next (or previous) unique key exactly as if you were skipping through a unique order. This function eliminates the problems associated with maintaining a unique order, while providing you with fast access to unique keys.

By default, this function operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example uses ORDSKIPUNIQUE() to build an array of unique last names beginning with the letter "J":

```
FUNCTION LASTUNIQUE()
   LOCAL aLast[0]
   SET INDEX TO Last          // Use the last name order
   ? ORDISUNIQUE()            // Result: .F.
   SET SCOPE TO "J"           // Only look at the J's

   GO TOP
   DO WHILE !EOF()            // Add all the unique J
      AADD(aLast, Last)       // last names to aLast
      ORDSKIPUNIQUE()
   ENDDO

   SET SCOPE TO               // Clear the scope
   RETURN aLast               // Return array of
                              // unique J names
```

Files Library is CLIPPER.LIB.

See Also INDEX, ORDISUNIQUE()

OS() function

Return the operating system name

Syntax

```
OS() → cOsName
```

Returns

OS() returns the operating system name as a character string.

Description

OS() is an environment function that returns the name of the disk operating system under which the current workstation is operating. The name is returned in the form of the operating system name followed by the version number.

Examples

- This example uses OS() to report the operating system under which the current workstation is running:

```
? OS()                  // Result: DOS 6.0
```

Files

Library is CLIPPER.LIB.

See Also

GETENV(), VERSION()

OUTERR() function

Write a list of values to the standard error device

Syntax

```
OUTERR(<exp list>) → NIL
```

Arguments

<exp list> is a list of values to display and can consist of any combination of data types including memo.

Returns

OUTERR() always returns NIL.

Description

OUTERR() is identical to OUTSTD() except that it writes to the standard error device rather than the standard output device. Output sent to the standard error device bypasses the CA-Clipper console and output devices as well as any DOS redirection. It is typically used to log error messages in a manner that will not interfere with the standard screen or printer output.

Examples

- This example displays an error message along with the date and time of occurrence to the screen:

```
OUTERR("File lock failure", DATE(), TIME())
```

Files Library is CLIPPER.LIB.

See Also DISPOUT(), OUTSTD()

OUTSTD() function

Write a list of values to the standard output device

Syntax

OUTSTD(<*exp list*>) → *NIL*

Arguments

<exp list> is a list of values to display and can consist of any combination of data types including memo.

Returns

OUTSTD() always returns NIL.

Description

OUTSTD() is a simple output function similar to QOUT() except that it writes to the STDOUT device (instead of to the CA-Clipper console output stream). Programs with very simple output requirements (i.e., that perform no full-screen input or output) can use this function to avoid loading the terminal output subsystems. The header file Simplio.ch redefines the ? and ?? commands to use the OUTSTD() function.

Since OUTSTD() sends its output to the standard output device, the output can be redirected using the DOS redirection symbols (>, >, |). This lets you redirect output from a CA-Clipper program to a file or pipe. Refer to your PC/MS-DOS documentation for more information about this operating system facility.

Examples

- This example uses OUTSTD() to display a list of expressions:

```
OUTSTD(Name, PADR(RTRIM(City) + "," + ;
       State, 20), ZipCode)
```

- This example redirects the output of a CA-Clipper program to a new file using the DOS redirection operator (>):

```
C>MYPROG > FILE.TXT
```

Files

Library is CLIPPER.LIB, header file is Simplio.ch.

See Also

DISPOUT(), OUTERR(), QOUT()

PACK command

Remove deleted records from a database file

Syntax

```
PACK
```

Description

PACK is a database command that removes all records marked for deletion from the current database file, REINDEXes all active indexes in the current work area, and recovers all the physical space occupied by the deleted records. During its operation, PACK does not create any backup files, although the associated REINDEX operation may. After the PACK command terminates, the record pointer is reset to the first logical record in the current work area.

In a network environment, PACK requires that the current database be USEd EXCLUSIVEly. If this condition is not met when PACK is invoked, CA-Clipper generates a runtime error.

Note that PACKing large database files can be a time-consuming process and may not be feasible in a high-volume transaction system on a network. By modifying the system design, you can remove the necessity of physically removing records from the database file altogether. See the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Examples

- The following example shows the result of a simple PACK:

```
USE Sales NEW
? LASTREC()                 // Result: 84
//
DELETE RECORD 4
PACK
? LASTREC()                 // Result: 83
```

Files

Library is CLIPPER.LIB.

See Also

DELETE, DELETED(), FLOCK(), RECALL, REINDEX, SET DELETED, ZAP

PAD() function

Pad character, date, and numeric values with a fill character

Syntax

```
PADL(<exp>, <nLength>, [<cFillChar>])
   → cPaddedString
PADC(<exp>, <nLength>, [<cFillChar>])
   → cPaddedString
PADR(<exp>, <nLength>, [<cFillChar>])
   → cPaddedString
```

Arguments

<exp> is a character, numeric, or date value to be padded with a fill character.

<nLength> is the length of the character string to be returned.

<cFillChar> is the character with which to pad *<exp>*. If not specified, the default is a space character.

Returns

PADC(), PADL(), and PADR() return the result of *<exp>* as a character string padded with *<cFillChar>* to a total length of *<nLength>*.

Description

PADC(), PADL(), and PADR() are character functions that pad character, date, and numeric values with a fill character to create a new character string of a specified length. PADC() centers *<exp>* within *<nLength>* adding fill characters to the left and right sides; PADL() adds fill characters on the left side; and PADR() adds fill characters on the right side. If the length of *<exp>* exceeds *<nLength>*, all of the PAD() functions truncate *cPaddedString* to *<nLength>*.

PADC(), PADL(), and PADR() display variable length strings within a fixed length area. They can be used, for instance, to ensure alignment with consecutive ?? commands. Another use is to display text to a fixed-width screen area assuring that previous text is completely overwritten.

PADC(), PADL(), and PADR() are the inverse of the ALLTRIM(), RTRIM(), and LTRIM() functions which trim leading and trailing space from character strings.

Examples

- This example uses PADR() to format a record number display on a status line filling the allocated space:

```
IF EOF()
   @ 23, 45 PADR("EOF/" + LTRIM(STR(LASTREC())), 20)
ELSEIF BOF()
   @ 23, 45 PADR("BOF/" + LTRIM(STR(LASTREC())), 20)
ELSE
   @ 23, 45 SAY PADR("Record " + LTRIM(STR(RECNO())) ;
         + "/" + LTRIM(STR(LASTREC())), 20)
ENDIF
```

Files Library is EXTEND.LIB.

See Also ALLTRIM(), LTRIM(), RTRIM()

PARAMETERS statement

Create private parameter variables

Syntax

```
PARAMETERS <idPrivate list>
```

Arguments

<idPrivate list> is one or more parameter variables separated by commas. The number of receiving variables does not have to match the number of arguments passed by the calling procedure or user-defined function.

Description

The PARAMETERS statement creates private variables to receive passed values or references. Receiving variables are referred to as *parameters*. The values or references actually passed by a procedure or user-defined function invocation are referred to as *arguments*.

When a PARAMETERS statement executes, all variables in the parameter list are created as private variables and all public or private variables with the same names are hidden until the current procedure or user-defined function terminates. A PARAMETERS statement is an executable statement and, therefore, can occur anywhere in a procedure or user-defined function, but must follow all compile-time variable declarations, such as FIELD, LOCAL, MEMVAR, and STATIC.

Parameters can also be declared as local variables if specified as a part of the PROCEDURE or FUNCTION declaration statement (see the example). Parameters specified in this way are referred to as formal parameters. Note that you cannot specify both formal parameters and a PARAMETERS statement with a procedure or user-defined function definition. Attempting to do this results in a fatal compiler error and an object file is not generated.

In CA-Clipper the number of arguments and parameters do not have to match. If you specify more arguments than parameters, the extra arguments are ignored. If you specify fewer arguments than parameters, the extra parameters are created with a NIL value. If you skip an argument, the corresponding parameter is initialized to NIL. The PCOUNT() function returns the position of the last argument passed in the list of arguments. This is different from the number of parameters passed since it includes skipped parameters.

For more information on passing parameters, refer to the Functions and Procedures section in the "Basic Concepts" chapter of the *Programming and Utilities Guide.*

Examples

- This user-defined function receives values passed into private parameters with a PARAMETERS statement:

```
FUNCTION MyFunc
   PARAMETERS cOne, cTwo, cThree
   ? cOne, cTwo, cThree
   RETURN NIL
```

- This example is similar, but receives values passed into local variables by declaring the parameter variables within the FUNCTION declaration:

```
FUNCTION MyFunc( cOne, cTwo, cThree )
   ? cOne, cTwo, cThree
   RETURN NIL
```

See Also

FUNCTION, LOCAL, PCOUNT(), PRIVATE, PROCEDURE, PUBLIC, STATIC

PCOL() function

Return the current column position of the printhead

Syntax

```
PCOL() → nColumn
```

Returns

PCOL() returns an integer numeric value representing the last printed column position, plus one. The beginning column position is zero.

Description

PCOL() is a printer function that reports the column position of the printhead after the last print operation. PCOL() is updated only if either SET DEVICE TO PRINTER or SET PRINTER ON is in effect. PCOL() is the same as COL() except that it relates to the printer rather than the screen. PCOL() is updated in the following ways:

- Application startup sets PCOL() to zero
- EJECT resets PCOL() to zero
- A print operation sets PCOL() to the last column print position plus one
- SETPRC() sets PCOL() to the specified column position

PCOL(), used with PROW(), prints a value relative to the last value printed on the same line. This makes it easier to align columns when printing a columnar report. A value is printed in the next column by specifying its position as PCOL() + <*column offset*>. Note that PCOL() is effective for alignment only if the column values are fixed-width. To guarantee fixed-width column values, format the output using TRANSFORM(), the PICTURE clause of @...SAY, or any of the PAD() functions.

Notes

- ***Printer control codes:*** Sending control codes to the printer causes PCOL() to become out of sync with the printhead position. Although control codes do not print, this discrepancy happens because CA-Clipper counts all characters sent to the printer regardless of how the printer treats them. To make the necessary adjustment, save the current PROW() and PCOL() values, send the control codes, and then use SETPRC() to restore the original PROW() and PCOL() values.

- ***SET MARGIN:*** PCOL() cannot reliably be used with SET MARGIN to print with @...SAY. When printing with @...SAY, the current MARGIN value is always added to the specified column position before output is sent to the printer. This effectively adds the MARGIN value to PCOL() for each invocation of @...SAY to the same print line.

Examples

- In this example, PCOL() creates a simple report that prints a listing of Customer names, addresses, and phone numbers:

```
LOCAL nLine := 99, nPage := 1
USE Customer INDEX CustName NEW
SET DEVICE TO PRINTER
DO WHILE !EOF()
   IF nLine > 55
      PageTop(nPage)
      nLine := 1
      nPage++
   ENDIF
   @ nLine, 10 SAY CustName
   @ nLine, PCOL() + 2;
      SAY RTRIM(City) + ", " + RTRIM(State) + ZipCode;
            PICTURE REPLICATE("X", 35)
   @ nLine, PCOL() + 2;
      SAY Phone;
            PICTURE "@R (999) 999-9999"
   nLine++
   SKIP
ENDDO
SET DEVICE TO SCREEN
CLOSE
```

Files

Library is CLIPPER.LIB.

See Also

?|??, @...SAY, COL(), EJECT, PAD(), PROW(), QOUT(), ROW(), SET DEVICE, SET PRINTER, SETPRC(), TRANSFORM()

PCOUNT() function

Determine the position of the last actual parameter passed

Syntax

```
PCOUNT() → nLastArgumentPos
```

Returns

PCOUNT() returns, as an integer numeric value, the position of the last argument passed. If no arguments are passed, PCOUNT() returns zero.

Description

PCOUNT() reports the position of the last argument in the list of arguments passed when a procedure or user-defined function is invoked. This information is useful when determining whether arguments were left off the end of the argument list. Arguments skipped in the middle of the list are still included in the value returned.

To determine if a parameter did not receive a value, test it for NIL. Skipped parameters are uninitialized and, therefore, return NIL when accessed. Another method is to test parameters with the VALTYPE() function. This can establish whether the argument was passed and enforce the correct type at the same time. If a parameter was not supplied, a default value can be assigned.

For more information on passing parameters, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide.*

Examples

- This example is a user-defined function that opens a database file and uses PCOUNT() to determine whether the calling procedure passed the name of the database file to be opened. If the name was not passed, OpenFile() asks for the name:

```
FUNCTION OpenFile( cFile )
    IF PCOUNT() = 0
       ACCEPT "File to use: " TO cFile
    ENDIF
    USE (cFile)
    RETURN (NETERR())
```

Files Library is CLIPPER.LIB.

See Also DO*, FUNCTION, PARAMETERS, PROCEDURE, VALTYPE()

PopUpMenu class

Create a pop-up menu

Description

Place items on the top bar menu or another pop-up menu.

Class Function

```
PopUp([<nTop>], [<nLeft>], [<nBottom>], [<nRight>])
    → oPopUp
```

Arguments

<nTop> is a numeric value that indicates the top screen row of the pop-up menu. If omitted, PopUpMenu:top is set to an appropriate value relative to *<nBottom>* that allows as many items as possible to show. If *<nBottom>* is also omitted, PopUpMenu:top is set to center the menu vertically on the screen. The default value is determined the first time the pop-up menu is displayed.

When the pop-up menu is a child of another menu, its top variable will be automatically set by the parent menu regardless of whether *<nTop>* is omitted.

<nLeft> is a numeric value that indicates the left screen column of the pop-up menu. If omitted, PopUpMenu:left is set to an appropriate value relative to *<nRight>* that allows as many menu columns as possible to show. If *<nRight>* is also omitted, PopUpMenu:left is set to center the menu horizontally on the screen. The default value is determined the first time the pop-up menu is displayed.

When the pop-up menu is a child of another menu, its left variable will be automatically set by the parent menu regardless of whether *<nLeft>* is omitted.

<nBottom> is a numeric value that indicates the bottom screen row of the pop-up menu. If omitted, PopUpMenu:bottom is set to an appropriate value relative to *<nTop>* that allows as many items as possible to show. If *<nTop>* is also omitted, PopUpMenu:bottom is set to center the menu vertically on the screen. The default value is determined the first time the pop-up menu is displayed.

When the pop-up menu is a child of another menu, its bottom variable will be automatically set by the parent menu regardless of whether *<nBottom>* is omitted.

<nRight> is a numeric value that indicates the right screen column of the pop-up menu. If omitted, PopUpMenu:right is set to an appropriate value relative to *<nLeft>* that allows as many menu columns as possible to show. If *<nLeft>* is also omitted, PopUpMenu:right is set to center the menu horizontally on the screen. The default value is determined the first time the pop-up menu is displayed.

When the pop-up menu is a child of another menu, its right variable will be automatically set by the parent menu regardless of whether *<nRight>* is omitted.

Returns

Returns a PopUpMenu object when all of the required arguments are present; otherwise, PopUp() returns NIL.

Exported Instance Variables

`border` `(Assignable)`

Contains an optional string that is used when drawing a border around the pop-up menu. Its default value is B_SINGLE + SEPARATOR_SINGLE. The string must contain either zero or exactly eleven characters. The first eight characters represent the border of the pop-up menu and the final three characters represent the left, middle, and right characters for the menu item separators. The eight characters which represent the pop-up menu border begin at the upper-left corner and rotate clockwise as follows: upper-left corner, top, upper-right corner, right, bottom, bottom-left corner, and left.

`bottom` `(Assignable)`

Contains a numeric value that indicates the bottommost screen row where the pop-up menu is displayed. If not specified when the PopUpMenu object is instantiated, PopUpMenu:bottom contains NIL until the first time it is displayed.

`cargo` `(Assignable)`

Contains a value of any type that is ignored by the PopUpMenu object. PopUpMenu:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a PopUpMenu object and retrieved later.

`colorSpec` `(Assignable)`

Contains a character string that indicates the color attributes that are used by the pop-up menu's display() method. The string must contain six color specifiers.

PopUpMenu Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The pop-up menu items that are not selected	Unselected
2	The selected pop-up menu item	Enhanced
3	The accelerator key for unselected pop-up menu items	Background
4	The accelerator key for the selected pop-up menu item	Enhanced
5	Disabled pop-up menu items	Standard
6	The pop-up menu's border	Border

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Workbench Properties window for that item.

`current`

Contains a numeric value that indicates which item is selected. PopUpMenu:current contains 0 when the pop-up menu is not open.

`itemCount`

Contains a numeric value that indicates the total number of items in the PopUpMenu object.

`left` `(Assignable)`

Contains a numeric value that indicates the leftmost screen row where the pop-up menu is displayed. If not specified when the PopUpMenu object is instantiated, PopUpMenu:left contains NIL until the first time it is displayed.

`right` `(Assignable)`

Contains a numeric value that indicates the rightmost screen row where the pop-up menu is displayed. If not specified when the PopUpMenu object is instantiated, PopUpMenu:right contains NIL until the first time it is displayed.

`top` (Assignable)

Contains a numeric value that indicates the topmost screen row where the pop-up menu is displayed. If not specified when the PopUpMenu object is instantiated, PopUpMenu:top contains NIL until the first time it is displayed.

`width`

Contains a numeric value that indicates the width required to display all of the pop-up menu's items in their entirety. This includes check marks and submenu indicators.

Exported Methods

`<oPopUp>:addItem(<oMenuItem>)` → *self*

<oMenuItem> is a MenuItem object.

addItem() is a method of the PopUpMenu class that is used for appending a new item to a pop-up menu.

`<oPopUp>:close([<lCloseChild>])` → *self*

<lCloseChild> is a logical value that indicates whether PopUpMenu:close() should deactivate the pop-up menu in its selected item, which in turn deactivates the pop-up menu in its selected item and so on. This is useful for nested menus where multiple levels of choices are presented. A value of true (.T.) indicates that child pop-up menu items should be closed. A value of false (.F.) indicates that child pop-up menu items should not be closed. The default value is true.

close() is a method of the PopUpMenu class that is used for deactivating a pop-up menu. When called, PopUpMenu:close() performs three operations. First, if the value of *<lCloseChild>* is not false (.F.) , close() determines if its selected menu item contains a PopUpMenu object. If so, it calls its selected menu item's close() method. Second, close() restores the previous contents of the region of the screen that it occupies. Third, close() sets its selected item to 0.

Note: This message only has meaning when the pop-up menu is open.

`<oPopUp>:delItem(<nPosition>) → self`

<nPosition> is a numeric value that indicates the position in the pop-up menu of the item to be deleted.

delItem() is a method of the PopUpMenu class that is used for removing an item from a pop-up menu.

`<oPopUp>:display() → self`

display() is a method of the PopUpMenu class that is used for showing a pop-up menu including its items on the screen. display() uses the values of the following instance variables to correctly show the list in its current context, in addition to providing maximum flexibility in the manner a pop-up menu appears on the screen:

- MenuItem:checked
- MenuItem:enabled
- MenuItem:isPopUp
- PopUpMenu:bottom
- PopUpMenu:colorSpec
- PopUpMenu:current
- PopUpMenu:itemCount
- PopUpMenu:left
- PopUpMenu:right
- PopUpMenu:top
- PopUpMenu:width

`<oPopUp>:getAccel(<nInkeyValue>) → nPosition`

<nInkeyValue> is a numeric value that indicates the inkey value to be checked.

Returns a numeric value that indicates the position in the pop-up menu of the first item whose accelerator key matches that which is specified by *<nInkeyValue>*. The accelerator key is defined using the & character in MenuItem:caption.

getAccel() is a method of the PopUpMenu class that is used for determining whether a key press should be interpreted as a user request to evoke the Data variable of a particular pop-up menu item.

`<oPopUp>:getFirst() → nPosition`

getFirst() is a method of the PopUpMenu class that is used for determining the position of the first selectable item in a pop-up menu. The term selectable is defined as a menu item that is enabled and whose caption is not a menu separator.

Returns a numeric value that indicates the position within the pop-up menu of the first selectable item. getFirst() returns 0 in the event that the pop-up menu does not contain a selectable item.

Note: getFirst() does not change the currently selected menu item. In order to change the currently selected pop-up menu item, you must call the PopUpMenu:select() method.

`<oPopUp>:getItem(<nPosition>) → oMenuItem`

<nPosition> is a numeric value that indicates the position in the pop-up menu of the item that is being retrieved.

Returns the MenuItem object at the position in the pop-up menu specified by *<nPosition>* or NIL when *<nPosition>* is invalid.

getItem() is a method of the PopUpMenu class that is used for accessing a MenuItem object after it has been added to a pop-up menu.

`<oPopUp>:getLast() → nPosition`

Returns a numeric value that indicates the position within the pop-up menu of the last selectable item. getLast() returns 0 in the event that the pop-up menu does not contain a selectable item.

getLast() is a method of the PopUpMenu class that is used for determining the position of the last selectable item in a pop-up menu. The term selectable is defined as a menu item that is enabled and whose caption is not a menu separator.

Note: getLast() does not change the currently selected menu item. In order to change the currently selected pop-up menu item, you must call the PopUpMenu:select() method.

`<oPopUp>:getNext() → nPosition`

Returns a numeric value that indicates the position within the pop-up menu of the next selectable item. getNext() returns 0 in the event that the current item is the last selectable item or the pop-up menu does not contain a selectable item.

getNext() is a method of the PopUpMenu class that is used for determining the position of the next selectable item in a pop-up menu. getNext() searches for the next selectable item starting at the item immediately after the current item. The term selectable is defined as a menu item that is enabled and whose caption, is not a menu separator.

Note: getNext() does not change the currently selected menu item. In order to change the currently selected pop-up menu item, you must call the PopUpMenu:select() method.

```
<oPopUp>:getPrev() → nPosition
```

Returns a numeric value that indicates the position within the pop-up menu of the previous selectable item. getPrev() returns 0 in the event that the current item is the first selectable item or the pop-up menu does not contain a selectable item.

getPrev() is a method of the PopUpMenu class that is used for determining the position of the previous selectable item in a pop-up menu. getPrev() searches for the previous selectable item starting at the item immediately before the current item. The term selectable is defined as a menu item that is enabled and whose caption is not a menu separator.

Note: getPrev() does not change the currently selected menu item. In order to change the currently selected pop-up menu item, you must call the PopUpMenu:select() method.

```
<oPopUp>:getShortct(<nInkeyValue>) → nPosition
```

<nInkeyValue> is a numeric value that indicates the inkey value to be checked.

Returns a numeric value that indicates the position in the pop-up menu of the first item whose shortcut key matches that which is specified by *<nInkeyValue>*. The shortcut key is defined using the MenuItem:shortcut instance variable.

getShortct() is a method of the PopUpMenu class that is used for determining whether a keystroke should be interpreted as a user request to select a particular pop-up menu item.

```
<oPopUp>:hitTest(<nMouseRow>, <nMouseCol>)
    → nHitStatus
```

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<*nMouseCol*> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the pop-up menu.

Applicable Hit Test Return Values

Value	Constant	Description
> 0	Not Applicable	The position in the pop-up menu of the item whose region the mouse is within
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the menu occupies
-1	HTTOPLEFT	The mouse cursor is on the top-left corner of the pop-up menu's border
-2	HTTOP	The mouse cursor is on the pop-up menu's top border
-3	HTTOPRIGHT	The mouse cursor is on the top-right corner of the pop-up menu's border
-4	HTRIGHT	The mouse cursor is on the pop-up menu's right border
-5	HTBOTTOMRIGHT	The mouse cursor is on the bottom-right corner of the pop-up menu's border
-6	HTBOTTOM	The mouse cursor is on the pop-up menu's bottom border
-7	HTBOTTOMLEFT	The mouse cursor is on the bottom-left corner of the pop-up menu's border
-8	HTLEFT	The mouse cursor is on the pop-up menu's left border
-4098	HTSEPARATOR	The mouse is on a menu separator line

Button.ch contains manifest constants for the PopUpMenu:hitTest() return value.

hitTest() is a method of the PopUpMenu class that is used for determining if the mouse cursor is within the region of the screen that the pop-up menu occupies.

```
<oPopUp>:insItem(<nPosition>, <oMenuItem>) → self
```

<nPosition> is a numeric value that indicates the position at which the new menu item is inserted.

<oMenuItem> is a MenuItem object.

insItem() is a method of the PopUpMenu class that is used for inserting a new item within a pop-up menu.

```
<oPopUp>:isOpen() → lIsOpen
```

Returns a logical value that indicates whether the pop-up menu is open or not. A value of true (.T.) indicates that the pop-up menu is open, a value of false (.F.) indicates that it is closed.

isOpen() is a method of the PopUpMenu class that is used for determining if a pop-up menu is open. A pop-up menu is considered open during the period after calling its open() method and before calling its close() method

```
<oPopUp>:open() → self
```

open() is a method of the PopUpMenu class that is used for activating a pop-up menu. When called, open() performs two operations. First, open() saves the previous contents of the region of the screen that the pop-up menu occupies. Second, open() calls its pop-up menu's display() method.

Note: This message only has meaning when the pop-up menu is closed.

```
<oPopUp>:select(<nPosition>) → self
```

<nPosition> is a numeric value that indicates the position in the pop-up menu of the item to be selected.

select() is a method of the PopUpMenu class that is used for changing the selected item. Its state is typically changed when one of the arrow keys is pressed or the mouse's left button is pressed when its cursor is within the pop-up menu's screen region.

```
<oPopUp>:setItem(<nPosition>, <oMenuItem>) → self
```

<nPosition> is a numeric value that indicates the position in the pop-up menu of the item that is being retrieved.

<oMenuItem> is the MenuItem object that replaces the one in the pop-up menu specified by *<nPosition>*.

setItem() is a method of the PopUpMenu class that is used for replacing a MenuItem object after it has been added to a pop-up menu. After the setItem() method is called, the display()method needs to be called in order to refresh the menu. It is not allowed to call the display() method for the currently opened pop-up menu in the code block of the menu items. Otherwise, there are screen refresh problems especially if the code block has also increased the number of items in the pop-up menu.

Examples

See the Menu.prg sample file in the \CLIP53\SOURCE\SAMPLE directory. This example demonstrates combining TopBarMenu, PopUpMenu, and MenuItem objects to create a menu with a number of available choices. See "Introduction to the Menu System" in the *Programming and Utilities Guide* for more information about using this class.

See Also

MenuItem class, MENUMODAL(), TopBarMenu class

PRIVATE statement

Create and initialize private memory variables and arrays

Syntax

```
PRIVATE <identifier> [[:= <initializer>], ... ]
```

Arguments

<identifier> is the name of a private variable or array to create. If the *<identifier>* is followed by square brackets ([ ]), an array is created and assigned to the *<identifier>*. When the *<identifier>* specification indicates an array, the syntax for specifying the number of elements for each dimension can be array[*<nElements>*, *<nElements2>*,...] or array[*<nElements>*][*<nElements2>*]... The maximum number of elements per dimension is 4096. The maximum number of dimensions is limited only by available memory.

<initializer> is the optional assignment of a value to a new private variable. An array cannot be given values with an *<initializer>*. An *<initializer>* for a private variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression including a literal array. If no explicit *<initializer>* is specified, the variable is initialized to NIL. In the case of an array, each element is initialized to NIL.

You can create and, optionally, initialize a list of variables and arrays with one PRIVATE statement if the definitions are separated by commas.

Description

The PRIVATE statement creates variables and arrays visible within the current and invoked procedures or user-defined functions. This class of variable is said to have dynamic scope. Private variables exist for the duration of the active procedure or until explicitly released with CLEAR ALL, CLEAR MEMORY, or RELEASE. When a private variable or array is created, existing and visible private and public variables of the same name are hidden until the current procedure or user-defined function terminates.

Attempting to specify a PRIVATE variable that conflicts with a previous FIELD, LOCAL, or STATIC declaration of the same name results in a fatal compiler error. This is true regardless of the scope of the declaration.

PRIVATE statements are executable statements and, therefore, must be specified within the body of a procedure or user-defined function and must follow all variable declarations, such as FIELD, LOCAL, MEMVAR, and STATIC.

In addition to the PRIVATE statement, private variables are also created in two other ways:

- Assignment to a variable that does not exist or is not visible will create a private variable
- Parameters received using the PARAMETERS statement are created as private variables with the same lifetime and visibility

No more than 2048 private and public variables and arrays can simultaneously exist in a single program.

For more information on variable declarations and scoping, refer to the Variables section in the "Basic Concepts" chapter of the *Programming and Utilities Guide*.

Notes

- ***Compatibility:*** The ALL, LIKE, and EXCEPT clauses of the PRIVATE statement supported by other dBASE dialects are not supported by CA-Clipper.

Examples

- This example creates two PRIVATE arrays and three other PRIVATE variables:

```
PRIVATE aArray1[10], aArray2[20], var1, var2, var3
```

- This example creates a multidimensional private array using each element addressing convention:

```
PRIVATE aArray[10][10][10], aArray2[10, 10, 10]
```

- This example uses PRIVATE statements to create and initialize arrays and variables:

```
PRIVATE aArray := { 1, 2, 3, 4 }, ;
      aArray2 := ARRAY(12, 24)
PRIVATE cChar := SPACE(10), cColor := SETCOLOR()
```

See Also

FIELD, LOCAL, MEMVAR, PARAMETERS, PUBLIC, STATIC

PROCEDURE statement

Declare a procedure name and formal parameters

Syntax

```
[STATIC] PROCEDURE <idProcedure> [(<idParam list>)]
   [FIELD <idField list> [IN <idAlias>]
   [LOCAL <identifier> [[:= <initializer>], ... ]]
   [MEMVAR <identifier list>]
   [STATIC <identifier> [[:= <initializer>], ... ]]
   .
   . <executable statements>
   .
   [RETURN]
```

Arguments

<idProcedure> is the name of the procedure to be declared. Procedure names can be any length, but only the first 10 characters are significant. Names can contain any combination of characters, numbers, or underscores, but leading underscores are reserved.

<idParam list> is the declaration of one or more parameter variables. Variables specified in this list are declared local.

STATIC PROCEDURE declares a procedure that can be called only by procedures and user-defined functions declared in the same program (.prg) file.

FIELD declares a list of identifiers, *<idField list>*, to use as field names whenever encountered. If the IN clause is specified, referring to the declared name, *<idAlias>* is a reference to the appropriate work area of the specified database.

LOCAL declares and optionally initializes a list of variables or arrays whose visibility and lifetime is the current procedure.

<identifier>, <identifier list> is a label or labels used as variable or array names. If the *<identifier>* is followed by square brackets ([ ]), it is created as an array. If the *<identifier>* is an array, the syntax for specifying the number of elements for each dimension can be array[*<nElements>*, *<nElements2>*,...] or array[*<nElements>*][*<nElements2>*]... The maximum number of elements per dimension is 4096. The maximum number of dimensions per array is limited only by available memory.

<initializer> is the value to which an optional inline assignment sets the *<identifier>* variable—essentially, the assignment operator, (:=) — followed by any valid CA-Clipper expression, including a literal array. If no *<initializer>* is specified, variables are initialized to NIL. In the case of arrays, all element are initialized to NIL.

MEMVAR declares a list of identifiers, *<identifier list>*, to use as private or public memory variables or arrays whenever encountered.

STATIC declares and, optionally, initializes a list of variables or arrays whose visibility is the current procedure and whose lifetime is the duration of the program.

RETURN passes control back to the calling procedure or user-defined function. If a RETURN is not specified, control passes back to the calling routine when the procedure definitions ends. In all cases, the compiler terminates the procedure definition when it encounters another PROCEDURE statement, FUNCTION statement, or end of file character.

Description

The PROCEDURE statement declares a procedure and an optional list of local variables to receive parameters passed from a calling routine. A procedure is a subprogram comprised of a set of declarations and statements executed whenever you refer to *<idProcedure>*, followed by an open and close parentheses pair or with the DO statement. A procedure definition begins with a PROCEDURE statement and ends with the next PROCEDURE statement, FUNCTION statement, or end of file.

Procedures that encapsulate computational blocks of code provide readability and modularity, isolate change, and help manage complexity.

A procedure in CA-Clipper is the same as a user-defined function, with the exception that it always returns NIL. Each procedure must begin with a PROCEDURE statement and may, optionally, contain a RETURN statement to return control to the calling procedure or user-defined function. A RETURN statement, however, is not required. Procedure declarations cannot be nested within other procedure definitions.

The visibility of procedure names falls into two classes. Procedures that are visible anywhere in a program are referred to as public procedures and declared with a PROCEDURE statement. Procedures that are visible only within the current program (.prg) file are referred to as static procedures and declared with a STATIC PROCEDURE statement. Static procedures have filewide scope.

Static procedures are quite useful for a number of reasons. First, they limit visibility of a procedure name thereby restricting access to the procedure. Because of this, subsystems defined within a single program (.prg) file can provide an access protocol with a series of public procedures and conceal the implementation details of the subsystem within static procedures and functions. Second, since the static procedure references are resolved at compile time, they preempt references to public procedures and functions which are resolved at link time. This ensures that, within a program file, a reference to a static procedure executes that procedure if there is a name conflict with a public procedure or function.

For more information on procedures, variable declarations, and parameter passing, refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide.*

Notes

- ***Calling a procedure:*** There are two ways to call a procedure in CA-Clipper. The first and preferred way is the function-calling convention. Here you call the procedure as you would a CA-Clipper function on a line by itself:

  ```
  <idProcedure>([<argument list>])
  ```

  The second and obsolete way is the command-calling convention using the DO...WITH command. The two methods of calling procedures differ only in the default method of passing parameters. The function-calling convention passes variables by value as a default, whereas the command-calling convention passes them by reference as a default.

  A procedure can also be called as an aliased expression if it is prefaced with an alias and invoked using the function-calling convention, like this:

  ```
  <idAlias> ->(<idProcedure>(<argument list>))
  ```

  When called as an aliased expression, the work area associated with *<idAlias>* is selected, the procedure is executed, and then the original work area is reselected. Like an expression or function, an aliased procedure can be specified on a line by itself.

  A procedure in CA-Clipper may call itself recursively. This means you can call a procedure in the same procedure definition.

- ***Parameters:*** Procedures like user-defined functions can receive parameters passed from a calling procedure, user-defined function, or the DOS command line. A parameter is a place for a value or reference. In CA-Clipper there are two ways to receive parameters: a list of local variable names can be declared as a part of the PROCEDURE declaration (referred to as formal parameters), or a list of private variables can be specified in a separate PARAMETERS statement. Note that you cannot mix a declaration of formal parameters with a PARAMETERS statement. Attempting this will cause a fatal compiler error.

  Procedures receive parameters in the order passed. In CA-Clipper the number of parameters need not match the number of arguments passed. Arguments can be skipped or left off the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. If arguments are specified, PCOUNT() returns the position of the last argument passed.

  Parameters specified in a procedure can receive arguments passed by value or by reference. The default method for expressions and variables depends on the calling convention. With the function-calling convention, the default passing method for expressions and variables is by value. This includes variables containing references to arrays and objects. With the command-calling convention, the default method for passing variables is by reference except for field variables, which are always passed by value. Whenever a field variable is passed, it must be specified enclosed in parentheses unless declared with the FIELD statement. Failure to do so will generate a runtime error.

Examples

- This example shows a skeleton of a typical CA-Clipper procedure that uses lexical variables:

```
PROCEDURE Skeleton( cName, cClassRoom, nBones, ;
                    nJoints )
   LOCAL nCrossBones, aOnHand := {"skull", ;
                                  "metacarpals"}
   STATIC nCounter := 0
   .
   . <executable statements>
   .
   RETURN
```

- This example determines whether an argument was skipped by comparing the parameter to NIL:

```
PROCEDURE MyProc( param1, param2, param3 )
   IF param2 != NIL
      param2 := "default value"
   ENDIF
   .
   . <statements>
   .
   RETURN
```

- This example invokes the procedure, UpdateAmount(), as an aliased expression:

```
USE Invoices NEW
USE Customer NEW
Invoices->(UpdateAmount(Amount + Amount * nInterest))
```

See Also

FUNCTION, LOCAL, PARAMETERS, PCOUNT(), PRIVATE, PUBLIC, RETURN, STATIC

PROCLINE() function

Return the source line number of the current or previous activation

Syntax

```
PROCLINE([<nActivation>]) → nSourceLine
```

Arguments

<nActivation> is a numeric value that specifies which activation to query. Zero refers to the current activation, one refers to the previous activation, etc. If not specified, the default value is zero.

Returns

PROCLINE() returns the line number of the last line executed in a currently executing procedure, function, or code block as an integer numeric value. If the /L compiler option suppresses line number information, PROCLINE() always returns zero.

Description

PROCLINE() queries the CA-Clipper activation stack to determine the last line executed in a currently executing procedure, user-defined function, or code block. The activation stack is an internal structure that maintains a record of each procedure, function, or code block invocation. A line number is relative to the beginning of the original source file. A line includes a comment, blank line, preprocessor directive, and a continued line. A multistatement line is counted as a single line.

For the current activation, PROCLINE() returns the number of the current line. For a previous activation, PROCLINE() returns the number of the line that invoked the procedure or a user-defined function in which PROCLINE() is invoked.

If the activation being queried is a code block evaluation, PROCLINE() returns the line number of the procedure in which the code block was originally defined.

PROCLINE() is used with PROCNAME() to report debugging information.

Examples

- In this example, PROCLINE() returns the line number for the current activation, followed by the line number of the previous activation:

```
// First line of source file
MyFunction()
   RETURN

FUNCTION MyFunction
   ? PROCLINE()      // Result: 6 (current activation)
   ? PROCLINE(1)     // Result: 2 (previous activation)
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also PROCNAME()

PROCNAME() function

Return the name of the current or previous procedure or user-defined function

Syntax

```
PROCNAME([<nActivation>]) → cProcedureName
```

Arguments

<nActivation> specifies which activation to query. A value of zero refers to the current activation, a value of one refers to the previous activation, etc. If unspecified, the default value is zero.

Returns

PROCNAME() returns the name of a currently executing procedure, function, or code block, as a character string.

Description

PROCNAME() queries the CA-Clipper activation stack to determine the name of a currently executing procedure, user-defined function, or code block. The activation stack is an internal structure that maintains a record of each procedure, function, or code block invocation.

For the current activation, PROCNAME() returns the name of the current procedure or user-defined function. For a previous activation, PROCNAME() returns the name of the procedure or user-defined function that invoked the current procedure.

If the activation being queried is a code block evaluation, PROCNAME() returns the name of the procedure or user-defined function that defined the code block, preceded by "b". If the activation being queried is a MEMVAR, PROCNAME() returns the name preceded by "*M*->".

PROCNAME() is used with PROCLINE() to report debugging information.

Examples

- This example is a user-defined function you can call during a debugging phase of program development in order to display the activation stack with line numbers:

```
FUNCTION ListStack( cMessage )
   LOCAL nActivation := 1
   ? cMessage
   DO WHILE !(PROCNAME(nActivation) == "")
      ? "Called from:", PROCNAME(nActivation),;
         "(" + LTRIM(STR(PROCLINE(nActivation))) + ")"
      nActivation++
   ENDDO
   QUIT
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also PROCLINE()

PROW() function

Return the current row position of the printhead

Syntax

```
PROW() → nRow
```

Returns

PROW() returns an integer numeric value that represents the number of the current line sent to the printer. The beginning row position is zero.

Description

PROW() is a printer function that reports the row position of the printhead after the last print operation. PROW() is updated only if either SET DEVICE TO PRINTER or SET PRINTER ON is in effect. PROW() is like ROW() except that it relates to the printer rather than the screen. PROW() is updated in the following ways:

- Application startup sets PROW() to zero
- EJECT resets PROW() to zero
- A print operation sets PROW() to the last row print position
- SETPRC() sets PROW() to the specified row position

PROW() used with PCOL() prints a value to a new row relative to the last row printed. If the printhead is positioned to a new row with a control code, a line feed (CHR(10)), or form feed (CHR(12)), PROW() is not updated and, therefore, will not return the expected value. To prevent this discrepancy, reset PROW() to the correct value with SETPRC() after sending any of these characters to the printer.

Examples

- This example uses PROW() and SETPRC() to create a simple one-across label program that prints with @...SAY instead of ?:

```
USE Customer INDEX CustName NEW
SET DEVICE TO PRINTER
SETPRC(2, 0)
DO WHILE !EOF()
    @ PROW(), 3 SAY CustName
    @ PROW() + 1, 3 SAY RTRIM(City) + ",;
          " + RTRIM(State) + ZipCode
    @ PROW() + 1, 3 SAY Phone PICTURE "@R ;
         (999) 999-9999"
    SETPRC(2, 0)
    SKIP
ENDDO
SET DEVICE TO SCREEN
CLOSE
```

Files Library is CLIPPER.LIB.

See Also ? | ??, @...SAY, COL(), EJECT, PCOL(), QOUT(), ROW(), SET DEVICE, SET PRINTER, SETPRC()

PUBLIC statement

Create and initialize public memory variables and arrays

Syntax

```
PUBLIC <identifier> [[:= <initializer>], ... ]
```

Arguments

<identifier> is the name of a public variable or array to create. If the <*identifier*> is followed by square brackets ([ ]), it is created as an array. If the <*identifier*> is an array, the syntax for specifying the number of elements for each dimension can be array[<*nElements*>, <*nElements2*>,...] or array[<*nElements*>][<*nElements2*>].... The maximum number of elements per dimension is 4096. The maximum number of dimensions per array is limited only by available memory.

<initializer> is the optional assignment of a value to a new public variable. Array identifiers, however, cannot be given values with an <*initializer*>. An <*initializer*> for a public variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression including a literal array. Except for arrays, if no <*initializer*> is specified, public variables are initialized to false (.F.). This is an exception to the general rule that uninitialized variables are NIL. With arrays, however, the initial value of each element is NIL.

A list of variables and arrays can be created and, optionally, initialized with one PUBLIC statement if each definition is separated by a comma.

Description

The PUBLIC statement creates variables and arrays visible to all procedures and user-defined functions in a program. Public variables exist for the duration of the program or until explicitly released with CLEAR ALL, CLEAR MEMORY, or RELEASE. Declaring private, local, or static variables or arrays with the same name as existing public variables temporarily hides those public variables until the overriding variables are released or are no longer visible. An attempt to create a public variable with the same name as an existing and visible private variable is simply ignored (see Notes below for an exception).

Attempting to specify a PUBLIC variable that conflicts with a previous FIELD, LOCAL, or STATIC declaration of the same name results in a fatal compiler error. This is true regardless of the scope of the declaration.

PUBLIC statements are executable statements and, therefore, must be specified within the body of a procedure or user-defined function definition. They also must follow all compile-time declarations, such as FIELD, LOCAL, MEMVAR, and STATIC.

The maximum number of public and private variables and arrays that can simultaneously exist in a single program is 2048.

For more information on variable declarations and scoping, refer to the Variables section in the "Basic Concepts" chapter of the *Programming and Utilities Guide*.

Notes

- ***PUBLIC Clipper:*** To include CA-Clipper extensions in a program and still allow the program to run under dBASE III PLUS, the special public variable, Clipper, is initialized to true (.T.) when created PUBLIC.
- ***Public array name conflicts with existing private variables:*** The statement, PUBLIC x[10], will not create the public array x if there is already a private or public variable x. It will, however, destroy the contents of the existing x, replacing it with a reference to a ten-element array.

Examples

- This example creates two PUBLIC arrays and one PUBLIC variable:

```
PUBLIC aArray1[10, 10], var2
PUBLIC aArray2[20][10]
```

- The following PUBLIC statements create variables and initialize them with values:

```
PUBLIC cString := SPACE(10), cColor := SETCOLOR()
PUBLIC aArray := {1, 2, 3}, aArray2 := ARRAY(12, 24)
```

See Also

LOCAL, MEMVAR, PARAMETERS, PRIVATE, STATIC

PushButton class

Create a push button

Description

Places a push button at the indicated position on the screen.

Class Function

```
PushButton( <nRow>, <nColumn>, [<cCaption>] )
    → oPushButton
```

Arguments

<nRow> is a numeric value that indicates the screen row of the push button.

<nColumn> is a numeric value that indicates the screen column of the push button.

<cCaption> is an optional character string that describes the push button on the screen. If omitted, the default is an empty string.

Returns

Returns a PushButton object when all of the required arguments are present; otherwise PushButton() returns NIL.

Exported Instance Variables

`bitmap` `(Assignable)`

Contains a character string that indicates a bitmap file to be displayed on the button. Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and, if it is not found, search in the bitmap library second. If no file is found either on disk or in the library, no bitmap will be displayed. This instance variable only affects applications running in graphic mode and is ignored in text mode.

`bmpXOff` (Assignable)

Contains a numeric value that indicates the offset where the bitmap is displayed. This instance variable represents the number of pixels in the x direction (horizontally) from the left edge of the button where the bitmap will be displayed. If this instance variable is not supplied, the bitmap will be placed at the left edge of the button. This instance variable only affects applications running in graphic mode and is ignored in text mode.

`bmpYOff` (Assignable)

Contains a numeric value that indicates the offset where the bitmap is displayed. This instance variable represents the number of pixels in the y direction (vertically) from the top edge of the button where the bitmap will be displayed. If this instance variable is not supplied, the bitmap will be placed at the top edge of the button. This instance variable only affects applications running in graphic mode and is ignored in text mode.

`buffer`

Contains a logical value indicating that the push button has been pushed. A value of true (.T.) indicates that the push button has been pushed; otherwise, a value of false (.F.) indicates that it has not.

`caption` (Assignable)

Contains a character string that describes the push button on the screen.

When present, the & character specifies that the character immediately following it in the caption is the push button's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to a push button. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

`capXOff` (Assignable)

Contains a numeric value that indicates the offset where the caption is displayed. This instance variable represents the number of pixels in the x direction (horizontally) from the left edge of the button where the caption will be displayed. If this instance variable is not supplied, the caption will be centered horizontally. This instance variable only affects applications running in graphic mode and is ignored in text mode.

`capYOff` (Assignable)

Contains a numeric value that indicates the offset where the caption is displayed. This instance variable represents the number of pixels in the y direction (vertically) from the top edge of the button where the caption will be displayed. If this instance variable is not supplied, the caption will be centered vertically. This instance variable only affects applications running in graphic mode and is ignored in text mode.

`caption` (Assignable)

Contains a character string that describes the push button on the screen.

When present, the & character specifies that the character immediately following it in the caption is the push button's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to a push button. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

`cargo` (Assignable)

Contains a value of any type that is not used by the PushButton object. PushButton:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a PushButton object and retrieved later.

`col` (Assignable)

Contains a numeric value that indicates the screen column where the push button is displayed.

`colorSpec` (Assignable)

Contains a character string that indicates the color attributes that are used by the push button's display() method. The string must contain four color specifiers.

Note: The background colors of the PushButton Color Attributes are ignored in graphic mode.

PushButton Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The push button when it does not have input focus.	Unselected
2	The push button when it has input focus and is not pressed.	Enhanced
3	The push button when it has input focus and is pressed.	Standard
4	The push button caption's accelerator key.	Background

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Workbench Properties window for that item.

`fBlock` `(Assignable)`

Contains an optional code block that, when present, is evaluated each time the PushButton object receives or loses input focus. The code block takes no implicit arguments. Use the PushButton:hasFocus variable to determine if the push button is receiving or losing input focus. A value of true (.T.) indicates that it is receiving input focus; otherwise, a value of false (.F.) indicates that it is losing input focus.

This code block is included in the PushButton class to provide a method of indicating when an input focus change event has occurred. The name "fBlock" refers to focus block.

`hasFocus`

Contains a logical value that indicates whether the PushButton object has input focus. PushButton:hasFocus contains true (.T.) if it has input focus; otherwise, it contains false (.F.).

`message` `(Assignable)`

Contains a character string that describes the push button. It is displayed on the screen's status bar line.

`row` (Assignable)

Contains a numeric value that indicates the screen row where the push button is displayed.

`sBlock` (Assignable)

Contains an optional code block that, when present, is evaluated each time the PushButton object's state changes. The code block takes no implicit arguments. Use the PushButton:buffer variable to determine if the push button is pressed or released. A value of true (.T.) indicates that it is being pressed; otherwise a value of false (.F.) indicates that it is being released.

This code block is included in the PushButton class to provide a method of indicating when a state change event has occurred. The name "sBlock" refers to state block.

`sizeX` (Assignable)

Contains a numeric value that indicates the size of the button in pixels on the X-coordinate (horizontally). This instance variable only affects applications running in graphic mode and is ignored in text mode.

`sizeY` (Assignable)

Contains a numeric value that indicates the size of the button in pixels on the Y-coordinate (vertically). This instance variable only affects applications running in graphic mode and is ignored in text mode.

`style` (Assignable)

Contains a character string that indicates the delimiter characters that are used by the push button's display() method. The string must contain either zero, two or eight characters. The default is two characters. The first is the left delimiter. Its default value is the less than (<) character. The second character is the right delimiter. Its default value is the greater than (>) character.

When the string is empty the button has no delimiters.

When the string length is two, the button has left and right delimiters and occupies one row on the screen. The first character is the left delimiter. The second character is the right delimiter.

When the string length is eight, the button is contained within a box that occupies three rows on the screen. A PushButton object may only be either one or three lines in height.

Standard Box Types

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top/bottom, double-line sides
B_DOUBLE_SINGLE	Double-line top/bottom, single-line sides

Box.ch contains manifest constants for the PushButton:style value.

The default style for the PushButton class is "<>".

Note: The style instance variable is ignored in graphic mode.

`typeOut`

Contains the logical value false (.F.). PushButton:typeOut never changes. It is not used by the PushButton object and is only provided for compatibility with the other GUI control classes.

Exported Methods

```
<oPushButton>:display() → self
```

display() is a method of the PushButton class that is used for showing a push button on the screen. display() uses the values of the following instance variables to correctly show the push button in its current context, in addition to providing maximum flexibility in the manner a push button appears on the screen: buffer, caption, col, colorSpec, hasFocus, row, and style.

```
<oPushButton>:hitTest( <nMouseRow>, <nMouseCol> )
    → <nHitStatus>
```

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the push button.

Applicable Hit Test Return Values

Value	Constant	Description
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the push button occupies
-1	HTTOPLEFT	The mouse cursor is on the top-left corner of the push button's border
-2	HTTOP	The mouse cursor is on the push button's top border
-3	HTTOPRIGHT	The mouse cursor is on the top-right corner of the push button's border
-4	HTRIGHT	The mouse cursor is on the push button's right border
-5	HTBOTTOMRIGHT	The mouse cursor is on the bottom-right corner of the push button's border
-6	HTBOTTOM	The mouse cursor is on the push button's bottom border
-7	HTBOTTOMLEFT	The mouse cursor is on the bottom-left corner of the push button's border
-8	HTLEFT	The mouse cursor is on the push button's left border
-2049	HTCLIENT	The mouse cursor is on the push button

Button.ch contains manifest constants for the PushButton:hitTest() return value.

hitTest() is a method of the PushButton class that is used for determining if the mouse cursor is within the region of the screen that the push button occupies.

```
<oPushButton>:killFocus() → self
```

killFocus() is a method of the PushButton class that is used for taking input focus away from a PushButton object. Upon receiving this message, the PushButton object redisplays itself and, if present, evaluates the code block within its fBlock variable.

This message is meaningful only when the PushButton object has input focus.

```
<oPushButton>:select( [<nInkeyValue>] ) → self
```

<*nInkeyValue*> is a numeric value that indicates the key that triggered the push button's activation. If passed, Select() waits for the key specified by <*nInkeyValue*> to be released before continuing.

select() is a method of the PushButton class that is used for activating a push button. When activated, a push button performs several operations. First, select() sets SELF:BUFFER to true (.T.). Then, it calls SELF:display() to show the button in its highlighted color. If <*nInkeyValue*> is passed, it waits for the key specified by <*nInkeyValue*> to be released. Then, if present, it evaluates its sBlock code block. select() then calls SELF:display() to show the button in its selected color. A push button's state is typically changed when the space bar or enter key is pressed or the mouse's left button is pressed when its cursor is within the push button's screen region.

This message is meaningful only when the PushButton object has input focus.

```
<oPushButton>:setFocus() → self
```

setFocus() is a method of the PushButton class that is used for giving focus to a PushButton object. Upon receiving this message, the PushButton object redisplays itself and, if present, evaluates the code block within its fBlock variable.

This message is meaningful only when the PushButton object does not have input focus.

Examples

- This example creates an object which is placed at row 2 and column 2 and has a caption of "Exit":

```
oBtn>:PushButton(2,2,"Exit")
```

QOUT() function

Display a list of expressions to the console

Syntax

```
QOUT([<exp list>]) → NIL
QQOUT([<exp list>]) → NIL
```

Arguments

<exp list> is a comma-separated list of expressions (of any data type other than array or block) to display to the console. If no argument is specified and QOUT() is specified, a carriage return/line feed pair is displayed. If QQOUT() is specified without arguments, nothing displays.

Returns

QOUT() and QQOUT() always return NIL.

Description

QOUT() and QQOUT() are console functions. These are the functional primitives that create the ? and ?? commands, respectively. Like the ? and ?? commands, they display the results of one or more expressions to the console. QOUT() outputs carriage return and line feed characters before displaying the results of *<exp list>*. QQOUT() displays the results of *<exp list>* at the current ROW() and COL() position. When QOUT() and QQOUT() display to the console, ROW() and COL() are updated. If SET PRINTER is ON, PROW() and PCOL() are updated instead. If *<exp list>* is specified, both QOUT() and QQOUT() display a space between the results of each expression.

You can use QOUT() and QQOUT() for console display within an expression. This is particularly useful for blocks, iteration functions such as AEVAL() and DBEVAL(), and in a list of statements in the output pattern of a user-defined command definition.

Examples

- This example uses QOUT() with AEVAL() to list the contents of a literal array to the console:

```
LOCAL aElements := { 1, 2, 3, 4, 5 }
AEVAL(aElements, { |element| QOUT(element) })
```

Files: Library is CLIPPER.LIB.

See also: ? | ??, @...SAY, SET ALTERNATE, SET CONSOLE, SET PRINTER, TEXT

QUIT command

Terminate program processing

Syntax

```
QUIT | CANCEL*
```

Description

QUIT and CANCEL both terminate program processing, close all open files, and return control to the operating system. Each of these commands can be used from anywhere in a program system. A RETURN executed at the highest level procedure or a BREAK, with no pending SEQUENCE, also QUITs the program.

Notes

- ***Return code:*** When a CA-Clipper program terminates, the return code is set to 1 if the process ends with a fatal error. If the process ends normally, the return code is set to zero or the last ERRORLEVEL() set in the program.

Examples

- This example uses QUIT in a dialog box:

```
IF DialogYesNo(10, 10, "Quit to DOS", "BG+/B,B/W", 2)
   QUIT
ENDIF
RETURN
```

Files: Library is CLIPPER.LIB.

See also: BEGIN SEQUENCE, ERRORLEVEL(), RETURN

RadioButto class

Create a radio button

Description

Radio buttons are typically presented in related groups and provide mutually exclusive responses to a condition where only one choice is appropriate. For example, a group of radio buttons might allow you to sort public variables either by name or by address.

Only one radio button can be on—when a new button is pressed, the previously selected button is turned off.

Class Function

```
RadioButto( <nRow>, <nColumn>, [<cCaption>] )
   → oRadioButto
```

Arguments

<nRow> is a numeric value that indicates the screen row of the radio button.

<nColumn> is a numeric value that indicates the screen column of the radio button.

<cCaption> is an optional character string that contains a text string that describes the radio button on the screen. The default is an empty string.

Returns

Returns a RadioButto object when all of the required arguments are present; otherwise, RadioButto() returns NIL.

Exported Instance Variables

`bitmaps` (Assignable)

Contains an array of exactly two elements. The first element of this array is the file name of the bitmap to be displayed when the radio button is selected. The second element of this array is the file name of the bitmap to be displayed when the radio button is not selected.

Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and, if it is not found, search in the bitmap library second. If no file is found either on disk or in the library, no bitmap will be displayed.

If this instance variable is not used, and the application is running in graphic mode, the files RADIO_F.BMU and RADIO_E.BMU will be used for the selected bitmap and unselected bitmap, respectively.

This instance variable only affects applications running in graphic mode and is ignored in text mode.

`buffer`

Contains a logical value that indicates whether a radio button is selected or not. A value of true (.T.) indicates that it selected; otherwise, a value of false (.F.) indicates that it is not.

`caption` (Assignable)

Contains a character string that describes the radio button on the screen. When present, the **&** character specifies that the character immediately following it in the caption is the radio button's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one radio button to another. The user performs the selection by pressing an accelerator key. The case of an accelerator key is ignored.

`capCol` (Assignable)

Contains a numeric value that indicates the screen column where the radio button's caption is displayed.

`capRow` (Assignable)

Contains a numeric value that indicates the screen row where the radio button's caption is displayed.

`cargo` (Assignable)

Contains a value of any type that is ignored by the RadioButto object. RadioButto:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a RadioButto object and retrieved later.

`col` (Assignable)

Contains a numeric value that indicates the screen column where the radio button is displayed.

`colorSpec` (Assignable)

Contains a character string that indicates the color attributes that are used by the radio button's display() method. The default is the system's current unselected and enhanced colors. The string must contain seven color specifiers.

Note: In graphic mode, colorSpec positions 1 through 4 have no affect and are ignored.

RadioButto Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	A radio button when it is unselected and does not have input focus	Unselected
2	A radio button when it is selected and does not have input focus	Unselected
3	A radio button when it is unselected and has input focus	Enhanced
4	A radio button when it is selected and has input focus	Enhanced
5	A radio button's caption	Standard
6	A radio button caption's accelerator key when it does not have input focus	Standard
7	A radio button caption's accelerator key when it has input focus	Background

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Workbench Properties window for that item.

`fBlock` `(Assignable)`

Contains an optional code block that, when present, is evaluated each time the RadioButto object receives or loses input focus. The code block takes no implicit arguments. Use the RadioButto:hasFocus instance variable to determine if the radio button is receiving or losing input focus. A value of true (.T.) indicates that it is receiving input focus; otherwise, a value of false (.F.) indicates that it is losing input focus.

This code block is included in the RadioButto class to provide a method of indicating when an input focus change event has occurred. The name "fBlock" refers to focus block.

`hasFocus`

Contains a logical value that indicates whether the RadioButto object has input focus. RadioButto:hasFocus contains true (.T.) if it has input focus; otherwise, it contains false (.F.) .

`row` `(Assignable)`

Contains a numeric value that indicates the screen row where the radio button is displayed.

`sBlock` `(Assignable)`

Contains an optional code block that, when present, is evaluated each time the RadioButto object's state changes. The code block takes no implicit arguments. Use the RadioButto:buffer instance variable to determine whether the radio button is being selected or unselected. A value of true (.T.) indicates that it is being selected; otherwise, a value of false (.F.) indicates that it is being unselected.

This code block is included in the RadioButto class to provide a method of indicating when a state change event has occurred. The name "sBlock" refers to state block.

`style` (Assignable)

Contains a character string that indicates the characters that are used by the radio button's display() method. The string must contain four characters. The first is the left delimiter. Its default value is the left parenthesis (( ) character. The second is the selected indicator. Its default value is the asterisk (*) character. The third is the unselected indicator. Its default is the space (" ") character. The fourth character is the right delimiter. Its default value is the right parenthesis ( )) character.

Note: In graphic mode, the style instance variable is ignored.

Exported Methods

`<oRadioButto>:display()` → *self*

display() is a method of the RadioButto class that is used for showing a radio button and its caption on the screen. display() uses the values of the following instance variables to correctly show the radio button in its current context in addition to providing maximum flexibility in the manner a radio button appears on the screen: buffer, caption, capCol, capRow, col, colorSpec, hasFocus, row, and style.

`<oRadioButto>:hitTest( <nMouseRow>, <nMouseCol> )`
→ *<nHitStatus>*

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the radio button.

Applicable Hit Test Return Values

Value	Constant	Description
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the radio button occupies
-1025	HTCAPTION	The mouse cursor is on the radio button's caption
-2049	HTCLIENT	The mouse cursor is on the radio button

Button.ch contains manifest constants for the RadioButto:hitTest() return value.

hitTest() is a method of the RadioButto class that is used for determining if the mouse cursor is within the region of the screen that the radio button occupies.

hitTest() returns a numeric value in order to maintain an appropriate level of symmetry with the hitTest() methods contained within the other data input control classes.

```
<oRadioButto>:isAccel( <nInkeyValue> )
    → <lHotKeyStatus>
```

<nInkeyValue> is a numeric value that indicates the inkey value to check.

Returns a logical value that indicates whether the value specified by *<nInkeyValue>* should be treated as a hot key. A value of true (.T.) indicates that the key should be treated as a hot key; otherwise, a value of false (.F.) indicates that it should not.

isAccel() is a method of the RadioButto class that is used for determining whether a key press should be interpreted as a user request to select a radio button.

```
<oRadioButto>:killFocus() → self
```

killFocus() is a method of the RadioButto class that is used for taking input focus away from a RadioButto object. Upon receiving this message, the RadioButto object redisplays itself and, if present, evaluates the code block within its fBlock instance variable.

This message is meaningful only when the RadioButto object has input focus.

```
<oRadioButto>:select( [<lNewState>] ) → self
```

<lNewState> is a logical value that indicates whether the radio button should be selected or not. Set to true (.T.) to select the button or false (.F.) to deselect the button. If omitted, the radio button state will toggle to its opposing state.

select() is a method of the RadioButto class that is used for changing the state of a radio button. Its state is typically changed when the space bar or one of the arrow keys is pressed or the mouse's left button is pressed when its cursor is within the radio button's screen region.

This message is meaningful only when the RadioButto object has input focus, or when the radio button is a member of a Group object that has input focus.

```
<oRadioButto>:setFocus() → self
```

setFocus() is a method of the RadioButto class that is used for giving focus to a RadioButto object. Upon receiving this message, the RadioButto object redisplays itself and, if present, evaluates the code block within its fBlock instance variable.

This message is meaningful only when the RadioButto object does not have input focus.

Examples

- This example creates two radio buttons, one with a caption of "Commands" and the other "Macros," and groups them together using the RadioGroup class:

```
oRadio1:RadioButto(2,2,"Commands")
oRadio2:RadioButto(3,2,"Macros")
oRadiogroup:RadioGroup(2,2,3,9)
```

RadioGroup class

Create a radio button group

Description

The RadioGroup class provides a convenient mechanism for manipulating radio buttons.

Class Function

```
RadioGroup( <nTop>, <nLeft>, <nBottom>, <nRight> )
   → <oRadioGroup>
```

Arguments

<nTop> is a numeric value that indicates the top screen row of the radio group.

<nLeft> is a numeric value that indicates the left screen column of the radio group.

<nBottom> is a numeric value that indicates the bottom screen row of the radio group.

<nRight> is a numeric value that indicates the right screen column of the radio group.

Returns

Returns a RadioGroup object when all of the required arguments are present; otherwise, RadioGroup() returns NIL.

Exported Instance Variables

`bottom` (Assignable)

Contains a numeric value that indicates the bottommost screen row where the radio group is displayed.

`buffer` (Assignable)

Contains a numeric value that indicates the position in the radio group of the selected radio button.

`capCol` (Assignable)

Contains a numeric value that indicates the screen column where the radio group's caption is displayed.

`capRow` (Assignable)

Contains a numeric value that indicates the screen row where the radio group's caption is displayed.

`caption` (Assignable)

Contains a character string that concisely describes the radio group on the screen.

When present, the & character specifies that the character immediately following it in the caption is the radio group's accelerator key. The accelerator key provides a quick and convenient mechanism for the user to move input focus from one data input control to a radio group. The user performs the selection by pressing the Alt key in combination with an accelerator key. The case of an accelerator key is ignored.

`cargo` (Assignable)

Contains a value of any type that is ignored by the RadioGroup object. RadioGroup:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a RadioGroup object and retrieved later.

`coldBox` (Assignable)

Contains an optional string that specifies the characters to use when drawing a box around the radio group when it does not have input focus. Its default value is a single-line box.

Standard Box Types

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top/bottom, double-line sides
B_DOUBLE_SINGLE	Double-line top/bottom, single-line sides

Box.ch contains manifest constants for the RadioGroup:coldBox value.

`colorSpec` `(Assignable)`

Contains a character string that indicates the color attributes that are used by the radio group's display() method. The string must contain three color specifiers.

RadioGroup Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The radio group's border	Border
2	The radio group's caption	Standard
3	The radio group caption's accelerator key	Background

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Workbench Properties window for that item.

`block` `(Assignable)`

Contains an optional code block that, when present, is evaluated each time the RadioGroup object receives or loses input focus. The code block takes no implicit arguments. Use the RadioGroup:hasFocus instance variable to determine if the radio group is receiving or losing input focus. A value of true (.T.) indicates that it is receiving input focus; otherwise, a value of false (.F.) indicates that it is losing input focus.

This code block is included in the RadioGroup class to provide a method of indicating when an input focus change event has occurred. The name "fBlock" refers to focus block.

`hasFocus`

Contains a logical value that indicates whether the RadioGroup object has input focus. It should contain true (.T.) when it has input focus; otherwise, it should contain false (.F.).

`hotBox` (Assignable)

Contains an optional string that specifies the characters to use when drawing a box around the radio group when it has input focus. Its default value is a double-line box.

Standard Box Types

Constant	Description
B_SINGLE	Single-line box
B_DOUBLE	Double-line box
B_SINGLE_DOUBLE	Single-line top/bottom, double-line sides
B_DOUBLE_SINGLE	Double-line top/bottom, single-line sides

Box.ch contains manifest constants for the RadioGroup:hotBox value.

`itemCount` (Assignable)

Contains a numeric value that indicates the total number of radio buttons in the RadioGroup object.

`left` (Assignable)

Contains a numeric value that indicates the leftmost screen column where the radio group is displayed.

`message` (Assignable)

Contains a character string that is the radio group's description that is displayed on the screen's status bar line.

`right` (Assignable)

Contains a numeric value that indicates the rightmost screen column where the radio group is displayed.

`top` (Assignable)

Contains a numeric value that indicates the topmost screen row where the radio group is displayed.

`typeOut` (Assignable)

Contains a logical value that indicates whether the group contains any buttons. A value of true (.T.) indicates the group contains selectable buttons; a false (.F.) value indicates that the group is empty.

Exported Methods

`<oRadioGroup>:addItem( <oRadioButto> ) → self`

<oRadioButto> is the radio button object to be added.

addItem() is a method of the RadioGroup class that is used for appending a new radio button to a radio group.

`<oRadioGroup>:delItem( nPosition> ) → self`

<nPosition> is a numeric value that indicates the position in the radio group of the radio button to be deleted.

delItem() is a method of the RadioGroup class that is used for removing a radio button from a radio group.

`<oRadioGroup>:display() → self`

display() is a method of the RadioGroup class used for showing its radio buttons on the screen. display() accomplishes this by calling the display() method of each of the radio buttons in its group.

`<oRadioGroup>:getAccel( <nInkeyValue> ) → <nPosition>`

<nInkeyValue> is a numeric value that indicates the inkey value to check.

Returns a numeric value that indicates the position in the radio group of the first item whose accelerator key matches that which is specified by <nInkeyValue>.

getAccel() is a method of the RadioGroup class that is used for determining whether a key press should be interpreted as a user request to select a particular radio button. getAccel() accomplishes this by calling the getAccel() method of each of the radio buttons in its group.

`<oRadioGroup>:getItem( <nPosition> ) → <oRadioButto>`

<nPosition> is a numeric value that indicates the position in the list of the item that is being retrieved.

Returns the RadioButto object specified by <nPosition>.

getItem() is a method of the RadioGroup class that is used for retrieving a radio button from a radio group.

```
<oRadioGroup>:hitTest( <nMouseRow>, <nMouseCol> )
    → <nHitStatus>
```

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the radio group.

Applicable Hit Test Return Values

Value	Constant	Description
> 0	Not Applicable	The position in the radio group of the radio button whose region the mouse is within
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the radio group occupies
-1	HTTOPLEFT	The mouse cursor is on the radio group's border
-2	HTTOP	The mouse cursor is on the radio group's top border
-3	HTTOPRIGHT	The mouse cursor is on the top right corner of the radio group's border
-4	HTRIGHT	The mouse cursor is on the radio group's right border
-5	HTBOTTOMRIGHT	The mouse cursor is on the bottom right corner of the radio group's border
-6	HTBOTTOM	The mouse cursor is on the radio group's bottom border
-7	HTBOTTOMLEFT	The mouse cursor is on the bottom left corner of the radio group's border
-8	HTLEFT	The mouse cursor is on the radio group's left border
-2049	HTCLIENT	The mouse cursor is within the radio group's screen region but not on a radio button

Button.ch contains manifest constants for the RadioGroup:hitTest() return value.

hitTest() is a method of the RadioGroup class that is used for determining if the mouse cursor is within the region of the screen that any of the RadioButto objects contained within the radio group occupies. hitTest() accomplishes this by calling the hitTest() method of each of the radio buttons in its group.

```
<oRadioGroup>:insItem( <nPosition>, <oRadioButto> )
   → self
```

<nPosition> is a numeric value that indicates the position at which the new item is inserted.

<oRadioButto> is the RadioButto object to be inserted.

insItem() is a method of the RadioGroup class that is used for inserting a new radio button into a radio group.

```
<oRadioGroup>:killFocus() → self
```

killFocus() is a method of the RadioGroup class that is used for taking input focus away from a RadioGroup object. Upon receiving this message, the RadioGroup object redisplays itself and, if present, evaluates the code block within its fBlock instance variable.

This message is meaningful only when the RadioGroup object has input focus.

```
<oRadioGroup>:nextItem() → self
```

nextItem() is a method of the RadioGroup class that is used for changing the selected radio button from the current item to the one immediately following it.

This message is meaningful only when the RadioGroup object has input focus.

```
<oRadioGroup>:prevItem() → self
```

prevItem() is a method of the RadioGroup class that is used for changing the selected item from the current radio button to the one immediately before it.

This message is meaningful only when the RadioGroup object has input focus.

```
<oRadioGroup>:select( <nPosition> ) → self
```

<*nPosition*> is a numeric value that indicates the position in the list of the radio button to be selected.

select() is a method of the RadioGroup class that is used for changing the selected radio button in a radio group. Its state is typically changed when one of the arrow keys is pressed or the mouse's left button is pressed when its cursor is within one of the radio button's screen region.

This message is meaningful only when the RadioGroup object has input focus.

```
<oRadioGroup>:setColor( [<cColorString>] ) → self
```

<*cColorString*> is a character string that indicates the color attributes that are used by the radio button's display() method. The string must contain seven color specifiers.

RadioButto Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	A radio button when it is unselected and does not have input focus	Unselected
2	A radio button when it is selected and does not have input focus	Unselected
3	A radio button when it is unselected and has input focus	Enhanced
4	A radio button when it is selected and has input focus	Enhanced
5	A radio button's caption	Standard
6	A radio button caption's accelerator key when it does not have input focus	Standard
7	A radio button caption's accelerator key when it has input focus	Background

setColor() is a method of the RadioGroup class that is used for uniformly setting the color attributes of all the radio buttons in its group. setColor() accomplishes this by setting the colorSpec

instance variable of each of the radio buttons in its group to the value specified by *<cColorString>*.

```
<oRadioGroup>:setFocus() → self
```

:tFocus() is a method of the RadioGroup class that is used for giving focus to a RadioGroup object. Upon receiving this message, the RadioGroup object redisplays itself and, if present, evaluates the code block within its fBlock instance variable.

This message is meaningful only when the RadioGroup object does not have input focus.

```
<oRadioGroup>:setStyle( [<cStyle>] ) → self
```

<cStyle> is a character string that indicates the characters that are used by the radio button's display() method. The string must contain four characters. The first is the left delimiter. The second is the selected indicator. The third is the unselected indicator. The fourth character is the right delimiter.

setStyle() is a method of the RadioGroup class that is used for uniformly setting the Style attribute of all the radio buttons in its group. setStyle() accomplishes this by setting the Style of each of the radio buttons in its group to the value specified by *<cStyle>*.

Examples

- This example creates a group of radio buttons, one with a caption of "Commands" and the other "Macros":

```
oRadio1:RadioButto(2,2,"Commands")
oRadio2:RadioButto(3,2,"Macros")
oRadiogroup:RadioGroup(2,2,3,9)
```

RAT() function

Return the position of the last occurrence of a substring

Syntax

```
RAT(<cSearch>, <cTarget>) → nPosition
```

Arguments

<cSearch> is the character string to be located.

<cTarget> is the character string to be searched.

Returns

RAT() returns the position of *<cSearch>* within *<cTarget>* as an integer numeric value. If *<cSearch>* is not found, RAT() returns zero.

Description

RAT() is a character function that returns the position of the last occurrence of a character substring within another character string. It does this by searching the target string from the right. RAT() is like the AT() function, which returns the position of the first occurrence of a substring within another string. RAT() is also like the $ operator, which determines whether a substring is contained within a string.

Both the RAT() and AT() functions are used with SUBSTR(), LEFT(), and RIGHT() to extract substrings.

Examples

- This example uses RAT() to create a user-defined function, FilePath(), that extracts the path from a file specification. If the path is unspecified, FilePath() returns a null string (""):

```
? FilePath("C:\DBF\Sales.dbf")      // Result: C:\DBF\

FUNCTION FilePath( cFile )
    LOCAL nPos, cFilePath
    IF (nPos := RAT("\", cFile)) != 0
       cFilePath = SUBSTR(cFile, 1, nPos)
    ELSE
       cFilePath = ""
    ENDIF
    RETURN cFilePath
```

Files

Library is EXTEND.LIB.

See Also

AT(), LEFT(), RIGHT(), STRTRAN(), SUBSTR()

RDDLIST() function

Return an array of the available Replaceable Database Drivers (RDDs)

Syntax

```
RDDLIST([<nRDDType>]) → aRDDList
```

Arguments

<nRDDType> is an integer that represents the type of the RDD you wish to list. The constants RDT_FULL and RDT_TRANSFER represent the two types of RDDs currently available.

RDDType Summary

Constant	Value	Meaning
RDT_FULL	1	Full RDD implementation
RDT_TRANSFER	2	Import/Export only driver

RDT_FULL identifies full-featured RDDs that have all the capabilities associated with an RDD.

RDT_TRANSFER identifies RDDs of limited capability. They can only transfer records between files. You cannot use these limited RDD drivers to open a file in a work area. The SDF and DELIM drivers are examples of this type of RDD. They are only used in the implementation of APPEND FROM and COPY TO with SDF or DELIMITED files.

Returns

RDDLIST() returns a one-dimensional array of the RDD names registered with the application as *<nRDDType>*.

Description

RDDLIST() is an RDD function that returns a one-dimensional array that lists the available RDDs.

If you do not supply *<nRDDType>*, all available RDDs regardless of type are returned.

Examples

- In this example RDDLIST() returns an array containing the character strings, "DBF", "SDF", "DELIM", "DBFCDX", and "DBFNTX":

```
REQUEST DBFCDX

.
. < statements >
.

aRDDs := RDDLIST()

      // Returns {"DBF", SDF", "DELIM", "DBFCDX", "DBFNTX" }
```

- In this example, RDDLIST() returns an array containing the character strings, "SDF" and "DELIM":

```
#include "rddsys.ch"
.
. < statements >
.
aImpExp := RDDLIST( RDT TRANSFER )
```

Files Rddsys.ch.

See Also DBSETDRIVER(), RDDNAME(), RDDSETDEFAULT()

RDDNAME() function

Return the name of the RDD active in the current or specified work area

Syntax

```
RDDNAME() → cRDDName
```

Returns

Returns a character string, *cRDDName*, the registered name of the active RDD in the current or specified work area.

Description

RDDNAME() is an RDD function that returns a character string for the name of the active RDD, *cRDDName*, in the current or specified work area.

You can specify a work area other than the currently active work area by aliasing the function.

Examples

```
USE Customer VIA "DBFNTX" NEW
USE Sales    VIA "DBFCDX" NEW

? RDDNAME()                          // Returns: DBFCDX
? Customer->( RDDNAME() )            // Returns: DBFNTX
? Sales->( RDDNAME() )               // Returns: DBFCDX
```

See Also

RDDLIST()

RDDSETDEFAULT() function

Set or return the default RDD for the application

Syntax

```
RDDSETDEFAULT([<cNewDefaultRDD>])
   → cPreviousDefaultRDD
```

Arguments

<cNewDefaultRDD> is a character string, the name of the RDD that is to be made the new default RDD in the application.

Returns

RDDSETDEFAULT() returns a character string, *cPreviousDefaultRDD*, the name of the previous default driver. The default driver is the driver that CA-Clipper uses if you do not explicitly specify an RDD with the VIA clause of the USE command.

Description

RDDSETDEFAULT() is an RDD function that sets or returns the name of the previous default RDD driver and, optionally, sets the current driver to the new RDD driver specified by *cNewDefaultRDD*. If *<cNewDefaultDriver>* is not specified, the current default driver name is returned and continues to be the current default driver.

This function replaces the DBSETDRIVER() function.

Examples

```
// If the default driver is not DBFNTX, make it the default

IF ( RDDSETDEFAULT() != "DBFNTX" )
   cOldRdd := RDDSETDEFAULT( "DBFNTX" )
ENDIF
```

See Also

DBSETDRIVER()

READ command

Activate full-screen editing mode using Get objects

Syntax

```
READ [SAVE] [MENU <oMenu>] [MSG AT <nRow>, <nLeft>,
   <nRight>] [MSG COLOR <cColorString>]
```

Arguments

SAVE retains the contents of the current *GetList* after the READ terminates. Later, you can edit the same Get objects by issuing another READ. If not specified, the current *GetList* is assigned an empty array deleting all of the previous Get objects when the READ terminates.

MENU <oMenu> specifies an optional TopBarMenu object that, when supplied, permits menu selection during data entry.

MSG AT <nMsgRow>, <nMsgLeft>, <nMsgRight> specify the row, left, and right margins where the Get object messages appear on the screen. If omitted, messages will not appear.

MSG COLOR <cMsgColor> defines the color setting of the message area. It consists of a single foreground/background color pair.

Description

READ executes a full-screen editing mode using all Get objects created and added to the current *GetList* since the most recent CLEAR, CLEAR GETS, CLEAR ALL or READ commands. If there is a format procedure active, READ executes that procedure before entering the full-screen editing mode.

Within a READ, the user can edit the buffer of each Get object as well as move from one Get object to another. Before the user can enter a Get object, control passes to the associated WHEN *<lPreCondition>* if one has been assigned to that Get object. If *<lPreCondition>* returns true (.T.), the user is allowed to edit the buffer of the Get object. Otherwise, control passes to the next Get object in the *GetList*. Within a GET buffer, the user can edit using the full complement of editing and navigation keys. See the tables below.

When the user presses a GET exit key, control passes to the associated RANGE or VALID postcondition if one has been specified. If either condition returns true (.T.), editing of the Get object is terminated and control passes to the next Get object. Otherwise, control remains within the current Get object until a valid value is entered or the user presses the Esc key.

When the user successfully enters a value into a Get object, the associated variable is assigned the value of the Get object's buffer.

The following tables list active keys within a READ:

READ Navigation Keys

Key	Action
Left arrow, Ctrl+S	Character left. Does not move cursor to previous GET.
Right arrow, Ctrl+D	Character right. Does not move cursor to next GET.
Ctrl+Left arrow, Ctrl+A	Word left.
Ctrl+Right arrow, Ctrl+F	Word right.
Up arrow, Shift+Tab, Ctrl+E	Previous GET.
Down arrow, Tab, Ctrl+X, Return, Ctrl+M	Next GET.
Home	First character of GET.
End	Last character of GET.
Ctrl+Home	Beginning of first GET.

READ Editing Keys

Key	Action
Del, Ctrl+G	Delete character at cursor position
Backspace, Ctrl+H	Destructive backspace
Ctrl+T	Delete word right
Ctrl+Y	Delete from cursor position to end of GET
Ctrl+U	Restore current GET to original value

READ Toggle Keys

Key	Action
Ins, Ctrl+V	Toggle insert mode

READ Exit Keys

Key	Action
Ctrl+W, Ctrl+C, PgUp, PgDn	Terminate READ saving current GET
Return, Ctrl+M	Terminate READ from last GET
Esc	Terminate READ without saving current GET
Up arrow	Terminate READ from first GET if READEXIT()=.T.
Down arrow	Terminate READ from last GET if READEXIT()=.T.

Notes

- ***Nested READs:*** To perform a nested READ within a SET KEY, VALID, or WHEN procedure or user-defined function, declare or create a new *GetList*, perform a series of @...GET statements, and then READ. When the procedure terminates, the new *GetList* is released and the previous *GetList* becomes visible again. See the example below.
- ***Quick keys:*** Pressing Home or End in quick succession goes to the first or last nonblank character in a Get object's buffer.
- ***Terminating a READ:*** A READ is terminated by executing a BREAK, CLEAR, CLEAR GETS, or CLEAR ALL from within a SET KEY procedure or a user-defined function initiated by VALID.
- ***UPDATED():*** If any Get object buffer was changed during the current READ, UPDATED() is set to true (.T.).

Examples

- This example defines several GETs then READs them:

```
CLEAR
cVar1 := cVar2 := cVar3 := SPACE(10)
@ 10, 10 SAY "Variable one:" GET cVar1 VALID ;
      !EMPTY(cVar1)
@ 11, 10 SAY "Variable two:" GET cVar2 ;
         WHEN RTRIM(cVar1) !=  "One"
@ 12, 10 SAY "Variable three:" GET cVar3 VALID ;
         !EMPTY(cVar3)
READ
```

- This example performs a nested READ within a SET KEY, WHEN, or VALID procedure or user-defined function:

```
LOCAL cName := SPACE(10)
@ 10, 10 GET cName VALID SubForm( cName )
READ
RETURN

FUNCTION SubForm( cLookup )
    LOCAL GetList := {}          // Create new GetList
    USE Sales INDEX Salesman NEW
    SEEK cLookup
    IF FOUND()
       @ 15, 10 GET Salesman     // Add Get objects to
       @ 16, 10 GET Amount       // new GetList
       READ                      // READ from new GetList
    ENDIF
    CLOSE Sales
    RETURN .T.                   // Release new GetList
```

Files

Library is CLIPPER.LIB.

See Also

@...GET, @...SAY, CLEAR GETS, LASTKEY(), READEXIT(), READMODAL()

READEXIT() function

Toggle Up arrow and Down arrow as READ exit keys

Syntax

```
READEXIT([<lToggle>]) → lCurrentState
```

Arguments

<lToggle> toggles the use of Up arrow and Down arrow as READ exit keys. Specifying true (.T.) enables them as exit keys, and false (.F.) disables them.

Returns

READEXIT() returns the current setting as a logical value.

Description

READEXIT() is an environment function that reports the current state of Up arrow and Down arrow as keys the user can press to exit a READ from the first or last Get object in a *GetList*. If the optional *<lToggle>* argument is specified, Up arrow and Down arrow are either enabled or disabled as READ exit keys. At program startup, Up arrow and Down arrow are not enabled as READ exit keys. Normally, READ exit keys include only PgUp, PgDn, Esc, or Return from the last GET.

Examples

- This example shows READEXIT() enabling Up arrow and Down arrow exit keys before a READ then resetting them after the READ terminates:

```
cMyvar = SPACE(10)
lLastExit = READEXIT(.T.)  // Result: Turn on exit keys
//
@ 10, 10 SAY "Enter: " GET cMyvar
READ
READEXIT(lLastExit)        // Result: Restore previous setting
```

Files

Library is CLIPPER.LIB.

See Also

@...GET, READ, READINSERT()

READFORMAT() function

Return and optionally, set the code block that implements a format (.fmt) file

Syntax

```
READFORMAT([<bFormat>]) → bCurrentFormat
```

Arguments

<bFormat> is the name of the code block, if any, to use for implementing a format file. If no argument is specified, the function simply returns the current code block without setting a new one.

Returns

READFORMAT() returns the current format file as a code block. If no format file has been set, READFORMAT() returns NIL.

Description

READFORMAT() is a Get system function that accesses the current format file in its internal code block representation. It lets you manipulate the format file code block from outside of the Get system's source code.

To set a format file, use SET FORMAT (see the SET FORMAT entry) or READFORMAT().

READFORMAT() is intended primarily for creating new READ layers. The code block that READFORMAT() returns, when evaluated, executes the code that is in the format file from which it was created.

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

READKILL(), READUPDATED(), SET FORMAT

READINSERT() function

Toggle the current insert mode for READ and MEMOEDIT()

Syntax

```
READINSERT([<lToggle>]) → lCurrentMode
```

Arguments

<lToggle> toggles the insert mode on or off. True (.T.) turns insert on, while false (.F.) turns insert off. The default is false (.F.) or the last user-selected mode in READ or MEMOEDIT().

Returns

READINSERT() returns the current insert mode state as a logical value.

Description

READINSERT() is an environment function that reports the current state of the insert mode for READ and MEMOEDIT() and, optionally, sets the insert mode on or off depending on the value of *<lToggle>*. When READINSERT() returns false (.F.) and the user enters characters into a Get object's buffer during a READ or a MEMOEDIT(), characters are overwritten. When READINSERT() returns true (.T.), entered characters are inserted instead. The insert mode is a global setting belonging to the system and not to any specific object.

You can execute READINSERT() prior to or during a READ or MEMOEDIT(). If used with READ, READINSERT() can be invoked within a WHEN or VALID clause of @...GET or within a SET KEY procedure. If used with MEMOEDIT(), it can be invoked with the user function as well as a SET KEY procedure.

Examples

- This example sets the insert mode prior to entering MEMOEDIT() and resets the mode when MEMOEDIT() terminates:

```
USE Sales NEW

// Turn on insert mode
lInsMode = READINSERT(.T.)
Sales->Notes := MEMOEDIT(Sales->Notes)
//
// Restore previous insert mode
READINSERT(lInsMode)
```

Files

Library is CLIPPER.LIB.

See Also

MEMOEDIT(), READ, READEXIT()

READKEY()* function

Determine what key terminated a READ

Syntax

```
READKEY() → nReadkeyCode
```

Returns

READKEY() returns a code representing the key pressed to exit a READ. In CA-Clipper, the following keys are the standard READ exit keys and their READKEY() return codes:

READKEY() Return Codes

Exit Key	Return Code
Up arrow	5
Down arrow	2
PgUp	6
PgDn	7
Ctrl+PgUp	31
Ctrl+PgDn	30
Esc	12
Ctrl+End, Ctrl+W	14
Type past end	15
Return	15

Description

READKEY() is a keyboard function that emulates the READKEY() function in dBASE III PLUS. Its purpose is to determine what key the user pressed to terminate a READ. If UPDATED() is true (.T.), READKEY() returns the code plus 256. Up arrow and Down arrow exit a READ only if READEXIT() returns true (.T.). The default value is false (.F.). To provide complete compatibility for these keys, execute a READEXIT (.T.) at the beginning of your main procedure.

READKEY() is supplied as a compatibility function and, therefore, its use is strongly discouraged. It is superseded entirely by LASTKEY() which determines the last keystroke fetched from the keyboard buffer. If the keystroke was a READ exit key, LASTKEY() will return the INKEY() code for that key. To determine whether any Get object's buffer was modified during a READ, it is superseded by the UPDATED() function.

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\READKEY.PRG

See Also

@...GET, LASTKEY(), NEXTKEY(), READ, READEXIT(), UPDATED()

READKILL() function

Return, and optionally set, whether the current READ should be exited

Syntax

```
READKILL([<lKillRead>]) → lCurrentSetting
```

Arguments

<lKillRead> sets the READKILL() flag. A value of true (.T.) indicates that the current read should be terminated, and a value of false (.F.) indicates that it should not.

Returns

READKILL() returns the current setting as a logical value.

Description

READKILL() is a Get system function that lets you control whether or not to terminate the current READ.

Unless directly manipulated, READKILL() returns true (.T.) after you issue a CLEAR GETS (see the CLEAR GETS entry) for the current READ; otherwise, it returns false (.F.).

By accessing the function directly, however, you can control the READKILL() flag with its function argument and use it to create new READ layers.

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

CLEAR GETS, READFORMAT(), READUPDATED()

READMODAL() function

Activate a full-screen editing mode for a *GetList*

Syntax

```
READMODAL(<aGetList>, [<nGet>], [<oMenu>], [<nMsgRow>,
   <nMsgLeft>, <nMsgRight>, <cMsgColor>])
    → <lUpdated>
```

Arguments

<aGetList> is an array containing a list of Get objects to edit.

<nGet> is an optional numeric value that indicates which Get object within *<aGetList>* should initially receive input focus.

<oMenu> is an optional Topbarmenu object that, when supplied, permits menu selection during data entry.

<nMsgRow>, ***<nMsgLeft>***, and ***<nMsgRight>*** specify the row, left, and right margins where the Get object messages appear on the screen.

<cMsgColor> defines the color setting of the message area. It consists of a single foreground/background color pair.

Returns

READMODAL() returns true (.T.) when *GetList* is updated, false (.F.) when it is not.

Description

READMODAL() is a user interface function that implements the full-screen editing mode for GETs, and is part of the open architecture Get system of CA-Clipper. READMODAL() is like the READ command, but takes a *GetList* array as an argument and does not reinitialize the *GetList* array when it terminates. Because of this, you can maintain multiple lists of Get objects and activate them any time in a program's execution as long as the array to activate is visible.

In order to retain compatibility with previous versions of CA-Clipper, the GET system in CA-Clipper is implemented using a public array called *GetList*. Each time an @...GET command executes, it creates a Get object and adds to the currently visible *GetList* array. The standard READ command is preprocessed into a call to READMODAL() using the *GetList* array as its argument. If the SAVE clause is not specified, the variable *GetList* is assigned an empty array after the READMODAL() function terminates.

Some of the functions in the Getsys.prg have been made public so that they can be used when implementing customized GET readers. These functions are listed in the table below.

Get System Functions

Function	Description
GETACTIVE()	Return the currently active Get object
GETAPPLYKEY()	Apply a key to a Get object from within a GET reader
GETDOSETKEY()	Process SET KEY during GET editing
GETPOSTVALIDATE()	Postvalidate the current Get object
GETPREVALIDATE()	Prevalidate a Get object
GETREADER()	Execute standard READ behavior for a Get object
GETPREVALIDATE()	Prevalidate a Get object
READFORMAT()	Return and, optionally, set the code block that implements a format (.fmt) file
READKILL()	Return and, optionally, set whether the current READ should be exited
READUPDATED()	Return and, optionally, set whether a GET has changed during a READ

For reference information on the Get objects and functions listed above, refer to the "Get System" chapter in the *Programming and Utilities Guide*.

For more information on the supported keys in the default READMODAL() function, refer to the READ command reference in this chapter.

Files Library is CLIPPER.LIB, source file is SOURCE\SYS\GETSYS.PRG

See Also @...GET, READ

READUPDATED() function

Determine whether any GET variables changed during a READ and optionally change the READUPDATED() flag

Syntax

```
READUPDATED([<lChanged>]) → lCurrentSetting
```

Arguments

<lChanged> sets the READUPDATED() flag. A value of true (.T.) indicates that data has changed, and a value of false (.F.) indicates that no change has occurred.

Returns

READUPDATED() returns the current setting as a logical value.

Description

READUPDATED() is a Get system function intended primarily for creating new READ Layers. It is identical in functionality to UPDATED() (see the UPDATED() entry), except that it allows the UPDATED() flag to be set.

READUPDATED() enables you to manipulate the UPDATED() flag from outside of the Get system's source code.

Files

Library is CLIPPER.LIB, source file is Getsys.prg.

See Also

READFORMAT(), READKILL(), UPDATED()

READVAR() function

Return the current GET/MENU variable name

Syntax

```
READVAR() → cVarName
```

Returns

READVAR() returns the name of the variable associated with the current Get object or the variable being assigned by the current MENU TO command as an uppercase character string.

Description

READVAR() is an environment function that primarily implements context-sensitive help for Get objects and lightbar menus. READVAR() only works during a READ or MENU TO command. If used during any other wait states, such as ACCEPT, INPUT, WAIT, ACHOICE(), DBEDIT(), or MEMOEDIT(), it returns a null string (""). Access it within a SET KEY procedure, or within a user-defined function invoked from a WHEN or VALID clause of a Get object.

Examples

- This example implements a simple help system for Get objects using a database file to store the help text. When the user presses F1, the help database file is searched using READVAR() as the key value. If there is help text available, it is displayed in a window:

```
#include "Inkey.ch"
//
SET KEY K_F1 TO HelpLookup
cString = SPACE(10)
@ 5, 5 SAY "Enter:" GET cString
READ
RETURN

FUNCTION HelpLookup
   USE Help INDEX Help NEW
   SEEK READVAR()
   IF FOUND()
      DisplayHelp(Help->Topic)
   ELSE
      DisplayHelp("No help for " + READVAR())
   ENDIF
   CLOSE Help
   RETURN NIL

FUNCTION DisplayHelp( cTopic )
   LOCAL cScreen := SAVESCREEN(5,5,15,70),;
         cColor := SETCOLOR("BG+/B")
   //
   SET CURSOR OFF
   @ 5, 5 CLEAR TO 15, 70
   @ 5, 5 TO 15, 70 DOUBLE
   @ 5, 30 SAY " Help for " + READVAR() + " "
   MEMOEDIT(cTopic, 6, 7, 14, 68, .F.)
   //
   RESTSCREEN(5, 5, 15, 70, cScreen)
   SETCOLOR(cColor)
   SET CURSOR ON
   //
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also @...GET, MENU TO, READ, SET KEY

RECALL command

Restore records marked for deletion

Syntax

```
RECALL [<scope>] [WHILE <lCondition>]
   [FOR <lCondition>]
```

Arguments

<scope> is the portion of the current database file to RECALL. The default scope is the current record, or NEXT 1. If a condition is specified, the default scope becomes ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to RECALL within the given scope.

Description

RECALL is a database command that restores records marked for deletion in the current work area. This is the inverse of the DELETE command. If DELETED is ON, RECALL can restore the current record or a specific record, if you specify a RECORD scope. Note that once you PACK a database file, all marked records have been physically removed from the file and cannot be recovered.

In a network environment, RECALLing the current record requires an RLOCK(). RECALLing several records requires an FLOCK() or EXCLUSIVE USE of the current database file. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Examples

- This examples show the results of RECALL:

```
USE Sales NEW
//
DELETE RECORD 4
? DELETED()              // Result: .T.
//
RECALL
? DELETED() .            // Result: .F.
```

Files Library is CLIPPER.LIB.

See Also DELETE, DELETED(), FLOCK(), PACK, RLOCK(), SET DELETED, ZAP

RECCOUNT()* function

Determine the number of records in the current database (.dbf) file

Syntax

```
RECCOUNT()* | LASTREC() → nRecords
```

Returns

RECCOUNT() returns the number of physical records in the current database file as an integer numeric value. Filtering commands such as SET FILTER or SET DELETED have no affect on the return value. RECCOUNT() returns zero if there is no database file open in the current work area.

Description

RECCOUNT() is a database function that is a synonym for LASTREC(). By default, RECCOUNT() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below). Note that RECCOUNT() is a compatibility function. LASTREC() should be used in its place.

Examples

- This example illustrates the relationship between COUNT and RECCOUNT():

```
USE Sales NEW
? RECCOUNT()                        // Result: 84
//
SET FILTER TO Salesman = "1001"
COUNT TO nRecords
? nRecords                          // Result: 14
? RECCOUNT()                        // Result: 84
```

- This example uses an aliased expression to access the number of records in an unselected work area:

```
USE Sales NEW
USE Customer NEW
? RECCOUNT(), Sales->(RECCOUNT())
```

Files

Library is CLIPPER.LIB.

See Also

COUNT, EOF(), LASTREC()

RECNO() function

Return the identity at the position of the record pointer

Syntax

```
RECNO() → Identity
```

Returns

RECNO() returns the identity found at the position of the record pointer.

Description

RECNO() is a database function that returns the identity found at the current position of the record pointer. Identity is a unique value guaranteed by the structure of the data file to reference a specific record of a data file. The data file need not be a traditional Xbase file. Therefore, unlike earlier versions of CA-Clipper, the value returned need not be a numeric data type.

Under all RDDs, RECNO() returns the value at the position of the record pointer; the data type and other characteristics of this value are determined by the content of the accessed data and the RDD active in the current work area. In an Xbase database this value is the record number.

Examples

```
USE Sales VIA "DBFNTX"
.
. < statements >
.
DBGOTOP()
RECNO()          // Returns 1
```

See Also

DBGOTO()

RECSIZE() function

Determine the record length of a database (.dbf) file

Syntax

```
RECSIZE() → nBytes
```

Returns

RECSIZE() returns, as a numeric value, the record length in bytes of the database file open in the current work area. RECSIZE() returns zero if no database file is open.

Description

RECSIZE() is a database function that determines the length of a record by summing the lengths of each field then adding one for the DELETED() status flag. When this value is multiplied by LASTREC(), the product is the amount of space occupied by the file's records.

RECSIZE() is useful in programs that perform automatic file backup. When used in conjunction with DISKSPACE(), the RECSIZE() function can assist in ensuring that sufficient free space exists on the disk before a file is stored.

By default, RECSIZE() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

- The following user-defined function, DbfSize(), uses RECSIZE() to calculate the size of the current database file:

```
FUNCTION DbfSize
   RETURN ((RECSIZE() * LASTREC()) + HEADER() + 1)
```

- This example illustrates the use of RECSIZE() to determine the record length of database files open in unselected work areas:

```
USE Customer NEW
USE Sales NEW
//
? RECSIZE(), Customer->(RECSIZE())
? DbfSize(), Customer->(DbfSize())
```

Files Library is EXTEND.LIB.

See Also DISKSPACE(), FIELDNAME(), HEADER(), LASTREC()

REINDEX command

Rebuild open indexes in the current work area

Syntax

```
REINDEX
[EVAL <lCondition>]
[EVERY <nRecords>]
```

Arguments

EVAL <lCondition> specifies a condition that is evaluated either for each record processed or at the interval specified by the EVERY clause. This clause is identical to the EVAL clause of the INDEX command, but must be respecified in order for the reindexing operation to be monitored since the value of *<lCondition>* is transient.

EVERY <nRecords> specifies a numeric expression that modifies how often EVAL is evaluated. When using EVAL, the EVERY option offers a performance enhancement by evaluating the condition for every *nth* record instead of evaluating each record reindexed. The EVERY keyword is ignored if no EVAL condition is specified.

Description

REINDEX is a database command that rebuilds all open indexes in the current work area. When the reindexing operation finishes, all rebuilt indexes remain open, order is reset to one, and the record pointer is positioned to the first record in the controlling index. If any of the indexes were created with SET UNIQUE ON, REINDEX adds only unique keys to the index. If any of the indexes were created using a FOR condition, only those key values from records matching the condition are added to the index.

In a network environment, REINDEX requires EXCLUSIVE USE of the current database file. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Caution! *REINDEX does not recreate the header of the index file when it recreates the index. Because of this, REINDEX does not help if there is corruption of the file header. To guarantee a valid index, always use INDEX ON in place of REINDEX to rebuild damaged indexes*

Notes

Index key order, UNIQUE status, and the FOR condition are known to the index (.ntx) file and are, therefore, respected and maintained by REINDEX.

Examples

- This example REINDEXes the index open in the current work area:

```
USE Sales INDEX Salesman, Territory NEW
REINDEX
```

- This example REINDEXes using a progress indicator:

```
USE Sales INDEX Salesman, Territory NEW
REINDEX EVAL NtxProgress() EVERY 10

FUNCTION NtxProgress
LOCAL cComplete := LTRIM(STR((RECNO()/LASTREC()) * 100))
@ 23, 00 SAY "Indexing..." + cComplete + "%"
RETURN .T.
```

Files Library is CLIPPER.LIB.

See Also INDEX, PACK, SET INDEX, USE

RELEASE command

Delete public and private memory variables

Syntax

```
RELEASE <idMemvar list>
RELEASE ALL [LIKE | EXCEPT <skeleton>]
```

Arguments

<idMemvar list> is a list of private or public variables or arrays to delete.

ALL [LIKE | EXCEPT <skeleton>] defines the set of visible private memory variables to assign, or to exclude from assignment of, a NIL value. *<skeleton>* is the wildcard mask to specify a group of memory variables to delete. The wildcard characters supported are * and ?.

Description

RELEASE is a memory variable command that performs one of two actions depending on how it is specified. If RELEASE is specified with *<idMemvar list>*, the specified public and private memory variables and/or arrays are deleted from memory. Previous hidden instances (public or private variables defined in higher-level procedures) become accessible upon termination of the procedure where the variable was originally created.

If RELEASE is specified with any form of the ALL clause, private memory variables created at the current procedure level are assigned a NIL and not deleted until the current procedure or user-defined function terminates. Public variables are unaffected by this form of the RELEASE command. To release public variables, you must RELEASE them explicitly or use CLEAR MEMORY.

Local or static variables are not affected by the RELEASE command. Local variables are released automatically when the procedure or user-defined function (where the variables were declared) terminates. Static variables cannot be released since they exist for the duration of the program.

Files Library is CLIPPER.LIB.

See Also CLEAR MEMORY, LOCAL, PRIVATE, PUBLIC, QUIT

RENAME command

Change the name of a file

Syntax

```
RENAME <xcOldFile> TO <xcNewFile>
```

Arguments

<xcOldFile> is the name of the file to be renamed including an extension, and optionally is preceded by a drive and/or path designator. *<xcOldFile>* can be a literal string or a character expression enclosed in parentheses.

TO <xcNewFile> specifies the new file name including extension and optionally prefaced by a drive and/or path designator. *<xcNewFile>* can be a literal string or a character expression enclosed in parentheses.

Description

RENAME is a file command that changes the name of a specified file to a new name. If the source directory is different from the target directory, the file moves to the new directory. RENAME does not use SET DEFAULT and SET PATH to locate *<xcOldFile>*. Instead, the *<xcOldFile>* is renamed only if it is located in the current DOS directory or in the specified path.

In the instance that either *<xcNewFile>* exists or is currently open, RENAME does nothing. To trap this condition as an error, use the FILE() function before executing the command. See the example.

Warning! *Files must be CLOSEd before renaming. Attempting to rename an open file will produce unpredictable results. When a database file is RENAMEd, remember that any associated memo (.dbt) file must also be RENAMEd. Failure to do so may compromise the integrity of your program.*

Examples

- This example renames a file, checking for the existence of the target file before beginning the RENAME operation:

```
xcOldFile := "OldFile.txt"
xcNewFile := "NewFile.txt"
IF !FILE(xcNewFile)
   RENAME (xcOldFile) TO (xcNewFile)
   ELSE
? "File already exists"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also COPY FILE, CURDIR(), ERASE, FILE(), FERASE(), FRENAME(), RUN

REPLACE command

Assign new values to field variables

Syntax

```
REPLACE <idField> WITH <exp>
   [, <idField2> WITH <exp2>...]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

<idField> is the name of the field variable to be assigned a new value. If *<idField>* is prefaced with an alias, the assignment takes place in the designated work area.

WITH <exp> defines the value to assign to *<idField>*.

<scope> is the portion of the current database file to REPLACE. The default is the current record, or NEXT 1. Specifying a condition changes the default to ALL records in the current work area.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to REPLACE within the given scope.

Description

REPLACE is a database command that assigns new values to the contents of one or more field variables in the current record in the specified work areas. The target field variables can be character, date, logical, memo, or numeric. REPLACE performs the same function as the assignment operator (:=) except that it assumes that an unaliased reference is to a field variable. This means that you can assign new values to field variables using assignment statements provided that the field variable references are prefaced with an alias, the FIELD alias, or declared using the FIELD declaration statement.

The default scope of REPLACE is the current record unless a scope or condition is specified. If a scope or condition is specified, the replace operation is performed on each record matching the scope and/or condition.

Warning! *When you REPLACE a key field, the index is updated and the relative position of the record pointer within the index is changed. This means that REPLACEing a key field with a scope or a condition may yield an erroneous result. To update a key field, SET ORDER TO 0 before the REPLACE. This ensures that the record pointer moves sequentially in natural order. All open indexes, however, are updated if the key field is REPLACEd.*

In a network environment, REPLACEing the current record requires an RLOCK(). REPLACEing with a scope and/or condition requires an FLOCK() or EXCLUSIVE USE of the current database file. If a field is being REPLACEd in another work area by specifying its alias, that record must also be locked with an RLOCK(). Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Examples

- This example shows a simple use of REPLACE:

```
USE Customer NEW
APPEND BLANK
USE Invoices NEW
APPEND BLANK
//
REPLACE Charges WITH Customer->Markup * Cost,;
    Custid WITH Customer->Custid,;
    Customer->TranDate WITH DATE()
```

- This example uses assignment statements in place of the REPLACE command:

```
FIELD->Charges := Customer->Markup * FIELD->Cost
FIELD->Custid := Customer->Custid
Customer->TranDate := DATE()
```

Files Library is CLIPPER.LIB.

See Also COMMIT, FLOCK(), RLOCK()

REPLICATE() function

Return a string repeated a specified number of times

Syntax

```
REPLICATE(<cString>, <nCount>) → cRepeatedString
```

Arguments

<cString> is the character string to be repeated.

<nCount> is the number of times to repeat *<cString>*.

Returns

REPLICATE() returns a character string up to a maximum of 65,535 (64K) bytes in length. Specifying a zero as the *<nCount>* argument returns a null string ("").

Description

REPLICATE() is a character function that repeatedly displays, prints, or stuffs the keyboard with one or more characters. REPLICATE() is like the SPACE() function, which returns a specified number of space characters.

Examples

- These examples demonstrate REPLICATE() repeating strings:

```
? REPLICATE("*", 5)          // Result: *****
? REPLICATE("Hi ", 2)        // Result: Hi Hi
? REPLICATE(CHR(42), 5)      // Result: *****
```

- This example uses REPLICATE() to stuff the keyboard with several Down arrow keys:

```
#include "Inkey.ch"
KEYBOARD REPLICATE(CHR(K_DOWN), 25)
```

Files

Library is CLIPPER.LIB.

See Also

SPACE()

REPORT FORM command

Display a report to the console

Syntax

```
REPORT FORM <xcReport>
   [TO PRINTER] [TO FILE <xcFile>] [NOCONSOLE]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [PLAIN | HEADING <cHeading>] [NOEJECT] [SUMMARY]
```

Arguments

<xcReport> is the name of the report form (.frm) file that contains the definition of the REPORT. If an extension is not specified, (.frm) is assumed. *<xcReport>* can be specified as a literal string or as a character expression enclosed in parentheses.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output without form feed characters (ASCII 12) to a file. If a file extension is not specified, .txt is added. You can specify *<xcFile>* as a literal string or as a character expression enclosed in parentheses.

NOCONSOLE suppresses all REPORT FORM output to the console. If not specified, output automatically displays to the console unless SET CONSOLE is OFF.

<scope> is the portion of the current database file to report. The default scope is ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to report within the given scope.

PLAIN suppresses the display of the date and page number, and causes the report to print without page breaks. In addition, the report title and column headings display only at the top of the report.

HEADING places the result of *<cHeading>* on the first line of each page. *<cHeading>* is evaluated only once at the beginning of the report before the record pointer is moved. If both PLAIN and HEADING are specified, PLAIN takes precedence.

NOEJECT suppresses the initial page eject when the TO PRINTER clause is used.

SUMMARY causes REPORT FORM to display only group, subgroup, and grand total lines. Detail lines are suppressed.

Description

REPORT FORM is a console command that sequentially accesses records in the current work area and displays a tabular and optionally grouped report with page and column headings from a definition held in a .frm file. The actual REPORT FORM file (.frm) is created using RL.EXE or dBASE III PLUS. Refer to the "Report and Label Utility" chapter in the *Programming and Utilities Guide* for more information about creating report definitions.

When invoked, REPORT FORM sends output to the screen and, optionally, to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF or use the NOCONSOLE keyword before the REPORT FORM invocation.

When invoked, REPORT FORM searches the current SET PATH drive and directory if the *<xcReport>* file is not found in the current directory and the path is not specified.

Notes

- ***Interrupting REPORT FORM:*** To allow the user to interrupt a REPORT FORM, use INKEY() to test for an interrupt key press, as a part of the FOR condition. See the example below.
- ***Printer margin:*** REPORT FORM obeys the current SET MARGIN value for output echoed to the printer.
- ***Forcing formfeed characters into an output file:*** To include form feed characters when sending a REPORT FORM TO FILE, redirect printer output to a file using SET PRINTER like this:

```
SET PRINTER TO <xcFile>
REPORT FORM <xcReport> TO PRINTER
SET PRINTER TO
```

- ***Reporting in a network environment:*** REPORT FORM commands executed in a network environment can be affected by changes made to database files by other users while the report is in progress. For example, if a user changes a key value from "A" to "Z" while the report is printing, the same record could appear twice.

Examples

- This example uses both a literal and an extended expression to execute a REPORT FORM:

```
LOCAL xcReport := "Sales"
USE Sales INDEX Sales NEW
REPORT FORM Sales TO PRINTER FOR Branch = "100";
          HEADING "Branch 100"
REPORT FORM (xcReport) TO PRINTER FOR Branch != "100"
```

- This example interrupts a REPORT FORM using INKEY() to test whether the user has pressed the Esc key:

```
#define K_ESC  27
USE Sales INDEX Sales NEW
REPORT FORM Sales WHILE INKEY() != K_ESC
```

Files Library is CLIPPER.LIB.

See Also LABEL FORM, LIST

REQUEST statement

Declare a module request list

Syntax

```
REQUEST <idModule list>
```

Arguments

<idModule list> is the list of modules that will be linked into the current executable (.EXE) file.

Description

REQUEST is a declaration statement that defines a list of module identifiers to the linker. Like all other declaration statements, a REQUEST statement must be specified before any executable statements in either the program file, or a procedure or user-defined function definition.

During the compilation of CA-Clipper source code, all explicit references to procedures and user-defined functions are made to the linker. In some instances, within a source file, there may be no references made to procedure or user-defined function names until runtime. REQUEST resolves this situation by forcing the named procedures or user-defined functions to be linked even if they are not explicitly referenced in the source file. This is important in several instances:

- Procedures, user-defined functions, or formats referenced with macro expressions or variables
- Procedures and user-defined functions used in REPORT and LABEL FORMs and not referenced in the source code
- User-defined functions used in index keys and not referenced in the source code
- ACHOICE(), DBEDIT(), or MEMOEDIT() user functions
- Initialization procedures declared with the INIT PROCEDURE statement
- Exit procedures declared with the EXIT PROCEDURE statement

To group common REQUESTs together, place them in a header file and then include (#include) the header file into each program file (.prg) that might indirectly use them.

Examples

- This example shows a typical header file consisting of common REQUESTs for REPORT FORMs:

```
// Request.ch

REQUEST HARDCR
REQUEST TONE
REQUEST MEMOTRAN
REQUEST STRTRAN
```

See Also

ACHOICE(), ANNOUNCE, DBEDIT(), EXIT PROCEDURE, EXTERNAL, INIT PROCEDURE, HARDCR(), MEMOTRAN(), STRTRAN(), TONE()

RESTORE command

Retrieve memory variables from a memory (.mem) file

Syntax

```
RESTORE FROM <xcMemFile> [ADDITIVE]
```

Arguments

<xcMemFile> is the memory (.mem) file to load from disk. If an extension is not specified, the extension .mem is assumed. The file name may be specified as a literal string or as a character expression enclosed in parentheses.

ADDITIVE causes memory variables loaded from the memory file to be added to the existing pool of memory variables.

Description

RESTORE is a memory variable command that recreates public and private variables previously SAVEd to a memory (.mem) file and initializes them with their former values. The scope of the variable is not SAVEd with the variable, but is instead established when the variable is RESTOREd. Arrays and local variables cannot be SAVEd or RESTOREd.

When memory variables are RESTOREd, they are recreated as private variables with the scope of the current procedure or user-defined function unless they exist as public variables and you specify the ADDITIVE clause. If ADDITIVE is specified, public and private variables with the same names are overwritten unless hidden with PRIVATE. If ADDITIVE is not specified, all public and private variables are released before the memory file is loaded.

Local and static variables are unaffected by RESTORE. If a local or static variable has been declared in the current procedure or user-defined function and a variable with the same name is RESTOREd, only the local or static variable is visible unless references to the RESTOREd variable are prefaced with the MEMVAR alias.

Examples

- This example demonstrates a typical application of SAVE and RESTORE. Here memory variables containing screens are SAVEd TO and RESTOREd FROM memory files:

```
// Create and use a pseudoarray of screens
SAVE SCREEN TO cScreen1
SAVE ALL LIKE cScreen* TO Screens
//
<statements>...
//
RESTORE FROM Screens ADDITIVE
nNumber = "1"
RESTORE SCREEN FROM ("cScreen" + nNumber)
```

Files Library is CLIPPER.LIB.

See Also LOCAL, PRIVATE, PUBLIC, SAVE

RESTORE SCREEN* command

Display a saved screen

Syntax

```
RESTORE SCREEN [FROM <cScreen>]
```

Arguments

FROM <cScreen> specifies a character expression to display to the screen.

Description

RESTORE SCREEN is a command synonym for the RESTSCREEN() function that redisplays a previously saved screen, and is used with SAVE SCREEN to avoid repainting the original screen painted with @...SAY, @...GET, and ? commands.

RESTORE SCREEN works in two ways depending on whether or not you specify the FROM clause. If you specify the FROM clause, the SCREEN is RESTOREd FROM *<cScreen>*. *<cScreen>* is a character expression, usually a variable assigned a screen image by SAVE SCREEN. If you do not specify the FROM clause, the SCREEN is RESTOREd from the default save screen buffer created by SAVE SCREEN specified without the TO clause.

SAVESCREEN() and RESTORESCREEN() functions supersede SAVE SCREEN and RESTORE SCREEN commands.

RESTORE SCREEN is a compatibility command and, therefore, not recommended.

Warning! *SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.*

Examples

- This example displays a small alert pop-up box using SAVE and RESTORE SCREEN:

```
IF FileAlert()
   COPY FILE Them.txt TO My.txt
ELSE
   BREAK
ENDIF
RETURN

FUNCTION FileAlert
   LOCAL lAnswer := .F., cScreen
   SAVE SCREEN TO cScreen
   @ 10, 10 CLEAR TO 12, 45
   @ 10, 10 TO 12, 45 DOUBLE
   @ 11, 12 SAY "File exists, overwrite? (y/n) ";
            GET lAnswer PICTURE "Y"
   READ
   RESTORE SCREEN FROM cScreen
   RETURN lAnswer
```

Files Library is CLIPPER.LIB.

See Also RESTORE, RESTSCREEN(), SAVE, SAVESCREEN()

RESTSCREEN() function

Display a saved screen region to a specified location

Syntax

```
RESTSCREEN([<nTop>], [<nLeft>],
   [<nBottom>], [<nRight>], <cScreen>) → NIL
```

Arguments

<nTop>, ***<nLeft>***, ***<nBottom>***, and ***<nRight>*** define the coordinates of the screen information contained in *<cScreen>*. If *<cScreen>* was saved without coordinates to preserve the entire screen, no screen coordinates are necessary with RESTSCREEN().

<cScreen> is a character string containing the saved screen region.

Returns

RESTSCREEN() always returns NIL.

Description

RESTSCREEN() is a screen function that redisplays a screen region saved with SAVESCREEN(). The target screen location may be the same as or different from the original location when the screen region was saved. If you specify a new screen location, the new screen region must be the same size or you will get ambiguous results. To use RESTSCREEN() to restore screen regions saved with SAVE SCREEN, specify the region coordinates as 0, 0, MAXROW(), MAXCOL().

Warning! *SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.*

Examples

- This example demonstrates RESTSCREEN() as part of a general purpose pop-up menu function, PopMenu():

```
? PopMenu({1, 1, 3, 10, {"ItemOne", "ItemTwo"}, ;
      "BG+/B"})

FUNCTION PopMenu( aList )
   LOCAL cScreen, nChoice, cOldColor := ;
            SETCOLOR(aList[6])
   cScreen := SAVESCREEN(aList[1], aList[2],;
      aList[3], aList[4])
   @ aList[1], aList[2], TO aList[3], aList[4] DOUBLE
   nChoice := ACHOICE(++aList[1], ++aList[2],;
      --aList[3], --aList[4], aList[5])
   SETCOLOR(cOldColor)
   RESTSCREEN(--aList[1], --aList[2], ++aList[3],;
      ++aList[4], cScreen)
   RETURN nChoice
```

Files Library is EXTEND.LIB.

See Also RESTORE, RESTORE SCREEN, SAVE, SAVESCREEN()

RETURN statement

Terminate a procedure, user-defined function, or program

Syntax

```
RETURN [<exp>]
```

Arguments

<exp> is an expression of any type that evaluates to the return value for user-defined functions. If a user-defined function terminates without executing a RETURN statement, the return value is NIL.

Description

RETURN terminates a procedure, user-defined function, or program by returning control to either the calling procedure or user-defined function. When RETURN executes in the highest level procedure, control passes to the operating system. All private variables created and local variables declared in the current procedure or user-defined function are released when control returns to the calling procedure.

There can be more than one RETURN in a procedure or user-defined function. A procedure or user-defined function need not, however, end with a RETURN. Since user-defined functions must return values, each must contain at least one RETURN statement with an argument.

Note: A procedure or user-defined function definition is terminated by a PROCEDURE statement, a FUNCTION statement, or end of file but not by a RETURN statement.

Notes

- ***Arrays:*** Since array is a data type like any other data type, instances of array type are really values like character strings and, therefore, can be RETURNed from a user-defined function.
- ***RETURN TO MASTER:*** CA-Clipper does not support RETURN TO MASTER or any other form of RETURN specifying the level to which the call is to return. You can, however, simulate these operations with BEGIN SEQUENCE...END.

Examples

- These examples illustrate the general form of the RETURN statement in a procedure and in a user-defined function:

```
PROCEDURE <idProcedure>
    //
    <statements>...
    //
    RETURN

FUNCTION <idFunction>
    //
    <statements>...
    //
    RETURN <expReturn>
```

- This example returns an array, created in a user-defined function, to a calling procedure or user-defined function:

```
FUNCTION PassArrayBack
    PRIVATE aArray[10][10]
    aArray[1][1] = "myString"
    RETURN aArray
```

See Also BEGIN SEQUENCE, FUNCTION, LOCAL, PRIVATE, PROCEDURE, PUBLIC, QUIT

RIGHT() function

Return a substring beginning with the rightmost character

Syntax

RIGHT(<*cString*>, <*nCount*>) → *cSubString*

Arguments

<*cString*> is the character string from which to extract characters.

<*nCount*> is the number of characters to extract.

Returns

RIGHT() returns the rightmost <*nCount*> characters of <*cString*>. If <*nCount*> is zero, RIGHT() returns a null string (""). If <*nCount*> is negative or larger than the length of the character string, RIGHT() returns <*cString*>. The maximum string size is 65,535 (64K) bytes.

Description

RIGHT() is a character function that extracts a substring beginning with the rightmost character in <*cString*>. It is the same as the character expression, SUBSTR(<*cString*>, <*nCount*>). For example, RIGHT("ABC", 1) is the same as SUBSTR("ABC", -1). RIGHT() is related to LEFT(), which extracts a substring beginning with the leftmost character in <*cString*>.

The RIGHT(), LEFT(), and SUBSTR() functions are often used with both the AT() and RAT() functions to locate either the first and/or the last position of a substring before extracting it.

Examples

- This example shows the relationship between RIGHT() and SUBSTR():

```
? RIGHT("ABCDEF", 3)              // Result: DEF
? SUBSTR("ABCDEF", -3)            // Result: DEF
```

- This example extracts a substring from the end of another string up to the last occurrence of a comma:

```
LOCAL cName := "James,William"
? RIGHT(cName,;
LEN(cName) - RAT(",", cName) - 1)      // Result: William
```

Files

Library is EXTEND.LIB.

See Also

LEFT(), LTRIM(), RTRIM(), STUFF(), SUBSTR()

RLOCK() function

Lock the current record in the active work area

Syntax

```
RLOCK() → lSuccess
```

Returns

RLOCK() returns true (.T.) if the record lock is obtained; otherwise, it returns false (.F.).

Description

RLOCK() is a network function that locks the current record, preventing other users from updating the record until the lock is released. RLOCK() provides a shared lock, allowing other users read-only access to the locked record while allowing only the current user to modify it. A record lock remains until another record is locked, an UNLOCK is executed, the current database file is closed, or an FLOCK() is obtained on the current database file.

For each invocation of RLOCK(), there is one attempt to lock the current record, and the result is returned as a logical value. An attempt to obtain a record lock fails if another user currently has a file or record lock on that particular record, or EXCLUSIVE USE of the database file. An attempt to RLOCK() in an empty database returns true (.T.).

By default, RLOCK() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below). This feature is useful since RLOCK() does not automatically attempt a record lock for related files.

As a general rule, RLOCK() operates solely on the current record. This includes the following commands:

- @...GET
- DELETE (single record)
- RECALL (single record)
- REPLACE (single record)

Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Notes

- ***SET RELATION:*** CA-Clipper does not automatically lock all records in the relation chain when you lock the current work area record. Also, an UNLOCK has no effect on related work areas.

Examples

- This example deletes a record in a network environment, using RLOCK():

```
USE Customer INDEX CustName SHARED NEW
SEEK "Smith"
IF FOUND()
    IF RLOCK()
       DELETE
       ? "Smith deleted"
    ELSE
       ? "Record in use by another"
    ENDIF
ELSE
    ? "Smith not in Customer file"
ENDIF
CLOSE
```

- This example specifies RLOCK() as an aliased expression to lock a record in an unselected work area:

```
USE Sales SHARED NEW
USE Customer SHARED NEW
//
IF !Sales->(RLOCK())
    ? "The current Sales record is in use by another"
ENDIF
```

Files Library is CLIPPER.LIB.

See Also APPEND BLANK, FLOCK(), SET EXCLUSIVE*, UNLOCK, USE

ROUND() function

Return a numeric value rounded to a specified number of digits

Syntax

```
ROUND(<nNumber>, <nDecimals>) → nRounded
```

Arguments

<nNumber> is the numeric value to be rounded.

<nDecimals> defines the number of decimal places to retain. Specifying a negative *<nDecimals>* value rounds whole number digits.

Returns

ROUND() returns a numeric value.

Description

ROUND() is a numeric function that rounds *<nNumber>* to the number of places specified by *<nDecimals>*. Specifying a zero or negative value for *<nDecimals>* allows rounding of whole numbers. A negative *<nDecimals>* indicates the number of digits to the left of the decimal point to round. Digits between five to nine (inclusive) are rounded up. Digits below five are rounded down.

The display of the return value does not obey the DECIMALS setting unless SET FIXED is ON. With SET FIXED OFF, the display of the return value contains as many decimal digits as you specify for *<nDecimals>*, or zero, if *<nDecimals>* is less than one.

Examples

- These examples round values with decimal digits:

```
SET DECIMALS TO 2
SET FIXED ON
//
? ROUND(10.4, 0)                    // Result: 10.00
? ROUND(10.5, 0)                    // Result: 11.00
? ROUND(10.51, 0)                   // Result: 11.00
? ROUND(10.49999999999999, 2)       // Result: 10.50
```

- These examples use a negative <*nDecimals*> argument to round numeric values to whole number values:

```
? ROUND(101.99, -1)                 // Result: 100.00
? ROUND(109.99, -1)                 // Result: 110.00
? ROUND(109.99, -2)                 // Result: 100.00
```

Files Library is CLIPPER.LIB.

See Also INT(), SET DECIMALS, SET FIXED, STR(), VAL()

ROW() function

Return the screen row position of the cursor

Syntax

```
ROW() → nRow
```

Returns

ROW() returns the cursor row position as an integer numeric value. The range of the return value is zero to MAXROW().

Description

ROW() is a screen function that returns the current row or line position of the screen cursor. The value of ROW() is updated by both console and full-screen commands and functions. @...SAY only updates ROW() when the current DEVICE is the SCREEN.

ROW() is used with COL() and all variations of the @ command to position the cursor to a new line relative to the current line. In particular, you can use ROW() and COL() to create screen position-independent procedures or functions where you pass the upper-left row and column as parameters.

ROW() is related to PROW() and PCOL(), which track the current printhead position instead of the screen cursor position.

Examples

- In this example, ROW() simulates the LIST command, displaying text on the same line but in different columns:

```
LOCAL nRow
USE Customer INDEX CustName NEW
DO WHILE .NOT. EOF()
   CLS
   @ 1, 1 SAY PADR("Name", LEN(CustName))
   @ ROW(), COL() + 2 SAY PADR("Address", ;
         LEN(Address))
   @ ROW(), COL() + 2 SAY PADR("Phone", LEN(Phone))
   nRow = 0
   DO WHILE nRow++ <= 15 .AND. (!EOF())
      @ ROW() + 1, 1 SAY CustName
      @ ROW(), COL() + 2 SAY Address
      @ ROW(), COL() + 2 SAY Phone
      SKIP
   ENDDO
   WAIT
ENDDO
CLOSE Customer
```

Files Library is CLIPPER.LIB.

See Also ?|??, @...GET, @...SAY, COL(), MAXROW(), PCOL(), PROW(), SET DEVICE

RTRIM() function

Remove trailing spaces from a character string

Syntax

```
RTRIM(<cString>) → cTrimString
```

Arguments

<cString> is the character string to be copied without trailing spaces.

Returns

RTRIM() returns a copy of *<cString>* with the trailing spaces removed. If *<cString>* is a null string ("") or all spaces, RTRIM() returns a null string ("").

Description

RTRIM() is a character function that formats character strings. It is useful when you want to delete trailing spaces while concatenating strings. This is typically the case with database fields which are stored in fixed-width format. For example, you can use RTRIM() to concatenate first and last name fields to form a name string.

RTRIM() is related to LTRIM() which removes leading spaces, and ALLTRIM() which removes both leading and trailing spaces. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADR(), and PADL() functions which center, right-justify, or left-justify character strings by padding them with fill characters. RTRIM() is exactly the same as TRIM() in function.

Notes

- ***Space characters:*** The RTRIM() function treats carriage returns, line feeds, and tabs as space characters and removes these as well.

Examples

- This is a user-defined function in which RTRIM() formats city, state, and zip code fields for labels or form letters:

```
FUNCTION CityState(cCity, cState, cZip)
    RETURN RTRIM(cCity) + ", " ;
     + RTRIM(cState) + "  " + cZip
```

- In this example the user-defined function, CityState(), displays a record from Customer.dbf:

```
USE Customer INDEX CustName NEW
SEEK "Kate"
? CityState(City, State, ZipCode)
// Result: Athens, GA 10066
```

Files Library is CLIPPER.LIB.

See Also ALLTRIM(), LTRIM(), PAD(), SUBSTR(), TRIM()

RUN command

Execute a DOS command or program

Syntax

```
RUN | !* <xcCommandLine>
```

Arguments

<xcCommandLine> is any executable program including resident DOS commands and COMMAND.COM. It may be specified either as a literal string or as a character expression enclosed in parentheses.

Description

RUN executes a DOS command or program from within a compiled application. When you RUN a DOS program, CA-Clipper executes another copy of COMMAND.COM, passing the DOS command line at the same time. This has two implications. First, you must have enough memory for COMMAND.COM (5K for DOS 6.2) and the program you wish to execute. Second, COMMAND.COM must be available on the path specified by COMSPEC (the default is the root directory of the disk where you boot DOS). If COMMAND.COM is not located on this disk or the disk is changed, SET COMSPEC to the new location prior to running the CA-Clipper application. Note that SET DEFAULT and SET PATH have no effect on RUN.

The ! form of the RUN command is provided for compatibility purposes only and, therefore, is not recommended.

Warning! *Do not RUN memory-resident programs from within CA-Clipper since you may lose memory when the control returns to your application program.*

Examples

- This example uses RUN with MEMOREAD() and MEMOWRIT() to create a user-defined function that calls a text editor with the current memo field:

```
lSuccess = EditorMemo("Qedit", "Notes")
RETURN

FUNCTION EditorMemo( cEditor, cMemofld )
    IF MEMOWRIT("Clipedit.tmp", &cMemofld.)
       RUN (cEditor + " Clipedit.tmp")
       REPLACE &cMemofld. WITH MEMOREAD("Clipedit.tmp")
       ERASE Clipedit.tmp
       RETURN .T.
    ELSE
       RETURN .F.
    ENDIF
```

- One of the options you may want to give your users is direct access to DOS. Do this with:

```
RUN COMMAND
```

  To make it easier for the user to return to the application program, change the DOS prompt in the application batch file like this:

```
REM Application Batch File
ECHO OFF
PROMPT DOS Access: Type EXIT to return to ;
         application$_$p$g
<your application program>
PROMPT $p$g
```

  Then, instruct the user to execute the application batch file in place of the application .EXE file.

Files

Library is CLIPPER.LIB.

SAVE command

Save variables to a memory (.mem) file

Syntax

```
SAVE TO <xcMemFile> [ALL [LIKE | EXCEPT <skeleton>]]
```

Arguments

<xcMemFile> is the memory (.mem) file to SAVE to disk. You may specify the file name as a literal string or as a character expression enclosed in parentheses. If you specify no extension, the file is created with a .mem extension.

ALL [LIKE | EXCEPT <skeleton>] defines the set of visible private and public memory variables to save to *<xcMemFile>*. *<skeleton>* is the wildcard mask that characterizes a group of memory variables to SAVE. The wildcard characters supported are * and ?.

Description

SAVE copies public and private memory variables visible within the current procedure or user-defined function to a memory (.mem) file. Arrays and local and static variables, however, cannot be SAVEd. When variables are SAVEd, they are copied without any reference to scope. Variables hidden by PRIVATE or LOCAL declarations are not SAVEd.

If you specify the ALL LIKE clause, variable names matching the *<skeleton>* mask are saved. By contrast, if you specify ALL EXCEPT, variable names not matching the *<skeleton>* are saved.

You can specify a *<skeleton>* that includes wildcard characters. The * wildcard character matches any group of adjacent characters ending a variable name and can be specified only at the end of the *<skeleton>*. The ? wildcard character matches any single character and can be specified anywhere within the *<skeleton>*.

Examples

- This example saves all visible private and public variables to Temp.mem:

```
PRIVATE cOne := "1"
SAVE ALL TO Temp
```

- This example saves all visible private and public variables with names beginning with "c" to Myvars.mem:

```
SAVE ALL LIKE c* TO MyVars
```

- This example saves all visible private and public variables with names that do not begin with "c" to Myvars2.mem:

```
SAVE ALL EXCEPT c* TO MyVars2
```

Files Library is CLIPPER.LIB.

See Also LOCAL, PRIVATE, PUBLIC, RESTORE

SAVE SCREEN* command

Save the current screen to a buffer or variable

Syntax

```
SAVE SCREEN [TO <idVar>]
```

Arguments

TO <idVar> specifies a variable to contain the current screen contents as a character value. If *<idVar>* is not visible or does not exist, a private memory variable is created and assigned to the screen.

Description

SAVE SCREEN is a command synonym for the SAVESCREEN() function that saves the screen from 0, 0 to MAXROW(), MAXCOL() in a default screen buffer, or in an optional variable. If the screen is saved to a variable, the variable can be any storage class including field, local, static, or an array element. Note, however, you cannot SAVE an array or local or static variable to .mem files to save multiple screens to disk.

SAVE SCREEN is used with RESTORE SCREEN to eliminate repainting an original screen that has been temporarily replaced. You may save multiple screens by assigning each screen to a separate variable.

SAVE SCREEN is a compatibility command and not recommended. It is superseded by the SAVESCREEN() function which can save partial or full screens.

Warning! *SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.*

Examples

- This code skeleton uses a static array to store saved screens:

```
STATIC aScreens[10]
SAVE SCREEN TO aScreens[1]
//
<statements>...
//
RESTORE SCREEN FROM aScreens[1]
```

- This example saves and restores screens using a database file:

```
USE Screens INDEX Name NEW
APPEND BLANK
Screens->Name := "Screen001"        // Save the screen name
SAVE SCREEN TO Screens->Image       // Save a new screen image
//
<statements>...
//
SEEK "Screen001"                       // Find the screen
RESTORE SCREEN FROM Screens->Image     // Restore it
```

Files Library is CLIPPER.LIB.

See Also MAXROW(), MAXCOL(), RESTORE, RESTSCREEN(), SAVE, SAVESCREEN()

SAVESCREEN() function

Save a screen region for later display

Syntax

```
SAVESCREEN([<nTop>], [<nLeft>],
   [<nBottom>], [<nRight>]) → cScreen
```

Arguments

<nTop>, ***<nLeft>***, ***<nBottom>***, and ***<nRight>*** define the coordinates of the screen region to be saved. If either *<nBottom>* or *<nRight>* is greater than MAXROW() or MAXCOL(), the screen is clipped. If you specify no coordinates, the entire screen (i.e., from 0,0 to MAXROW(), MAXCOL()) is saved.

Returns

SAVESCREEN() returns the specified screen region as a character string.

Description

SAVESCREEN() is a screen function that saves a screen region to a variable of any storage class including a field variable. Later, you can redisplay the saved screen image to the same or a new location using RESTSCREEN(). Screen regions are usually saved and restored when using a pop-up menu routine or dragging a screen object.

Warning! *SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.*

Examples

- The following user-defined function creates a pop-up menu using ACHOICE() with SAVESCREEN() and RESTSCREEN(), returning the selection in the array of choices:

```
FUNCTION PopMenu( nTop, nLeft, nBottom, nRight, ;
                  aItems, cColor )
   LOCAL cScreen, nChoice, cLastColor := ;
         SETCOLOR(cColor)
   //
   cScreen:= SAVESCREEN(nTop, nLeft, nBottom, nRight)
   @ nTop, nLeft TO nBottom, nRight DOUBLE
   //
   nChoice:= ACHOICE(++nTop, ++nLeft, ;
               --nBottom, --nRight, aItems)
   //
   RESTSCREEN(--nTop, --nLeft, ++nBottom, ++nRight, ;
                cScreen)
   SETCOLOR(cLastColor)
   RETURN nChoice
```

Files

Library is EXTEND.LIB.

See Also

ACHOICE(), MAXCOL(), MAXROW(), RESTORE SCREEN(), RESTSCREEN(), SAVE SCREEN(), SAVESCREEN

SCROLL() function

Scroll a screen region up or down, right or left

Syntax

```
SCROLL([<nTop>], [<nLeft>],[<nBottom>], [<nRight>],
   [<nVert>] [<nHoriz>]) → NIL
```

Arguments

<nTop>, ***<nLeft>***, ***<nBottom>***, and ***<nRight>*** define the scroll region coordinates. Row and column values can range from 0, 0 to MAXROW(), MAXCOL(). If you do not specify coordinate arguments, the dimensions of the visible display are used.

<nVert> defines the number of rows to scroll vertically. A positive value scrolls up the specified number of rows. A negative value scrolls down the specified number of rows. A value of zero disables vertical scrolling. If *<nVert>* is not specified, zero is assumed.

<nHoriz> defines the number of rows to scroll horizontally. A positive value scrolls left the specified number of columns. A negative value scrolls right the specified number of columns. A value of zero disables horizontal scrolling. If *<nHoriz>* is not specified, zero is assumed.

If you supply neither the *<nVert>* nor *<nHoriz>* parameters to SCROLL(), the area specified by the first four parameters will be blanked.

Warning! *Horizontal scrolling is not supported on all of the alternate terminal drivers (i.e., ANSITERM, NOVTERM, PCBIOS).*

Returns

SCROLL() always returns NIL.

Description

SCROLL() is a screen function that scrolls a screen region up or down a specified number of rows. When a screen scrolls up, the first line of the region is erased, all other lines are moved up, and a blank line is displayed in the current standard color on the bottom line of the specified region. If the region scrolls down, the operation is reversed. If the screen region is scrolled more than one line, this process is repeated.

SCROLL() is used primarily to display status information into a defined screen region. Each time a new message is displayed, the screen region scrolls up one line and a new line displays at the bottom.

Examples

- This user-defined function displays a message string at the bottom of a screen region after scrolling the region up one line:

```
FUNCTION ScrollUp( nTop, nLeft, nBottom, nRight, ;
                   expDisplay )
   //
   SCROLL(nTop, nLeft, nBottom, nRight, 1)
   @ nBottom, nLeft SAY expDisplay
   //
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also @...BOX, @...CLEAR, @...TO, CLEAR SCREEN, MAXCOL(), MAXROW()

Scrollbar class

Creates a scroll bar

Description

Scroll bars allow users to view data that exists beyond the limit of a window. By scrolling up, down, left, or right, you can reveal previously hidden pieces of data. Applications should provide scroll bars for any screen in which the data displayed is larger than the current window.

Class Function

```
Scrollbar <nStart>, <nEnd>, <nOffset>, [<bSBlock>]
   [,<nOrient>] ) → oScrollBar
```

Arguments

<nStart> is a numeric value that indicates the screen position where the scroll bar begins. *<nStart>* refers to the topmost row of a vertically oriented scroll bar, or the leftmost column of a horizontally oriented scroll bar.

<nEnd> is a numeric value that indicates the screen position where the scroll bar ends. *<nEnd>* refers to the bottommost row of a vertically oriented scroll bar, or the rightmost column of a horizontally oriented scroll bar.

<nOffset> is a numeric value that indicates the screen column of a vertically oriented scroll bar or the screen row of a horizontally oriented scroll bar.

<bSBlock> is an optional code block that, when present, is evaluated immediately before and after the ScrollBar object's state changes.

<nOrient> is an optional numeric value that indicates whether the scroll bar is vertically or horizontally oriented. The default is a vertically oriented scroll bar.

Returns

Returns a ScrollBar object when all of the required arguments are present; otherwise, Scrollbar() returns NIL.

Exported Instance Variables

`barLength`

Contains a numeric value that indicates the number of character positions that the scroll bar client area occupies. Each position refers to one row for a vertically oriented scroll bar or one column for a horizontally oriented scroll bar. The client area is the area between (but not including) the scroll bar's previous arrow and its next arrow.

`bitmaps` `(Assignable)`

Contains an array of exactly three elements. The first element of this array is the file name of the bitmap to be displayed at the top of a vertical scroll bar or to the left of a horizontal scroll bar. The second element of this array is the file name of the bitmap to be displayed at the bottom of a vertical scroll bar or to the right of a horizontal scroll bar. The third element of this array is the file name of the bitmap to be used as the thumbwheel.

Drive and directory names are not allowed; the file name extension is required. A bitmap file can be stored as a file on disk or in a bitmap library. If stored as a file, the file must reside in the same directory as the application. If stored in a bitmap library, the library must reside in the same directory as the application and it also must have the same name as the application with a .BML extension.

CA-Clipper will search for the file name first and then, if it is not found, it will search in the bitmap library. If no file is found either on disk or in the library, no bitmap will be displayed.

If this instance variable is not used, and the application is running in graphic mode, the files ARROW_L.BMU, ARROW_R.BMU, and ARROW_E.BMU will be used for a horizontal scroll bar and the files ARROW_U.BMU, ARROW_D.BMU, and ARROW_E.BMU will be used for a horizontal scroll bar.

This instance variable only affects applications running in graphic mode and is ignored in text mode.

`cargo` `(Assignable)`

Contains a value of any type that is ignored by the ScrollBar object. ScrollBar:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a ScrollBar object and retrieved later.

`colorSpec` `(Assignable)`

Contains a character string that indicates the color attributes that are used by the scroll bar's display() method. The string must contain two color specifiers.

Note: In graphic mode, colorSpec position 2 has no affect and is ignored.

ScrollBar Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The scroll bar's client area.	Unselected
2	The previous arrow, next arrow and thumb.	Enhanced

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Workbench Properties window for that item.

`current` `(Assignable)`

Contains a numeric value that indicates the number of the current item to which the scroll bar refers.

`end` `(Assignable)`

Contains a numeric value that indicates the screen position of the scroll bar's next arrow. ScrollBar:end refers to the bottommost row of a vertically oriented scroll bar, or the rightmost column of a horizontally oriented scroll bar.

`offset` `(Assignable)`

Contains a numeric value that indicates the screen column of a vertically oriented scroll bar or the screen row of a horizontally oriented scroll bar.

`orient` (Assignable)

Contains a numeric value that indicates whether the scroll bar is vertically or horizontally oriented.

Scroll Bar Types

Value	Constant	Description
1	SCROLL_VERTICAL	Vertical scroll bar
2	SCROLL_HORIZONTAL	Horizontal scroll bar

Button.ch contains manifest constants for the ScrollBar:orient value.

`sBlock` (Assignable)

Contains an optional code block that, when present, is evaluated immediately after the ScrollBar object's state changes. The code block takes no implicit arguments.

This code block is included in the ScrollBar class to provide a method of indicating when a state change event has occurred. The name "sBlock" refers to state block.

`start` (Assignable)

Contains a numeric value that indicates the screen position of the scroll bar's previous arrow. ScrollBar:start refers to the topmost row of a vertically oriented scroll bar, or the leftmost column of a horizontally oriented scroll bar.

`style` (Assignable)

Contains a character string that indicates the characters that are used by the scroll bar's display() method. The string must contain four characters. The first character is the previous arrow character. Its default value is the up arrow (↑) character for a vertically oriented scroll bar and the left arrow (←) character for a horizontally oriented scroll bar. The second character is the client area character. Its default value is the ▒ character. The third is the thumb character. Its default is the ■ character. The fourth character is the next arrow character. Its default value is the down arrow (↓) character for a vertically oriented scroll bar or the right arrow (→) character for a horizontally oriented scroll bar.

Note: In graphic mode, the style instance variable is ignored.

`thumbPos`

Contains a numeric value that indicates the relative screen position of the thumb within a scroll bar. Valid ScrollBar:thumbPos values range from 1 to ScrollBar:barlength.

`total` `(Assignable)`

Contains a numeric value that indicates the total number of items to which the scroll bar refers.

Exported Methods

`<oScrollBar>:display()` → *self*

display() is a method of the ScrollBar class that is used for showing a scroll bar including its thumb on the screen. display() uses the values of the following instance variables to correctly show the scroll bar in its current context, in addition to providing maximum flexibility in the manner a scroll bar appears on the screen: colorSpec, end, offset, orient, start, style, and thumbPos.

`<oScrollBar>:update()` → *self*

update() is a method of the ScrollBar class that is used for changing the thumb position on the screen. update() uses the values of the following instance variables to correctly show the thumb in its current context in addition to providing maximum flexibility in the manner a scroll bar thumb appears on the screen: colorSpec, offset, orient, stable, start, style, and thumbPos.

`<oScrollBar>:hitTest( <nMouseRow>, <nMouseCol> )`
→ *nHitStatus*

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the action to perform based on the logical position of the mouse cursor in the scroll bar's region of the screen.

Scroll Bar hittest Return Values

Value	Constant	Mouse Cursor Position
0	HTNOWHERE	Not within the scroll bar's region of the screen
-3073	HTSCROLLTHUMBDRAG	On the thumb
-3074	HTSCROLLUNITDEC	On the previous arrow
-3075	HTSCROLLUNITINC	On the next arrow
-3076	HTSCROLLBLOCKDEC	Between the previous arrow and the thumb
-3077	HTSCROLLBLOCKINC	Between the thumb and the next arrow

Button.ch contains manifest constants for the ScrollBar:hitTest return value.

hitTest() is a method of the ScrollBar class that is used for determining if the mouse cursor is within the region of the screen that the scroll bar occupies.

Examples

- This example creates a vertical scroll bar from row 2 to 10:

```
oScr:=(2,10,0)
```

SECONDS() function

Return the number of seconds elapsed since midnight

Syntax

SECONDS() → *nSeconds*

Returns

SECONDS() returns the system time as a numeric value in the form *seconds.hundredths*. The numeric value returned is the number of seconds elapsed since midnight, and is based on a twenty-four hour clock in a range from 0 to 86399.

Description

SECONDS() is a time function that provides a simple method of calculating elapsed time during program execution, based on the system clock. It is related to the TIME() function which returns the system time as a string in the form *hh:mm:ss*.

Examples

- This example contrasts the value of TIME() with SECONDS():

```
? TIME()              // Result: 10:00:00
? SECONDS()           // Result: 36000.00
```

- This example uses SECONDS() to track elapsed time in seconds:

```
LOCAL nStart, nElapsed
nStart:= SECONDS()
.
. <statements>
.
nElapsed:= SECONDS() - nStart
? "Elapsed: " + LTRIM(STR(nElapsed)) + " seconds"
```

Files

Library is CLIPPER.LIB.

See Also

TIME()

SEEK command

Search an order for a specified key value

Syntax

```
SEEK <expSearch> [SOFTSEEK]
```

Arguments

<expSearch> is an expression to match with an order key value.

SOFTSEEK causes the record pointer to be moved to the next record with a higher key value after a failed order search. Default behavior moves the record pointer to EOF() after a failed order search.

Description

SEEK is a database command that searches the controlling order from the first or last key value (depending on whether the LAST keyword is specified) and proceeds until a match is found or there is a key value greater than *<expSearch>*. If there is a match, the record pointer is positioned to the identity found in the order. If SOFTSEEK is OFF (the default) and SEEK does not find a match, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

SOFTSEEK enables a method of searching an order and returning a record even if there is no match for a specified key.

When SOFTSEEK is ON and a match for a SEEK is not found, the record pointer is set to the next record in the order with a higher key value than the SEEK argument. Records are not visible because SET FILTER and/or SET DELETED are skipped when searching for the next higher key value. If there is no record with a higher key value, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). FOUND() returns true (.T.) only if the record is actually found. FOUND() never returns true (.T.) for a relative find.

When SOFTSEEK is OFF and a SEEK is unsuccessful, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

SEEK with the SOFTSEEK clause is, effectively, the same as performing SET SOFTSEEK and then SEEK in earlier versions of CA-Clipper except that it does not change the global setting of SOFTSEEK.

Examples

- The following example searches for "Doe" using the SEEK command:

```
USE Customer NEW
SET ORDER TO Customer
? SET( _SET_SOFTSEEK )      // (.F.)
SEEK "Doe"
? SET( _SET_SOFTSEEK )      // Still (.F.)
IF FOUND()
    .
    . < statements >
    .
ENDIF
```

- The following example performs a soft seek for "Doe" using SOFTSEEK clause of the SEEK command:

```
USE Customer NEW
SET ORDER TO Customer
? SET( _SET_SOFTSEEK )      // (.F.)
SEEK "Doe" SOFTSEEK
? SET( _SET_SOFTSEEK )      // Still (.F.)
IF !FOUND()
   ? Customer->Name     // Returns next logical name after "Doe"
ENDIF
```

See Also

DBSEEK(), DBSETINDEX(), DBSETORDER(), EOF(), SET INDEX, SET ORDER

SELECT command

Change the current work area

Syntax

```
SELECT <xnWorkArea> | <idAlias>
```

Arguments

<xnWorkArea> is the work area number between 0 and 250 inclusive. This argument is an extended expression and can be specified either as a literal number or as a numeric expression enclosed in parentheses.

<idAlias> is the name of an existing work area to SELECT if there is a database file open in that area.

Description

SELECT is a database command that changes work areas. CA-Clipper supports 250 work areas, with each work area a logical handle to an open database file and all of its attributes. You can refer to work areas with SELECT by number or by alias. The alias of a work area is automatically assigned when a database file is USEd in that work area or by using the ALIAS clause.

Work area 0 refers to the first empty or next available work area. Using this, you can SELECT 0 and USE *<xcDatabase>* as a method of opening database files.

Notes

- ***Aliased expressions:*** Aliased expressions are a much more powerful method of selecting new work areas than the SELECT command. Instead of SELECTing a work area, and then performing an operation for that work area, you can apply an alias to an expression that performs the operation. This is done by specifying the alias of the remote work area and the expression enclosed in parentheses. For example, to access the value of EOF() in an unselected work area, you would normally execute a series of statements like the following:

```
SELECT Remote
? EOF()
SELECT Main
```

Using the aliased expression form, these statements become:

```
? Remote->(EOF())
```

- ***USE...NEW:*** Instead of using SELECT0 and USE <*xcDatabase*> to open a database file in a new work area, the preferred method is to USE <*xcDatabase*> NEW.

Examples

- This example opens a series of database files by SELECTing each work area by number then USEing each database file in that work area:

```
SELECT 1
USE Customer
SELECT 2
USE Invoices
SELECT 3
USE Parts
SELECT Customer
```

- A better method is to open each database in the next available work area by specifying the NEW clause on the USE command line. In this example USE...NEW is employed instead of SELECT0 and then USE:

```
USE Customer NEW
USE Invoices NEW
SELECT Customer
```

- This code fragment changes work areas while saving the current work area name to a variable using the SELECT() function. After executing an operation for the new work area, the original work area is restored:

```
nLastArea := SELECT()
USE Newfile NEW
//
<statements>...
//
SELECT (nLastArea)
```

Files Library is CLIPPER.LIB.

See Also ALIAS(), EOF(), SELECT(), SET INDEX, USE, USED()

SELECT() function

Determine the work area number of a specified alias

Syntax

```
SELECT([<cAlias>]) → nWorkArea
```

Arguments

<cAlias> is the target work area alias name.

Returns

SELECT() returns the work area of the specified alias as an integer numeric value.

Description

SELECT() is a database function that determines the work area number of an alias. The number returned can range from 0 to 250. If *<cAlias>* is not specified, the current work area number is returned. If *<cAlias>* is specified and the alias does not exist, SELECT() returns zero.

Note: The SELECT() function and SELECT command specified with an extended expression argument look somewhat alike. This should not be a problem since the SELECT() function is not very useful on a line by itself.

Examples

- This example uses SELECT() to determine which work area USE...NEW selected:

```
USE Sales NEW
SELECT 1
? SELECT("Sales")          // Result: 4
```

- To reselect the value returned from the SELECT() function, use the SELECT command with the syntax, SELECT (*<idMemvar>*), like this:

```
USE Sales NEW
nWorkArea:= SELECT()
USE Customer NEW
SELECT (nWorkArea)
```

Files

Library is CLIPPER.LIB.

See Also

ALIAS(), SELECT, USE, USED()

SET ALTERNATE command

Echo console output to a text file

Syntax

```
SET ALTERNATE TO [<xcFile> [ADDITIVE]]
SET ALTERNATE on | OFF | <xlToggle>
```

Arguments

TO <xcFile> opens a standard ASCII text file for output with a default extension of .txt. The file name may optionally include an extension, drive letter, and/or path. You may specify *<xcFile>* either as a literal file name or as a character expression enclosed in parentheses. Note that if a file with the same name exists, it is overwritten.

ADDITIVE causes the specified alternate file to be appended instead of being overwritten. If not specified, the specified alternate file is truncated before new information is written to it.

ON causes console output to be written to the open text file.

OFF discontinues writing console output to the text file without closing the file.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET ALTERNATE is a console command that lets you write the output of console commands to a text file. Commands such as LIST, REPORT FORM, LABEL FORM, and ? that display to the screen without reference to row and column position are console commands. Most of these commands have a TO FILE clause that performs the same function as SET ALTERNATE. Full-screen commands such as @...SAY cannot be echoed to a disk file using SET ALTERNATE. Instead you can use SET PRINTER TO <*xcFile*> with SET DEVICE TO PRINTER to accomplish this.

SET ALTERNATE has two basic forms. The TO <*xcFile*> form creates a DOS text file with a default extension of .txt and overwrites any other file with the same name. Alternate files are not related to work areas with only one file open at a time. To close an alternate file, use CLOSE ALTERNATE, CLOSE ALL, or SET ALTERNATE TO with no argument.

The on | OFF form controls the writing of console output to the current alternate file. SET ALTERNATE ON begins the echoing of output to the alternate file. SET ALTERNATE OFF suppresses output to the alternate file but does not close it.

Examples

- This example creates an alternate file and writes the results of the ? command to the file for each record in the Customer database file:

```
SET ALTERNATE TO Listfile
SET ALTERNATE ON
USE Customer NEW
DO WHILE !EOF()
   ? Customer->Lastname, Customer->City
   SKIP
ENDDO
SET ALTERNATE OFF
CLOSE ALTERNATE
CLOSE Customer
```

Files

Library is CLIPPER.LIB.

See Also

CLOSE, FCREATE(), FOPEN(), FWRITE(), SET CONSOLE, SET PRINTER

SET BELL command

Toggle automatic sounding of the bell during full-screen operations

Syntax

```
SET BELL on | OFF | <xlToggle>
```

Arguments

ON enables the BELL.

OFF disables the BELL.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET BELL is an environment command that toggles the sound of the bell. If SET BELL is ON, the bell sounds in the following situations:

- The user enters a character at the last position in a GET.
- The user attempts to enter invalid data into a GET. The data is validated by the data type of the GET variable, the PICTURE template, and by the RANGE clause. Violating a VALID condition does not sound the bell, regardless of the SET BELL status.

To sound the bell explicitly, you can use either ?? CHR(7) or the TONE() function. TONE() is perhaps more useful since you can vary both the pitch and duration of the sound.

Files

Library is CLIPPER.LIB.

See Also

@...GET, CHR(), SET CONFIRM, TONE()

SET CENTURY command

Modify the date format to include or omit century digits

Syntax

```
SET CENTURY on | OFF | <xlToggle>
```

Arguments

ON allows input and display of the century digits for dates.

OFF suppresses the input and display of the century digits for dates.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CENTURY modifies the current date format as set by SET DATE. SET CENTURY ON changes the date format to contain four digits for the year. With the date format set to four digits for the year, date values display with a four-digit year, and dates of any century can be input.

SET CENTURY OFF changes the date format to contain only two digits for the year. With the date format set to only two digits for the year (CENTURY OFF), the century digits of dates are not displayed and cannot be input.

Note that only the display and input format of dates is affected; date calculations maintain the century information regardless of the date format.

CA-Clipper supports all dates in the range 01/01/0100 to 12/31/2999.

Examples

- This example shows the results of a simple SET CENTURY command:

```
SET CENTURY OFF
? DATE()                  // Result: 09/15/90
SET CENTURY ON
? DATE()                  // Result: 09/15/1990
```

Files

Library is CLIPPER.LIB.

See Also

CTOD(), DATE(), DTOC(), SET DATE, SET EPOCH

SET COLOR* command

Define screen colors

Syntax

```
SET COLOR | COLOUR TO [[<standard>]
   [,<enhanced>] [,<border>] [,<background>]
   [,<unselected>]] | (<cColorString>)
```

Arguments

<standard> is the color that paints all console, full-screen, and interface commands and functions when displaying to the screen. This includes commands such as @...PROMPT, @...SAY, and ?; as well as functions such as ACHOICE(), DBEDIT(), and MEMOEDIT().

<enhanced> is the color that paints highlighted displays. This includes GETs with INTENSITY ON, the MENU TO, DBEDIT(), and ACHOICE() selection highlight.

<border> is the color that paints the area around the screen that cannot be written to.

<background> is not currently supported by any machines for which Computer Associates provides drivers. This setting is supplied for compatibility purposes only.

<unselected> is a color pair that provides input focus by displaying the current GET in the enhanced color while other GETs are displayed in this color.

<cColorString> is a character string enclosed in parentheses containing the color settings. This facility lets you specify the color settings as an expression in place of a literal string or macro variable.

SET COLOR TO with no argument restores the default colors to W/N, N/W, N, N, N/W.

Description

SET COLOR, a command synonym for the SETCOLOR() function, defines colors for subsequent screen painting activity. Each SET COLOR command specifies a list of color settings for the five types of screen painting activity. Each setting is a foreground and background color pair separated by the slash (/) character. Foreground defines the color of characters displayed on the screen. Background defines the color displayed behind the character. Spaces and nondisplay characters display as background only.

In addition to color, a foreground setting can have an attribute, high intensity or blinking. With a monochrome display, high intensity enhances brightness of painted text. With a color display, high intensity changes the hue of the specified color making it a different color. For example, N displays foreground text as black where N+ displays the same text as gray. High intensity is denoted by +. The blinking attribute causes the foreground text to flash on and off at a rapid interval. Blinking is denoted with *. An attribute character can occur anywhere in a setting, but is always applied to the foreground color regardless where it occurs.

Each color can be specified using either a letter or a number, but numbers and letters cannot be mixed within a setting. Note that numbers are supplied for compatibility purposes and are not recommended.

All settings are optional. If a setting is skipped, its previous value is retained with only new values set. Skipping a foreground or background color within a setting sets the color to black.

The following colors are supported:

Color Table

Color	Letter	Number	Monochrome
Black	N, Space	0	Black
Blue	B	1	Underline
Green	G	2	White
Cyan	BG	3	White
Red	R	4	White
Magenta	RB	5	White
Brown	GR	6	White
White	W	7	White
Gray	N+	8	Black
Bright Blue	B+	9	Bright Underline
Bright Green	G+	10	Bright White
Bright Cyan	BG+	11	Bright White
Bright Red	R+	12	Bright White
Bright Magenta	RB+	13	Bright White
Yellow	GR+	14	Bright White
Bright White	W+	15	Bright White
Black	U		Underline
Inverse Video	I		Inverse Video
Blank	X		Blank

SET COLOR is a compatibility command and is not recommended. It is superseded by the SETCOLOR() function which can return the current color as well as set a new color.

Notes

- ***Monochrome monitors:*** Color is not supported on monochrome monitors. CA-Clipper, however, supports the monochrome attributes inverse video (I) and underlining (U).
- ***Screen drivers:*** SET COLOR TO, using numbers, may not be supported by screen drivers other than the default screen driver.

Examples

- This example uses the unselected setting to make the current GET red on white while the rest are black on white:

```
cColor:= "W/N,R/W,,,N/W"
SET COLOR TO (cColor)
cOne := cTwo := SPACE(10)
@ 1, 1 SAY "Enter One: " GET cOne
@ 2, 1 SAY "Enter Two: " GET cTwo
READ
```

- In this example a user-defined function gets a password from the user using the blank (X) enhanced setting to hide the password as the user types:

```
IF !DialogPassWord(12, 13, "W+/N", "FUNSUN", 3)
   ? "Sorry, your password failed"
   QUIT
ENDIF

FUNCTION DialogPassWord( nRow, nCol, ;
       cStandard, cPassword, nTries )
   LOCAL nCount := 1, cColor := SETCOLOR()
   SET COLOR TO (cStandard + ", X")    // Blank input
   //
   DO WHILE nCount < nTries
      cUserEntry:= SPACE(6)
      @ nRow, nCol SAY  "Enter password: " GET ;
               cUserEntry
      READ
      //
      IF LASTKEY() == 27
         SET COLOR TO (cColor)
         RETURN .F.
      ELSEIF cUserEntry == cPassword
         SET COLOR TO (cColor)
         RETURN .T.
      ELSE
         nCount++
      ENDIF
   ENDDO
   //
   SET COLOR TO (cColor)
   RETURN .F.
```

Files Library is CLIPPER.LIB.

See Also @...GET, @...SAY, ISCOLOR(), SETCOLOR(), SETBLINK()

SET CONFIRM command

Toggle required exit key to terminate GETs

Syntax

```
SET CONFIRM on | OFF | <xlToggle>
```

Arguments

ON requires the user to press an exit key to leave a GET.

OFF allows the user to leave a GET by typing past the end without pressing an exit key.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CONFIRM determines whether an exit key is required to leave a GET. If CONFIRM is OFF, the user can type past the end of a GET and the cursor will move to the next GET, if there is one. If there is not another GET, the READ terminates. If, however, CONFIRM is ON, an exit key must be pressed to leave the current GET.

In all cases, attempting to leave the current GET executes the RANGE or VALID clauses, unless the user presses the Esc key.

See @...GET for more information on the behavior of GETs.

Files Library is CLIPPER.LIB.

See Also @...GET, READ, SET BELL

SET CONSOLE command

Toggle console display to the screen

Syntax

```
SET CONSOLE ON | off | <xlToggle>
```

Arguments

ON displays the output of console commands on the screen.

OFF suppresses the screen display of console commands.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CONSOLE determines whether or not console commands send output to the screen. Console commands are commands that display to the screen without reference to row and column position. In addition to sending output to the screen, console commands can simultaneously send output to the printer and/or a DOS text file. Output is sent to the printer using the TO PRINTER clause common to many console commands, or with the SET PRINTER ON command. Output is sent to a file using the TO FILE clause, SET ALTERNATE, or SET PRINTER TO.

With CONSOLE ON, console commands display to the screen. With CONSOLE OFF, the screen display of console commands is suppressed, but the echoing of output to either a file or the printer is unaffected. This lets you send the output of console commands such as REPORT and LABEL FORM to the printer without the screen display—a common occurrence.

Notes

- ***Keyboard input:*** For console commands that accept input (including ACCEPT, INPUT, and WAIT), SET CONSOLE affects the display of the prompts as well as the input areas. As a consequence, a SET CONSOLE OFF before one of these commands will not only prevent you from seeing what you type, but will also prevent the display of the message prompt.
- ***Full-screen commands:*** Full-screen commands such as @...SAY display to the screen independent of the current CONSOLE SETting. For this category of output commands, device control is performed using SET DEVICE to control whether output goes to the screen or printer, and SET PRINTER TO echo output to a file.

Examples

- This example uses REPORT FORM to output records to the printer while suppressing output to the screen:

```
USE Sales NEW
SET CONSOLE OFF
REPORT FORM Sales TO PRINTER
SET CONSOLE ON
```

Files Library is CLIPPER.LIB.

See Also SET DEVICE

SET CURSOR command

Toggle the screen cursor on or off

Syntax

```
SET CURSOR ON | off | <xlToggle>
```

Arguments

ON enables the cursor display.

OFF disables the cursor display.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CURSOR toggles the screen cursor on or off. When the CURSOR is OFF, keyboard entry and screen display are unaffected. The cursor is merely hidden and data entry may still be accomplished without the cursor being visible. ROW() and COL() are updated as if the cursor were visible.

This command suppresses the cursor while the screen is being painted. Ideally, the only time the cursor shows in a production program is when the user is editing GETs, MEMOEDIT(), or some kind of line edit.

Examples

- This example shows a typical use of SET CURSOR:

```
LOCAL lAnswer := .F.
@ 24, 0
@ 24, 15 SAY "Do you want to QUIT? [Y/N]";
   GET lAnswer;
   PICTURE "Y"
SET CURSOR ON
READ
SET CURSOR OFF
```

Files

Library is CLIPPER.LIB.

See Also

COL(), ROW(), SET CONSOLE, SETCURSOR(), SETPOS()

SET DATE command

Set the date format for input and display

Syntax

```
SET DATE FORMAT [TO] <cDateFormat>
SET DATE [TO] AMERICAN | ansi | British | French
    | German | Italian | Japan | USA
```

Arguments

<cDateFormat> is a character expression that directly specifies the date format when the FORMAT clause is specified. *<cDateFormat>* must evaluate to a string of 12 or fewer characters.

When specified, *<cDateFormat>* is analyzed to determine the proper placement and number of digits for the day, month, and year. The position of the day, month, and year digits is determined by scanning the string for one or more occurrences of the letters *d*, *m*, and *y*, respectively. Other characters in the string are copied verbatim into displayed date values.

When FORMAT is not used, one of several keywords describes the date format. The following table shows the format for each keyword setting:

SET DATE Formats

SETting	Format
AMERICAN	mm/dd/yy
ANSI	yy.mm.dd
BRITISH	dd/mm/yy
FRENCH	dd/mm/yy
GERMAN	dd.mm.yy
ITALIAN	dd-mm-yy
JAPAN	yy/mm/dd
USA	mm-dd-yy

Description

SET DATE is an environment command that sets the display format for date values. SET DATE is a global setting that affects the behavior of dates throughout a program, allowing you to control date formatting in a way that facilitates porting applications to foreign countries.

Examples

- In this example the FORMAT clause directly specifies the date format:

```
SET DATE FORMAT "yyyy:mm:dd"
```

- This example configures the date setting at runtime by passing a DOS environment variable to the program, retrieving its value with GETENV(), and setting DATE with the retrieved value:

```
C>SET CLIP_DATE=dd/mm/yy
```

  In the configuration section of the application program, the date format is set like this:

```
FUNCTION AppConfig
   SET DATE FORMAT TO GETENV("CLIP_DATE")
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also CTOD(), DATE(), DTOC(), DTOS(), SET CENTURY, SET EPOCH

SET DECIMALS command

Set the number of decimal places to be displayed

Syntax

```
SET DECIMALS TO [<nDecimals>]
```

Arguments

TO <nDecimals> is the number of decimal places to be displayed. The default value is two.

SET DECIMALS TO with no argument is equivalent to SET DECIMALS TO 0.

Description

SET DECIMALS determines the number of decimal places displayed in the results of numeric functions and calculations. Its operation depends directly on the FIXED setting. If FIXED is OFF, SET DECIMALS establishes the minimum number of decimal digits displayed by EXP(), LOG(), SQRT(), and division operations. If FIXED is ON, all numeric values are displayed with exactly the number of decimal places specified by SET DECIMALS. Note that neither SET DECIMALS nor SET FIXED affects the actual numeric precision of calculations—only the display format is affected.

To provide finer control of numeric display, you can use the PICTURE clause of @...SAY, @...GET, and the TRANSFORM() function.

Examples

- These examples show various results of the SET DECIMALS command:

```
SET FIXED ON
SET DECIMALS TO 2          // The default setting
? 2/4                      // Result: 0.50
? 1/3                      // Result: 0.33
SET DECIMALS TO 4
? 2/4                      // Result: 0.5000
? 1/3                      // Result: 0.3333
```

Files

Library is CLIPPER.LIB.

See Also

@...GET, @...SAY, SET FIXED, TRANSFORM()

SET DEFAULT command

Set the CA-Clipper default drive and directory

Syntax

```
SET DEFAULT TO [<xcPathspec>]
```

Arguments

TO <xcPathspec> identifies a disk drive and the directory as the default and can be specified either as a literal path specification or as a character expression enclosed in parentheses. If you specify both a drive and directory, a colon must be included after the drive letter.

SET DEFAULT TO specified without an argument defaults to the current DOS drive and directory.

Description

SET DEFAULT sets the drive and directory where the application program creates and saves files, with the exception of temporary files and files created with the low-level file functions.

SET DEFAULT does not change the DOS drive and directory. When attempting to access files, the DEFAULT drive and directory are searched first. To set additional search paths for file access, use SET PATH.

Notes

- ***Initial default:*** When a CA-Clipper program starts, the default drive and directory are the current DOS drive and directory. Within the program, you can change this with SET DEFAULT.
- ***Running external programs:*** Executing a RUN command accesses the current DOS drive and directory.

Examples

- This example shows a typical use of SET DEFAULT:

```
SET PATH TO
? FILE("Sales.dbf")      // Result: .F.
//
SET DEFAULT TO C:\CLIPPER\FILES
? FILE("Sales.dbf")      // Result: .T.
//
SET DEFAULT TO C:        // Change default drive
SET DEFAULT TO \         // Change to root directory
SET DEFAULT TO ..        // Change to parent directory
```

Files Library is CLIPPER.LIB.

See Also CURDIR(), SET PATH

SET DELETED command

Toggle filtering of deleted records

Syntax

```
SET DELETED on | OFF | <xlToggle>
```

Arguments

ON ignores deleted records.

OFF processes deleted records.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET DELETED toggles automatic filtering of records marked for deletion in all work areas. When SET DELETED is ON, most commands ignore deleted records. If, however, you refer to a record by record number (GOTO or any command that supports the RECORD scope), the record is not ignored even if marked for deletion. Additionally, SET DELETED ON has no affect on INDEX or REINDEXing.

RECALL ALL honors SET DELETED and does not recall any records.

Notes

- ***Filtering deleted records in a single work area:*** To confine the filtering of deleted records to a particular work area, SELECT the work area, and then SET FILTER TO DELETED().

Examples

- This example illustrates the effect of using SET DELETED:

```
USE Sales NEW
? LASTREC()                 // Result: 84
//
DELETE RECORD 4
COUNT TO nCount
? nCount                    // Result: 84
//
SET DELETED ON
COUNT TO nCount
? nCount                    // Result: 83
```

Files Library is CLIPPER.LIB.

See Also DELETE, DELETED(), RECALL ALL, SET FILTER

SET DELIMITERS command

Toggle or define GET delimiters

Syntax

```
SET DELIMITERS on | OFF | <xlToggle>
SET DELIMITERS TO [<cDelimiters> | DEFAULT]
```

Arguments

ON displays delimiters for GET variables.

OFF suppresses the delimiter display.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

TO <cDelimiters> defines a one or two character delimiter. Specifying a single character uses the same character as both the beginning and ending delimiter. Specifying two characters uses the first as the beginning delimiter and the second as the ending delimiter.

TO DEFAULT or no delimiters SETs the delimiters to colons which are the default delimiters.

Description

SET DELIMITERS is a dual purpose command that both defines characters used to delimit GETs and toggles the automatic display of delimiters ON or OFF. The @...GET command can display delimiters that surround a Get object's display. If DELIMITERS is ON, the delimiters add two characters to the length of the Get object display.

You can configure the delimiter characters using the TO *<cDelimiters>* clause. The DEFAULT delimiter character is the colon (:). When specifying delimiters, the beginning and ending delimiter characters can be different. If you wish to suppress either the right, left, or both delimiters, use a space instead of the delimiter character.

Typically, delimiters are unnecessary since GETs display in reverse video or enhanced color if INTENSITY is ON.

Examples

- This example SETs DELIMITERS TO a colon and a space for the first GET and the square bracket characters for the second:

```
LOCAL cVar := SPACE(5), cVar2 := SPACE(5)
SET DELIMITERS ON
SET DELIMITERS TO ": "
@ 1, 0 SAY "Enter" GET cVar
SET DELIMITERS TO "[]"
@ 2, 0 SAY "Enter" GET cVar2
READ
```

Files Library is CLIPPER.LIB.

See Also @...GET, SET INTENSITY

SET DESCENDING command

Change the descending flag of the controlling order

Syntax

```
SET DESCENDING ON | OFF | (<lToggle>)
```

Arguments

ON enables the descending flag.

OFF disables the descending flag.

<lToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Note: The initial default of this setting depends on whether the controlling order was created with DESCENDING as an attribute.

Description

SET DESCENDING is functionally equivalent to ORDDESCEND(). Refer to this function for more information.

Examples

- The following example illustrates SET DESCENDING. Every order can be both ascending and descending:

```
USE Customer VIA "DBFCDX"
INDEX ON LastName TAG Last
INDEX ON FirstName TAG First DESCENDING

SET ORDER TO TAG Last
// last was originally created in ascending order

// Swap it to descending
SET DESCENDING ON
// Last will now be processed in descending order

SET ORDER TO TAG First
// First was originally created in descending order

// Swap it to ascending
SET DESCENDING OFF
// First will now be processed in ascending order
```

Files

Library is CLIPPER.LIB.

See Also

ORDDESCEND()

SET DEVICE command

Direct @...SAYs to the screen or printer

Syntax

```
SET DEVICE TO SCREEN | printer
```

Arguments

TO SCREEN directs all @...SAYs to the screen and is independent of the SET PRINTER and CONSOLE settings.

TO PRINTER directs all @...SAYs to the device set with SET PRINTER TO. This can include a local printer port, a network spooler, or a file.

Description

SET DEVICE directs the output of @...SAY commands to either the screen or the printer. When DEVICE is SET TO PRINTER, @...SAY commands are sent to the printer and not echoed to the screen. In addition, @...SAY commands observe the current SET MARGIN value.

When sending @...SAYs to the printer, CA-Clipper performs an automatic EJECT whenever the current printhead row position is less than the last print row position. An EJECT resets PCOL() and PROW() values to zero. To reset PCOL() and PROW() to new values, use the SETPRC() function.

To send @...SAYs to a file, use SET PRINTER TO *<xcFile>* with SET DEVICE TO PRINTER.

Examples

- This example directs @...SAYs to the printer:

```
SET DEVICE TO PRINTER
@ 2,10 SAY "Hello there"
EJECT
```

- This example directs @...SAYs to a file:

```
SET PRINTER TO Output.txt
SET DEVICE TO PRINTER
@ 10, 10 SAY "File is: Output.txt"
@ 11, 10 SAY DATE()
SET PRINTER TO          // Close the output file
SET DEVICE TO SCREEN
```

Files

Library is CLIPPER.LIB.

See Also

@...SAY, EJECT, PCOL(), PROW(), SET PRINTER, SETPRC()

SET EPOCH command

Control the interpretation of dates with no century digits

Syntax

```
SET EPOCH TO <nYear>
```

Arguments

TO <nYear> specifies the base year of a 100-year period in which all dates containing only two year digits are assumed to fall.

Description

SET EPOCH is an environment command that determines the interpretation of date strings containing only two year digits. When such a string is converted to a date value, its year digits are compared with the year digits of *<nYear>*. If the year digits in the date are greater than or equal to the year digits of *<nYear>*, the date is assumed to fall within the same century as *<nYear>*. Otherwise, the date is assumed to fall in the following century.

The default value for SET EPOCH is 1900, causing dates with no century digits to be interpreted as falling within the twentieth century.

CA-Clipper supports all dates in the range 01/01/0100 to 12/31/2999.

Examples

- This example shows the effects of SET EPOCH:

```
SET DATE FORMAT TO "mm/dd/yyyy"
? CTOD("05/27/1904")          // Result: 05/27/1904
? CTOD("05/27/67")            // Result: 05/27/1967
? CTOD("05/27/04")            // Result: 05/27/1904
//
SET EPOCH TO 1960
? CTOD("05/27/1904")          // Result: 05/27/1904
? CTOD("05/27/67")            // Result: 05/27/1967
? CTOD("05/27/04")            // Result: 05/27/2004
```

Files

Library is CLIPPER.LIB.

See Also

CTOD(), DATE(), DTOC(), DTOS(), SET CENTURY, SET DATE

SET ESCAPE command

Toggle Esc as a READ exit key

Syntax

```
SET ESCAPE ON | off | <xlToggle>
```

Arguments

ON enables Esc as a READ exit key.

OFF disables Esc as a READ exit key.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

If SET ESCAPE is ON, Esc terminates the current READ. Any changes made to the current Get object are lost, and validation with RANGE or VALID is bypassed. When SET ESCAPE is OFF and the user presses Esc, the key press is ignored. With SET KEY, however, you can reassign Esc for special handling, regardless of the status of SET ESCAPE.

Files Library is CLIPPER.LIB.

See Also READ, READEXIT(), SETCANCEL(), SET KEY, SETKEY()

SET EVENTMASK command

Specify events to be returned by the INKEY() function

Syntax

```
SET EVENTMASK TO <nEventMask>
```

Arguments

<nEventMask> specifies which events should be returned by the INKEY() function. This argument can be any combination of the following values which are defined in Inkey.ch:

Inkey Constants

Constant	Value	Description
INKEY_MOVE	1	Mouse Events
INKEY_LDOWN	2	Mouse Left Click Down
INKEY_LUP	4	Mouse Left Click Up
INKEY_RDOWN	8	Mouse Right Click Down
INKEY_RUP	16	Mouse Right Click Up
INKEY_KEYBOARD	128	Keyboard Events
INKEY_ALL	159	All Mouse and Keyboard Events

If a value is not specified for SET EVENTMASK, the default value of 128 (keyboard events only) will be used.

Description

The SET EVENTMASK command specifies which events should be returned by the INKEY() function. Using this mask you can have INKEY() return only the events in which you are interested.

Example

The following example will inform INKEY() to terminate if a keyboard event occurs or the left mouse button has been clicked. If no events occur within 5 seconds, INKEY() will terminate.

```
SET EVENTMASK TO INKEY_KEYBOARD + INKEY_LDOWN
? INKEY( 5 )
```

See Also

INKEY()

SET EXACT* command

Toggle exact matches for character strings

Syntax

```
SET EXACT on | OFF | <xlToggle>
```

Arguments

ON enforces exact comparison of character strings including length.

OFF resumes normal character string comparison.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET EXACT determines how two character strings are compared using the relational operators (=, >, <, =>, =<). When EXACT is OFF, strings are compared according to the following rules. Assume two character strings cLeft and cRight where the expression to test is (cLeft = cRight):

- If cRight is a null string (""), return true (.T.).
- If LEN(cRight) is greater than LEN(cLeft), return false (.F.).
- Otherwise, compare all characters in cRight with cLeft. If all characters in cRight equal cLeft, return true (.T.); otherwise, return false (.F.).

With EXACT ON, all relational operators except the double equal operator (==) treat two strings as equal, if they match exactly, excluding trailing spaces. With the double equal operator (==), all characters in the string are significant, including trailing spaces.

SET EXACT is a compatibility command and not recommended.

Notes

- ***Compatibility:*** In CA-Clipper, unlike other dialects, SET EXACT has no affect on operations other than relational operators. This includes the SEEK and FIND commands. If you need to seek exact matches of character keys, use the example user-defined function SeekExact() in the SEEK command reference.

Examples

- These examples show various results of the equal operator (=) with SET EXACT:

```
SET EXACT OFF
? "123" = "12345"          // Result: .F.
? "12345" = "123"          // Result: .T.
? "123" = ""               // Result: .T.
? "" = "123"               // Result: .F.
? "123" = "123   "         // Result: .F.
//
SET EXACT ON
? "123" = "12345"          // Result: .F.
? "12345" = "123"          // Result: .F.
? "123" = ""               // Result: .F.
? "" = "123"               // Result: .F.
? "123" = "123   "         // Result: .T.
```

Files Library is CLIPPER.LIB.

See Also SEEK

SET EXCLUSIVE* command

Establish shared or exclusive USE of database files

Syntax

```
SET EXCLUSIVE ON | off | <xlToggle>
```

Arguments

ON causes database files to be opened in exclusive (nonshared) mode.

OFF causes database files to be opened in shared mode.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

In a network environment, SET EXCLUSIVE determines whether a USE command specified without the EXCLUSIVE or SHARED clause automatically opens database, memo, and index files EXCLUSIVE. When database files are opened EXCLUSIVE, other users cannot USE them until they are CLOSEd. In this mode, file and record locks are unnecessary.

When EXCLUSIVE is ON (the default), all database and associated files open in a nonshared (exclusive) mode unless the USE command is specified with the SHARED clause. Use EXCLUSIVE only for operations that absolutely require EXCLUSIVE USE of a database file, such as PACK, REINDEX, and ZAP.

When EXCLUSIVE is OFF, all files are open in shared mode unless the USE command is specified with the EXCLUSIVE clause. Control access by other users programmatically using RLOCK() and FLOCK().

SET EXCLUSIVE is a compatibility command and not recommended. It is superseded by the EXCLUSIVE and SHARED clauses of the USE command.

Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Notes

- ***Error handling:*** Attempting to USE a database file already opened EXCLUSIVE by another user generates a runtime error and sets NETERR() to true (.T.). After control returns to the point of error, you can test NETERR() to determine whether the USE failed.

Files Library is CLIPPER.LIB.

See Also FLOCK(), NETERR(), RLOCK(), USE

SET FILTER command

Hide records not meeting a condition

Syntax

```
SET FILTER TO [<lCondition>]
```

Arguments

TO <lCondition> is a logical expression that defines a specific set of current work area records accessible for processing.

SET FILTER TO without an argument deactivates the filter condition.

Description

When a FILTER condition is SET, the current work area acts as if it contains only the records that match the specified condition. A filter condition is one of the properties of a work area. Once a FILTER has been SET, the condition can be returned as a character string using the DBFILTER() function.

Most commands and functions that move the record pointer honor the current filter with the exception of those commands that access records by record number. This includes GOTO, commands specified with the RECORD clause, and RELATIONs linked by numeric expression to a work area with no active index.

Once a FILTER is SET, it is not activated until the record pointer is moved from its current position. You can use GO TOP to activate it.

As with SET DELETED, a filter has no effect on INDEX and REINDEX.

Note: Although SET FILTER makes the current work area appear as if it contains a subset of records, it, in fact, sequentially processes all records in the work area. Because of this, the time required to process a filtered work area will be the same as an unfiltered work area.

Examples

- This example filters Employee.dbf to only those records where the age is greater than 50:

```
USE Employee INDEX Name NEW
SET FILTER TO Age > 50
LIST Lastname, Firstname, Age, Phone
SET FILTER TO
```

Files Library is CLIPPER.LIB.

See Also DBFILTER(), DBSETFILTER(), SET DELETED

SET FIXED command

Toggle fixing of the number of decimal digits displayed

Syntax

```
SET FIXED on | OFF | <xlToggle>
```

Arguments

ON fixes the decimal places display at the number of digits specified by SET DECIMALS.

OFF allows the operation or function to determine the number of decimal places to display.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET FIXED toggles control of the display of decimal digits by the current DECIMALS setting. When FIXED is ON, display of all numeric output is fixed at the DECIMALS setting (two places if the SET DECIMALS default value is in effect). When FIXED is OFF, numeric output displays according to the default rules for numeric display. These are described in the "Basic Concepts" chapter of the *Programming and Utilities Guide*.

Note that SET FIXED and SET DECIMALS affect only the display format of numeric values and not the actual numeric precision of calculations.

Files

Library is CLIPPER.LIB.

See Also

EXP(), LOG(), SET DECIMALS, SQRT(), VAL()

SET FORMAT* command

Activate a format when READ is executed

Syntax

```
SET FORMAT TO [<idProcedure>[.<ext>]]
```

Arguments

TO <idProcedure> is a format (.fmt) file, a program (.prg) file, or a procedure.

<ext> is the extension of the format file. If not specified, the default extension is (.fmt).

SET FORMAT TO with no argument deactivates the current format.

Description

SET FORMAT defines a procedure to execute when a READ is invoked. Unlike the interpreted environment, formats are not opened and executed at runtime. Instead, the CA-Clipper compiler treats SET FORMAT the same as a DO command. The compiler first looks to see whether it has already compiled a procedure with the same name as <*idProcedure*>. If it has, it uses that procedure for the reference. If <*idProcedure*> is not found, the compiler looks to disk for a file with the same name. If this file is not found, an external reference is generated that must be resolved at link time.

SET FORMAT is a compatibility command and not recommended.

Notes

- ***Active format procedures:*** Unlike other dialects where each work area can have an active format, CA-Clipper supports only one active format procedure for all work areas.
- ***Screen CLEARing:*** CA-Clipper does not clear the screen when a format procedure is executed.
- ***Legal statements:*** Format procedures allow statements and commands in addition to @...SAY and @...GET.
- ***Multiple pages:*** CA-Clipper does not support multiple-page format procedures.

Examples

- This example uses a format procedure to add records to a database file until the user presses Esc:

```
USE Sales NEW
SET FORMAT TO SalesScr
DO WHILE LASTKEY() != 27
   APPEND BLANK
   READ
ENDDO
RETURN

PROCEDURE SalesScr
   @ 12, 12 SAY "Branch     : " GET  Branch
   @ 13, 12 SAY "Salesman   : " GET  Salesman
   RETURN
```

Files Library is CLIPPER.LIB.

See Also @...GET, @...SAY, PROCEDURE, READ

SET FUNCTION command

Assign a character string to a function key

Syntax

```
SET FUNCTION <nFunctionKey> TO <cString>
```

Arguments

<nFunctionKey> is the number of the function key to receive the assignment.

TO <cString> specifies the character string to assign to *<nFunctionKey>*.

Description

SET FUNCTION assigns a character string to a function key numbered between 1 and 40 inclusive. When the user presses the assigned function key, *<cString>* is stuffed into the keyboard buffer. *<cString>* can include control characters, such as a Ctrl+C or Ctrl+S to perform navigation or editing actions in a wait state.

List of Function Key Mappings

Function Key	Actual Key
1 - 10	F1 - F10
11 - 20	Shift+F1 - Shift+F10
21 - 30	Ctrl+F1 - Ctrl+F10
31 - 40	Alt+F1 - Alt+F10

Warning! *In CA-Clipper, SET FUNCTION is preprocessed into SET KEY and KEYBOARD commands. This means that SET FUNCTION has the effect of clearing any SET KEY for the same key number and vice versa. This is incompatible with previous releases, which maintained separate lists of SET FUNCTION keys and SET KEY keys.*

Files

Library is CLIPPER.LIB.

See Also

KEYBOARD, SET KEY, SETKEY()

SET INDEX command

Open one or more order bags in the current work area

Syntax

```
SET INDEX TO [<xcOrderBagName list>] [ADDITIVE]
```

Arguments

<cOrderBagName list> specifies order bags to be emptied into the order list of the current work area.

ADDITIVE adds order bags to an existing order list.

Description

By default, SET INDEX, without the ADDITIVE clause, clears the currently active order list, and then constructs a new order list from the orders in the specified order bags in the current work area. When several order bags are opened, the first order in the first order bag becomes the *controlling order* (has focus). The record pointer is initially positioned at the first logical identity in this order.

If an order list exists when you SET INDEX ... ADDITIVE, the orders in the new order bag are added to the end of the order list. The previous controlling order continues to be the controlling order.

If no order list exists when you SET INDEX ... ADDITIVE, the first order in the first order bag in *<cOrderBagName list>* becomes the controlling order.

During database processing, all open orders are updated whenever a key value is appended or changed, unless the order was created using a scoping condition and the key value does not match. To change the controlling order without issuing another SET INDEX command, use SET ORDER or ORDSETFOCUS(). To add orders without closing the currently open orders, use the ADDITIVE clause.

Examples

- This example opens a database and its associated indexes:

```
USE Sales NEW
SET INDEX TO Sales, Sales1, Sales2
```

- This example opens an index without closing any indexes that are already open:

```
SET INDEX TO Sales3 ADDITIVE
```

See Also

CLOSE, DBCLEARINDEX(), DBSETINDEX(), INDEX, REINDEX, SET ORDER, USE

SET INTENSITY command

Toggle enhanced display of GETs and PROMPTs

Syntax

```
SET INTENSITY ON | off | <xlToggle>
```

Arguments

ON enables both standard and enhanced display colors.

OFF disables enhanced display color. All screen output then uses the current standard color.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET INTENSITY toggles the display of GETs and menu PROMPTs between enhanced and standard color settings. When INTENSITY is OFF, GETs and SAYs appear in the standard color setting. When INTENSITY is ON, GETs appear in the enhanced color setting.

When INTENSITY is OFF, all menu PROMPTs appear in the standard color setting, and the cursor appears at the current PROMPT. If INTENSITY is ON (the default), the current PROMPT appears in the enhanced color setting, and the cursor is hidden.

Note that INTENSITY has no effect on ACHOICE() or DBEDIT().

Files

Library is CLIPPER.LIB.

See Also

@...GET, @...PROMPT, @...SAY, SETCOLOR()

SET KEY command

Assign a procedure invocation to a key

Syntax

```
SET KEY <nInkeyCode> TO [<idProcedure>]
```

Arguments

<nInkeyCode> is the INKEY() value of the key that receives the assignment.

TO <idProcedure> specifies the name of a procedure that executes when the user presses the assigned key. If *<idProcedure>* is not specified, the current *<nInkeyCode>* definition is released.

Description

SET KEY is a keyboard command that allows a procedure to be executed from any wait state when a designated key is pressed. A wait state is any mode that extracts keys from the keyboard except for INKEY(). These modes include ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT. After a key is redefined, pressing it executes the specified procedure, passing three automatic parameters corresponding to PROCNAME(), PROCLINE(), and READVAR(). The procedure and variable parameters are character data type, while the line number is numeric data type.

You may define a maximum of 32 keys at one time. At startup, the system automatically defines the F1 key to execute Help. If a procedure with this name is linked into the current program and it is visible, pressing F1 from a wait state invokes it.

Note that SET KEY procedures should preserve the state of the application (i.e., screen appearance, current work area, etc.) and restore it before exiting.

Warning! *In CA-Clipper, SET FUNCTION is preprocessed into the SET KEY and KEYBOARD commands. This means that SET FUNCTION has the effect of clearing any SET KEY for the same key number and vice versa. This is incompatible with previous releases, which maintained separate lists of SET FUNCTION keys and SET KEY keys.*

Notes

- ***Precedence:*** SET KEY definitions take precedence over SET ESCAPE and SETCANCEL().
- ***CLEAR with a SET KEY procedure:*** Do not use CLEAR to clear the screen within a SET KEY procedure since it also CLEARs GETs and, therefore, terminates READ. When you need to clear the screen, use CLEAR SCREEN or CLS instead.
- ***Terminating a READ from a SET KEY procedure:*** The following table illustrates several ways to terminate a READ from within a SET KEY procedure.

Terminating a READ from a SET KEY Procedure

Command	Action
CLEAR GETS	Terminates READ without saving current GET
BREAK	Terminates READ without saving current GET
KEYBOARD Ctrl-W	Terminates READ and saves the current GET
KEYBOARD Esc	Terminates READ without saving current GET

Examples

- This example uses SET KEY to invoke a procedure that presents a picklist of account identification numbers when the user presses F2 while entering data into the account identification field:

```
#include   "Inkey.ch"
//
SET KEY K_F2 TO ScrollAccounts
USE Accounts NEW
USE Invoices NEW
@ 10, 10 GET Invoices->Id
READ
RETURN

PROCEDURE ScrollAccounts( cProc, nLine, cVar )
   IF cVar = "ID"
      SAVE SCREEN
      Accounts->(DBEDIT(10, 10, 18, 40, {"Company"}))
      KEYBOARD CHR(K_CTRL_Y) + Accounts->Id + ;
               CHR(K_HOME)
      RESTORE SCREEN
   ELSE
      TONE(100, 2)
   ENDIF
   RETURN
```

Files

Library is CLIPPER.LIB, header file is Inkey.ch.

See Also

INKEY(), KEYBOARD, LASTKEY(), PROCEDURE, PROCLINE(), PROCNAME(), READ, READVAR(), SET ESCAPE, SET FUNCTION, SETKEY()

SET MARGIN command

Set the page offset for all printed output

Syntax

```
SET MARGIN TO [<nPageOffset>]
```

Arguments

TO <nPageOffset> is a positive number that defines the number of column positions to indent from the left side of the page for subsequent printed output. A negative value resets the MARGIN to zero.

SET MARGIN TO with no argument resets the page offset to zero, the default value.

Description

SET MARGIN is valid for all output directed to the printer from console commands and @...SAY. With console output, the *<nPageOffset>* indent is output whenever there is a new line. With @...SAY, *<nPageOffset>* is added to each column value. SET MARGIN has no effect on screen output.

Note: Printing with @...SAY and PCOL() with a MARGIN SET in most cases adds the MARGIN to each column position. This happens because PCOL() accurately reflects the print column position including the last *<nPageOffset>* output. The best approach is to avoid the use of SET MARGIN with PCOL() for relative column addressing.

Examples

- This example sets a page offset of 5, and then prints a list from Sales.dbf:

```
USE Sales NEW
SET MARGIN TO 5
LIST Branch, Salesman TO PRINTER
SET MARGIN TO
```

Files Library is CLIPPER.LIB.

See Also @...SAY, PCOL(), SET DEVICE, SET PRINTER

SET MEMOBLOCK command

Change the block size for memo files

Syntax

```
SET MEMOBLOCK TO <nSize>
```

Arguments

<nSize> is the memo file block size. The initial memo file block size depends on the RDD. For most drivers that support the .dbt memo file format, it is 512 bytes. However, if you are using BLOB files (.dbv memo file format) via inheritance from the DBFMEMO driver, the default is 1.

Description

SET MEMOBLOCK is functionally equivalent to calling DBINFO(DBI_MEMOBLOCKSIZE, *<nSize>*). Refer to this function for more information. SET MEMOBLOCK sets the block size for the memo file associated with the database.

Examples

- The following example illustrates the SET MEMOBLOCK command:

```
USE Inventor NEW
SET MEMOBLOCK TO 256
? DBINFO(DBI_MEMOBLOCKSIZE)          // Result: 256
```

Files Library is CLIPPER.LIB.

See Also DBINFO()

SET MESSAGE command

Set the @...PROMPT message line row

Syntax

```
SET MESSAGE TO [<nRow> [CENTER | CENTRE]]
```

Arguments

TO <nRow> specifies the message row position.

CENTER | CENTRE centers the message on the specified row.

Specifying SET MESSAGE TO 0 or SET MESSAGE TO without an argument suppresses the display of messages.

Description

SET MESSAGE is a menu command that defines the screen row where the @...PROMPT messages display. When a CA-Clipper program is invoked the default message row value is zero, suppressing all defined messages. Messages appear on *<nRow>*, column 0 unless the CENTER option is used.

Examples

- This example creates a small lightbar menu with an activated and centered message line:

```
SET MESSAGE TO 23 CENTER
SET WRAP ON
@ 5, 5 PROMPT "One" MESSAGE "Choice one"
@ 6, 5 PROMPT "Two" MESSAGE "Choice two"
MENU TO nChoice
//
IF nChoice == 0
   EXIT
ELSEIF nChoice == 1
   Proc1()
ELSEIF nChoice == 2
   Proc2()
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

@...PROMPT, MENU TO, SET WRAP

SET OPTIMIZE command

Change the setting that determines whether to optimize using the open orders when processing a filtered database file

Syntax

```
SET OPTIMIZE ON | OFF | (<lToggle>)
```

Arguments

ON enables optimization.

OFF disables optimization.

<lToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Note: The initial default of this setting depends on the RDD.

Description

For RDDs that support optimization, such as DBFCDX, SET OPTIMIZE determines whether to optimize filters based on the orders open in the current work area. If this flag is ON, the RDD will optimize the search for records that meet the filter condition to the fullest extent possible, minimizing the need to read the actual data from the database file.

If this flag is OFF, the RDD will not optimize.

Examples

- The following example enables optimization for the Inventor database file using the SET OPTIMIZE command:

```
USE Inventor NEW VIA "DBFCDX"
SET OPTIMIZE ON
```

Files Library is CLIPPER.LIB.

See Also DBSETFILTER(), SET FILTER

SET ORDER command

Select the controlling order

Syntax

```
SET ORDER TO [<nOrder> | [TAG <cOrderName>]
   [IN <xcOrderBagName>]]
```

Arguments

TAG is an optional clause that provides compatibility with RDDs that access multiple-order order bags. You must use this keyword anytime you specify *<cOrderName>*.

<cOrderName> is the name of an order, a logical arrangement of a database according to a keyed pair. This order will become the controlling order in the order list. If you specify *<cOrderName>*, you must use the keyword TAG.

Note: This differs from dBASE and FoxPro where TAG is totally optional.

<nOrder> is the number of the target order in the order list. You may represent the order as an integer or as a character string enclosed in quotes.

IN <xcOrderBagName> is the name of a disk file containing one or more orders. You may specify *<xcOrderBagName>* as the file name with or without the path name or appropriate extension. If you do not include the extension as part of *<xcOrderBagName>*, CA-Clipper uses the default extension of the current RDD.

Description

When you SET ORDER TO a new controlling order (index), all orders are properly updated when you either append or edit records. This is true even if you SET ORDER TO 0. After a change of controlling order, the record pointer still points to the same record.

SET ORDER TO 0 restores the database access to natural order, but leaves all orders open. SET ORDER TO with no arguments closes all orders and empties the order list

Though you may use *<cOrderName>* or *<nOrder>* to specify the target order, *<nOrder>* is only provided for compatibility with earlier versions of CA-Clipper. Using *<cOrderName>* is a surer way of accessing the correct order in the order list.

If you supply *<xcOrderBagName>*, only the orders belonging to *<xcOrderBagName>* in the order list are searched. Usually you need not specify *<xcOrderBagName>* if you use unique order names throughout an application.

To determine which order is the controlling order use the ORDSETFOCUS() function.

In RDDs that support production or structural indices (e.g., DBFCDX), if you specify a tag but do not specify an order bag, the tag is created and added to the index. If no production or structural index exists, it will be created and the tag will be added to it. When using RDDs that support multiple order bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling order. If you do not specify a controlling order, the data file will be viewed in natural order.

SET ORDER can open orders in a network environment instead of the INDEX clause of the USE command. Generally, specify USE, and then test to determine whether the USE succeeded. If it did succeed, open the associated orders with SET ORDER. See the example below.

Examples

```
USE Customer NEW
IF (! NETERR())
   SET ORDER TO Customer
ENDIF

SET ORDER TO "CuAcct"          // CuAcct is an Order in Customer
```

See Also

INDEX, INDEXORD(), SEEK, SET INDEX, USE

SET PATH command

Specify the CA-Clipper search path for opening files

Syntax

```
SET PATH TO [<xcPathspec list>]
```

Arguments

TO <xcPathspec list> identifies the paths CA-Clipper uses when searching for a file not found in the current directory. You can specify it as a literal list or as a character expression enclosed in parentheses. The list of paths can be separated by commas or semicolons. However, continuation of a SET PATH command line with a semicolon is not supported unless *<xcPathspec>* is specified as a character expression enclosed in parentheses.

Description

SET PATH allows commands and functions that open database and associated files to find and open existing files in another drive and/or directory. It does this by specifying a path list to search if a referenced file cannot be found in the DEFAULT or specified directory. Note that memo and low-level file functions respect neither the DEFAULT nor the path setting.

A path is a pointer to a directory. It consists of an optional drive letter and colon, followed by a list of directories from the root to the desired directory separated by backslash (\) characters. A path list is the sequence of paths to search, each separated by a comma or semicolon.

When you attempt to access a file, CA-Clipper first searches the default drive and directory. The default disk drive and directory are established by DOS when your CA-Clipper application is loaded or, during execution, by SET DEFAULT. If the file is not found, CA-Clipper then searches each path in the specified path list until the first occurrence of the file is found.

To create new files in another drive or directory, use SET DEFAULT TO *<xcPathspec>* or explicitly declare the path when specifying a new file name.

SET PATH TO with no argument releases the path list and CA-Clipper searches only the DEFAULT directory.

Examples

- This example is a typical PATH command:

```
SET PATH TO A:\INVENTORY;B:\VENDORS
```

- This example configures a path setting at runtime by passing a DOS environment variable to a program, retrieving its value with GETENV(), and then setting path with this value. For example, in DOS:

```
SET CLIP_PATH=C:\APPS\DATA,C:\APPS\PROGS
```

  Later in the configuration section of your application program:

```
SET PATH TO (GETENV("CLIP_PATH"))
```

Files Library is CLIPPER.LIB.

See Also CURDIR(), SET DEFAULT

SET PRINTER command

Toggle echo of console output to the printer or set the destination of printed output

Syntax

```
SET PRINTER on | OFF | <xlToggle>
SET PRINTER TO [<xcDevice> | <xcFile> [ADDITIVE]]
```

Arguments

ON echoes console output to the printer.

OFF suppresses the printing of console output.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

TO <xcDevice> identifies the name of the device where all subsequent printed output will be sent. You can specify a device name as a literal character string or a character expression enclosed in parentheses. Additionally, a device can be either local or network. SETting PRINTER TO a nonexistent device creates a file with the name of the device. When specifying device names, do not use a trailing colon.

TO <xcFile> identifies the name of the output file. You can specify the file name as a literal string or as a character expression enclosed in parentheses. If a file extension is not specified, .prn is assumed.

ADDITIVE causes the specified output file to be appended to instead of overwritten. If ADDITIVE is not specified, an existing output file is truncated before new information is written to it. The ADDITIVE clause is only meaningful when SETting PRINTER TO an output file.

If SET PRINTER TO is specified with no arguments, the currently specified device or file is closed and the default destination is then reselected.

Description

SET PRINTER, like many other SET commands, has two basic forms with each having its own functionality. The on | OFF form of SET PRINTER controls whether the output of console commands is echoed to the printer. Console commands generally do not specify row and column coordinates. All of these commands, except ? | ??, have a TO PRINTER clause that also directs output to the printer. Output from console commands is displayed to the screen unless CONSOLE is OFF. Be aware that @...SAYs are not affected by SET PRINTER ON. To send them to the printer, use SET DEVICE TO PRINTER instead.

SET PRINTER TO determines the destination of output from all commands and functions that send output to the printer. This includes @...SAYs if DEVICE is SET TO PRINTER. Output can be sent to a device or to a file. If the destination is a device, the following names are valid: LPT1, LPT2, LPT3 (all parallel ports), COM1, and COM2 (serial ports), CON and PRN. The default device is PRN.

If the destination is a file, it is created in the current DEFAULT directory. If a file with the same name exists in the same location, it is overwritten by the new file without warning. All subsequent output to the printer is then written to this file until the file is closed using SET PRINTER TO with no argument.

Use SET PRINTER TO for:

- Managing multiple printers by swapping ports
- Directing output to a file for printing later or for transferring to a remote computer via telecommunications
- Emptying the printer spooler and resetting the default device

Notes

- ***Compatibility:*** CA-Clipper does not support the syntax SET PRINTER TO \\SPOOLER or \\CAPTURE. Specifying SET PRINTER with either of these options creates the files Spooler.prn or Capture.prn. The symbols \\ are ignored.
- ***End of file marks:*** When printer output is redirected to a file, an end of file mark (CHR(26)) is not written when the file is closed. To terminate a file with an end of file mark, issue a ?? CHR(26) just before the SET PRINTER command that closes the file.
- ***Networking:*** For some networks, the workstation's printer should first be redirected to the file server (usually by running the network spooler program).

Examples

- This example echoes the output of the ? command to printer, suppressing the console screen display by SETting CONSOLE OFF:

```
USE Customer NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE !EOF()
    ? Customer->Name, Customer->Phone
    SKIP
ENDDO
EJECT
SET PRINTER OFF
SET CONSOLE ON
CLOSE
RETURN
```

- This example directs printer output to LPT1 and empties the print spooler upon completion:

```
SET PRINTER TO LPT1
<Printing statements>...
SET PRINTER TO          // Empty the print spooler
```

- This example sends printer output to a text file, overwriting an existing file with the same name:

```
SET PRINTER TO Prnfile.txt
SET DEVICE TO PRINTER
SET PRINTER ON
//
@ 0, 0 SAY "This goes to Prnfile.txt"
? "So will this!"
//
SET DEVICE TO SCREEN
SET PRINTER OFF
SET PRINTER TO          // Close the print file
```

Files Library is CLIPPER.LIB.

See Also @...SAY, EJECT, SET CONSOLE, SET DEVICE, SETPRC()

SET PROCEDURE* command

Compile procedures and functions into the current object (.OBJ) file

Syntax

```
SET PROCEDURE TO [<idProgramFile>[.<ext>]]
```

Arguments

TO <idProgramFile> is the name of the procedure file to compile into the current object file. It can optionally include a path and/or drive designator.

<ext> is the optional extension of the procedure. If not specified, .prg is assumed.

SET PROCEDURE TO with no argument is ignored.

Description

SET PROCEDURE directs the compiler to compile all procedures and user-defined functions declared within the specified procedure file into the current object (.OBJ) file.

SET PROCEDURE is a compatibility command and not recommended. It has been superseded by other facilities more appropriate to the compiled environment (e.g., the compiler script (.clp)) file.

See the CA-Clipper "Compiler" chapter in the *Programming and Utilities Guide* for a full discussion of program architecture and configuration.

See Also

#include, DO*, FUNCTION, PROCEDURE, RETURN

SET RELATION command

Relate two work areas by a key value or record number

Syntax

```
SET RELATION TO [<expKey> | <nRecord> INTO <xcAlias>]
   [, [TO] <expKey2> | <nRecord2> INTO <xcAlias2>...]
   [ADDITIVE]
```

Arguments

TO <expKey> is an expression that performs a SEEK in the child work area each time the record pointer moves in the parent work area. For this to work, the child work area must have an index in USE.

TO <nRecord> is an expression that performs a GOTO to the matching record number in the child work area each time the record pointer moves in the parent work area. If *<nRecord>* evaluates to RECNO(), the relation uses the parent record number to perform a GOTO to the same record number in the child work area. For a numeric expression type of relation to execute correctly, the child work area must not have an index in USE.

INTO <xcAlias> identifies the child work area and can be specified either as the literal alias name or as a character expression enclosed in parentheses.

ADDITIVE adds the specified child relations to existing relations already set in the current work area. If this clause is not specified, existing relations in the current work area are released before the new child relations are set.

SET RELATION TO with no arguments releases all relations defined in the current work area.

Description

SET RELATION is a database command that links a parent work area to one or more child work areas using a key expression, record number, or numeric expression. Each parent work area can be linked to as many as eight child work areas. A relation causes the record pointer to move in the child work area in accordance with the movement of the record pointer in the parent work area. If no match is found in the child work area, the child record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

The method of linking the parent and child work areas depends on the type of *<expKey>* and presence of an active index in the child work area. If the child work area has an active index, the lookup is a standard SEEK. If the child work area does not have an active index and the type of *<expKey>* is numeric, a GOTO is performed in the child work area instead.

Notes

- ***Cyclical relations:*** Do not relate a parent work area to itself either directly or indirectly.
- ***Soft seeking:*** SET RELATION does not support SOFTSEEK and always behaves as if SOFTSEEK is OFF even if SOFTSEEK is ON. This means that if a match is not found in the child work area, the child record pointer is always positioned to LASTREC() + 1.
- ***Record number relations:*** To relate two work areas based on matching record numbers, use RECNO() for the SET RELATION TO expression and make sure the child work area has no active indexes.

Examples

- This example relates three work areas in a multiple parent-child configuration with Customer related to both Invoices and Zip:

```
USE Invoices INDEX Invoices NEW
USE Zip INDEX Zipcode NEW
USE Customer NEW
SET RELATION TO CustNum INTO Invoices, Zipcode INTO Zip
LIST Customer, Zip->City, Invoices->Number, ;
        Invoices->Amount
```

- Sometime later, you can add a new child relation using the ADDITIVE clause, like this:

```
USE BackOrder INDEX BackOrder NEW
SELECT Customer
SET RELATION TO CustNum INTO BackOrder ADDITIVE
```

Files

Library is CLIPPER.LIB.

See Also

DBRELATION(), DBRSELECT(), FOUND(), RECNO(), SET INDEX, SET ORDER, SET SOFTSEEK

SET SCOPE command

Change the top and/or bottom boundaries for scoping key values in the controlling order

Syntax

```
SET SCOPE TO [<expNewTop> [, <expNewBottom>]]
```

Arguments

<expNewTop> is the top range of key values that will be included in the controlling order's current scope. *<expNewTop>* can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.

<expNewBottom> is the bottom range of key values that will be included in the controlling order's current scope. *<expNewBottom>* can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.

Note: If *<expNewBottom>* is not specified, *<expNewTop>* is taken for *both* the top and bottom range values.

Description

SET SCOPE, when used with no arguments, clears the top and bottom scopes; this is equivalent to ORDSCOPE(0, NIL) followed by ORDSCOPE(1, NIL). If *<expNewTop>* is specified alone, SET SCOPE sets the top and bottom scope to the indicated value (i.e., ORDSCOPE(0, *<expNewTop>*) followed by ORDSCOPE(1, *<expNewTop>*). If both *<expNewTop>* and *<expNewBottom>* are specified, SET SCOPE sets the top and bottom scope as indicated (i.e., ORDSCOPE(0, *<expNewTop>*) followed by ORDSCOPE(1, *<expNewBottom>*). Refer to the ORDSCOPE() function for more information.

Examples

- The following example illustrates the SET SCOPE command:

```
USE Inventor NEW
INDEX ON PartNo TO Parts

SET SCOPE TO 1750, 2000
// Only part numbers between 1750 and 2000 will be used
```

Files

Library is CLIPPER.LIB.

See Also

ORDSCOPE()

SET SCOPEBOTTOM command

Change the bottom boundary for scoping key values in the controlling order

Syntax

```
SET SCOPEBOTTOM TO [<expNewBottom>]
```

Arguments

<expNewBottom> is the bottom range of key values that will be included in the controlling order's current scope. *<expNewBottom>* can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.

Description

SET SCOPEBOTTOM, when used with the *<expNewBottom>* argument, is functionally equivalent to ORDSCOPE(1, *<expNewBottom>*). SET SCOPEBOTTOM, when used with no argument, is functionally equivalent to ORDSCOPE(1, NIL). Refer to the ORDSCOPE() function for more information.

Examples

- The following example illustrates the SET SCOPEBOTTOM command:

```
USE Inventor NEW
INDEX ON PartNo TO Parts

SET SCOPEBOTTOM TO 1000
// Only part numbers less than or equal to 1000
// will be used.
```

Files

Library is CLIPPER.LIB.

See Also

ORDSCOPE()

SET SCOPETOP command

Change the top boundary for scoping key values in the controlling order

Syntax

```
SET SCOPETOP TO [<expNewTop>]
```

Arguments

<expNewTop> is the top range of key values that will be included in the controlling order's current scope. *<expNewTop>* can be an expression that matches the data type of the key expression in the controlling order or a code block that returns the correct data type.

Description

SET SCOPETOP, when used with the *<expNewTop>* argument, is functionally equivalent to ORDSCOPE(0, *<expNewTop>*). SET SCOPETOP, when used with no argument, is functionally equivalent to ORDSCOPE(0, NIL). Refer to the ORDSCOPE() function for more information.

Examples

- The following example illustrates the SET SCOPETOP command:

```
USE Inventor NEW
INDEX ON PartNo TO Parts

SET SCOPETOP TO 1000
// Only part numbers greater than or equal to 1000
// will be used.
```

Files

Library is CLIPPER.LIB.

See Also

ORDSCOPE()

SET SCOREBOARD command

Toggle the message display from READ or MEMOEDIT()

Syntax

```
SET SCOREBOARD ON | off | <xlToggle>
```

Arguments

ON allows the display of messages from READ and MEMOEDIT() on line zero of the screen.

OFF suppresses these messages.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET SCOREBOARD controls whether or not messages from READ and MEMOEDIT() display on line zero. When SCOREBOARD is ON, READ displays messages for RANGE errors, invalid dates, and insert status. MEMOEDIT() displays an abort query message and the insert status.

To suppress the automatic display of these messages, SET SCOREBOARD OFF.

Files

Library is CLIPPER.LIB.

See Also

@...GET, MEMOEDIT(), READ

SET SOFTSEEK command

Toggle relative seeking

Syntax

```
SET SOFTSEEK on | OFF | <xlToggle>
```

Arguments

ON causes the record pointer to be moved to the next record with a higher key after a failed index search.

OFF causes the record pointer to be moved to EOF() after a failed index search.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET SOFTSEEK enables relative seeking, a method of searching an index and returning a record even if there is no match for a specified key.

When SOFTSEEK is ON and a match for a SEEK is not found, the record pointer is set to the next record in the index with a higher key value than the SEEK argument. Records are not visible because SET FILTER and/or SET DELETED are skipped when searching for the next higher key value. If there is no record with a higher key value, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). FOUND() returns true (.T.) only if the record is actually found. It never returns true (.T.) for a relative find.

When SOFTSEEK is OFF and a SEEK is unsuccessful, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

Notes

- ***SET RELATION:*** SET RELATION ignores SOFTSEEK updating the record pointer in all linked child work areas as if SOFTSEEK is OFF.

Examples

- This example illustrates the possible results of a SEEK with SET SOFTSEEK ON:

```
SET SOFTSEEK ON
USE Salesman INDEX Salesman NEW
ACCEPT "Enter Salesman: " TO cSearch
SEEK cSearch
DO CASE
CASE FIELD->Salesman = cSearch
    ? "Match found:", FOUND(), EOF(), FIELD->Salesman
CASE !EOF()
    ? "Soft match found:", FOUND(), EOF(), ;
              FIELD->Salesman
OTHERWISE
    ? "No key matches:", FOUND(), EOF(), FIELD->Salesman
ENDCASE
```

Files Library is CLIPPER.LIB.

See Also FOUND(), SEEK, SET INDEX, SET ORDER, SET RELATION

SET TYPEAHEAD command

Set the size of the keyboard buffer

Syntax

```
SET TYPEAHEAD TO <nKeyboardSize>
```

Arguments

TO <nKeyboardSize> specifies the number of keystrokes the keyboard buffer can hold from a minimum of zero to a maximum of 4096. The default size of the keyboard buffer is machine-dependent but 16 is the minimum size.

Description

SET TYPEAHEAD defines the size of the CA-Clipper keyboard buffer that caches keystrokes input directly by the user. SET TYPEAHEAD, however, does not affect the number of characters that can be stuffed programmatically using the KEYBOARD command. When executed, SET TYPEAHEAD clears the keyboard buffer and sets the size to *<nKeyboardSize>*.

When TYPEAHEAD is SET TO zero, keyboard polling is suspended. An explicit request for keyboard input, however, will temporarily enable the keyboard and read any pending keystrokes from the BIOS buffer. Calling NEXTKEY() constitutes such an explicit request. NEXTKEY() reads any pending keystrokes from the BIOS buffer and returns the INKEY() value of the first keystroke read, or zero if no keystrokes are pending.

Files

Library is CLIPPER.LIB.

See Also

ALTD(), CLEAR TYPEAHEAD, INKEY(), KEYBOARD, NEXTKEY(), SETCANCEL()

SET UNIQUE* command

Toggle inclusion of non-unique keys into an index

Syntax

```
SET UNIQUE on | OFF | <xlToggle>
```

Arguments

ON causes index files to be created with a uniqueness attribute.

OFF causes index files to be created without a uniqueness attribute.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET UNIQUE is a database command that controls whether indexes are created with uniqueness as an attribute. With UNIQUE ON, new indexes are created including only unique keys. This is the same as creating an index with the INDEX...UNIQUE command.

If, during the creation or update of an unique index, two or more records are encountered with the same key value, only the first record is included in the index. When the unique index is updated, REINDEXed, or PACKed, only unique records are maintained, without regard to the current SET UNIQUE value.

Changing key values in a unique index has important implications. First, if a unique key is changed to the value of a key already in the index, the changed record is lost from the index. Second, if there is more than one instance of a key value in a database file, changing the visible key value does not bring forward another record with the same key until the index is rebuilt with REINDEX, PACK, or INDEX...UNIQUE.

With UNIQUE OFF, indexes are created with all records in the index. Subsequent updates to the database files add all key values to the index independent of the current UNIQUE SETting.

SET UNIQUE is a compatibility command not recommended. It is superseded by the UNIQUE clause of the INDEX command.

Files

Library is CLIPPER.LIB.

See Also

DBCREATEINDEX(), INDEX, PACK, REINDEX, SEEK

SET VIDEOMODE command

Change the current video mode of the current application.

Syntax

```
SET VIDEOMODE TO <nVideoMode>
```

Arguments

nVideoMode is a numeric value representing a particular video mode.

Description

SET VIDEOMODE changes the current display to text mode and different graphic modes. There are two modes supported by CA-Clipper: LLG_VIDEO_TEXT and LLG_VIDEO_VGA_640_480_16.

Notes

When switching from LLG_VIDEO_TEXT to LLG_VIDEO_VGA_640_480_16, all displayed text lines are converted to the equivalent graphic display. This conversion does not happen when switching back to LLG_VIDEO_TEXT mode.

If you wish to have a part of your application switch to LLG_VIDEO_VGA_640_480_16 mode and clear the screen, issue the CLS command before switching modes.

Files Libraries are CLIPPER.LIB and LLIBG.LIB.

See Also GMODE()

*SET WRAP command

Toggle wrapping of the highlight in menus

Syntax

```
SET WRAP on | OFF | <xlToggle>
```

Arguments

ON enables the highlight to wrap around when navigating a lightbar menu.

OFF disables wrapping when navigating a lightbar menu.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET WRAP is a menu command that toggles wrapping of the highlight in an @...PROMPT menu from the first menu item to the last menu item and vice versa. When WRAP is ON and the last menu item is highlighted, Right arrow or Down arrow moves the highlight to the first menu item. Likewise, when the first menu item is highlighted, Left arrow or Up arrow moves the highlight to the last menu item.

When WRAP is OFF, pressing Up arrow or Left arrow from the first menu item or Down arrow or Right arrow from the last menu item does nothing.

Files Library is CLIPPER.LIB.

See Also @...PROMPT, MENU TO, SET MESSAGE

SET() function

Inspect or change a system setting

Syntax

```
SET(<nSpecifier>, [<expNewSetting>], [<lOpenMode>])
    → CurrentSetting
```

Arguments

<nSpecifier> is a numeric value that identifies the setting to be inspected or changed. *<nSpecifier>* should be supplied as a manifest constant (see below).

<expNewSetting> is an optional argument that specifies a new value for the *<nSpecifier>*. The type of *<expNewSetting>* depends on *<nSpecifier>*.

<lOpenMode> is a logical value that indicates whether or not files opened for the following settings,

_SET_ALTFILE, _SET_PRINTFILE, _SET_EXTRAFILE

should be truncated or opened in append mode. A value of false (.F.) means the file should be truncated. A value of true (.T.) means the file should be opened in append mode. In either case, if the file does not exist, it is created.

If this argument is not specified, the default is append mode.

Returns

SET() returns the current value of the specified setting.

Description

SET() is a system function that lets you inspect or change the values of the CA-Clipper system settings. For information on the meaning and legal values for a particular setting, refer to the associated command or function.

Use a manifest constant to specify the setting to be inspected or changed. These constants are defined in a header file called Set.ch. This header file should be included at the top of any source file which uses SET().

Set.ch also defines a constant called _SET_COUNT. This constant is equal to the number of settings that can be changed or inspected with SET(), allowing the construction of a generic function that preserves all settings (see example below).

Note: The numeric values of the manifest constants in Set.ch are version-dependent and should never be used directly; the manifest constants should always be used.

If *<nSpecifier>* or *<expNewSetting>* is invalid, the call to SET() is ignored.

Set Values Defined in Set.ch

Constant	Value Type	Associated Command or Function
_SET_EXACT	Logical	SET EXACT
_SET_FIXED	Logical	SET FIXED
_SET_DECIMALS	Numeric	SET DECIMALS
_SET_DATEFORMAT	Character	SET DATE
_SET_EPOCH	Numeric	SET EPOCH
_SET_PATH	Character	SET PATH
_SET_DEFAULT	Character	SET DEFAULT
_SET_EXCLUSIVE	Logical	SET EXCLUSIVE
_SET_SOFTSEEK	Logical	SET SOFTSEEK
_SET_UNIQUE	Logical	SET UNIQUE
_SET_DELETED	Logical	SET DELETED
_SET_CANCEL	Logical	SETCANCEL()
_SET_DEBUG	Numeric	ALTD()
_SET_COLOR	Character	SETCOLOR()
_SET_CURSOR	Numeric	SETCURSOR()
_SET_CONSOLE	Logical	SET CONSOLE
_SET_ALTERNATE	Logical	SET ALTERNATE
_SET_ALTFILE	Character	SET ALTERNATE TO
_SET_DEVICE	Character	SET DEVICE
_SET_PRINTER	Logical	SET PRINTER
_SET_PRINTFILE	Character	SET PRINTER TO
_SET_MARGIN	Numeric	SET MARGIN
_SET_BELL	Logical	SET BELL
_SET_CONFIRM	Logical	SET CONFIRM
_SET_ESCAPE	Logical	SET ESCAPE
_SET_INSERT	Logical	READINSERT()
_SET_EXIT	Logical	READEXIT()
_SET_INTENSITY	Logical	SET INTENSITY
_SET_SCOREBOARD	Logical	SET SCOREBOARD
_SET_DELIMITERS	Logical	SET DELIMITERS
_SET_DELIMCHARS	Character	SET DELIMITERS TO
_SET_WRAP	Logical	SET WRAP
_SET_MESSAGE	Numeric	SET MESSAGE
_SET_MCENTER	Logical	SET MESSAGE

Note: _SET_EXTRAFILE and _SET_SCROLLBREAK have no corresponding commands. _SET_EXTRAFILE lets you specify an additional alternate file, and _SET_SCROLLBREAK lets you toggle the interpretation of Ctrl+S.

Examples

- In this example a user-defined function preserves or restores all global settings. This function might be used on entry to a subsystem to ensure that the subsystem does not affect the state of the program that called it:

```
#include "Set.ch"
//

FUNCTION SetAll( aNewSets )
   LOCAL aCurrentSets[_SET_COUNT], nCurrent
   IF ( aNewSets != NIL ) // Set new and return current
      FOR nCurrent := 1 TO _SET_COUNT
         aCurrentSets[nCurrent] := ;
            SET(nCurrent, aNewSets[nCurrent])
      NEXT
   ELSE        // Just return current
      FOR nCurrent := 1 TO _SET_COUNT
         aCurrentSets[nCurrent] := SET(nCurrent)
      NEXT
   ENDIF
   RETURN (aCurrentSets)
```

Files

Library is CLIPPER.LIB, header file is Set.ch.

SETBLINK() function

Toggle asterisk (*) interpretation in the SETCOLOR() string between blinking and background intensity

Syntax

```
SETBLINK([<lToggle>]) → lCurrentSetting
```

Arguments

<lToggle> changes the meaning of the asterisk (*) character when it is encountered in a SETCOLOR() string. Specifying true (.T.) sets character blinking on, and false (.F.) sets background intensity. The default is true (.T.).

Returns

SETBLINK() returns the current setting as a logical value.

Description

SETBLINK() is an environment function that toggles the blinking/background intensity attribute and reports the current state of SETBLINK(). When SETBLINK() is on, characters written to the screen can be made to blink by including an asterisk (*) in a color string passed to SETCOLOR(). When SETBLINK() is off, the asterisk (*) causes the background color to be intensified instead. Thus, blinking and background intensity attributes are not available at the same time.

Note: This function is meaningful only on the IBM PC or compatible computers with CGA, EGA, or VGA display hardware.

Examples

- This example saves the current SETBLINK() state before passing control to a user-defined function. Upon return, SETBLINK() is restored to its original value:

```
lOldBlink := SETBLINK()
MyFunc()
SETBLINK(lOldBlink)
```

Files

Library is CLIPPER.LIB.

See Also

SET COLOR, SETCOLOR()

SETCANCEL() function

Toggle Alt+C and Ctrl+Break as program termination keys

Syntax

```
SETCANCEL([<lToggle>]) → lCurrentSetting
```

Arguments

<lToggle> changes the availability of Alt+C and Ctrl+Break as termination keys. Specifying true (.T.) allows either of these keys to terminate an application and false (.F.) disables both keys. The default is true (.T.).

Returns

SETCANCEL() returns the current setting as a logical value.

Description

SETCANCEL() is a keyboard function that toggles the state of the termination keys, Alt+C and Ctrl+Break, and reports the current state of SETCANCEL(). Use SETCANCEL() when you want to suppress a user's ability to terminate a program without using the specified method.

Note that if Alt+C or Ctrl+Break is redefined with SET KEY, the SET KEY definition takes precedence even if SETCANCEL() returns true (.T.).

Warning! *When SETCANCEL() has been set to false (.F.), the user cannot terminate a runaway program unless you provide an alternative escape mechanism.*

Examples

- This example provides an escape route from a wait state with SETCANCEL() set off:

```
#define K_ALTC 302
//
SETCANCEL(.F.)              // Disable termination keys
SET KEY K_ALTC TO AltC      // Redefine Alt-C
.
. <statements>
.
RETURN

FUNCTION AltC
    LOCAL cScreen, nChoice, cLastColor := ;
          SETCOLOR("W/B, N/G")
    //
    SAVE SCREEN TO cScreen
    @ 6, 20 CLEAR TO 9, 58
    @ 6, 20 TO 9, 58 DOUBLE
    @ 7, 26 SAY "Alt-C: Do you want to quit?"
    @ 8, 35 PROMPT " Yes "
    @ 8, 41 PROMPT " No "
    MENU TO nChoice
    SETCOLOR(cLastColor)
    RESTORE SCREEN FROM cScreen
    //
    IF nChoice = 1
       QUIT
    ENDIF
    //
    RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also ALTD(), SET ESCAPE, SET KEY, SETKEY()

SETCOLOR() function

Return the current colors and optionally set new colors

Syntax

SETCOLOR([<*cColorString*>]) → *cColorString*

Arguments

<cColorString> is a character string containing a list of color attribute settings for subsequent screen painting. The following is a list of settings and related scopes:

Color Settings

Setting	Scope
Standard	All screen output commands and functions
Enhanced	GETs and selection highlights
Border	Border around screen, not supported on EGA and VGA
Background	Not supported
Unselected	Unselected GETs

Each setting is a foreground and background color pair separated by the slash (/) character and followed by a comma. All settings are optional. If a setting is skipped, its previous value is retained with only new values set. Settings may be skipped within the list or left off the end as illustrated in the examples below.

Returns

SETCOLOR() returns the current color settings as a character string.

Description

SETCOLOR() is a screen function that saves the current color setting or sets new colors for subsequent screen painting. A color string is made from several color settings, each color corresponding to different regions of the screen. As stated above, each setting is made up of a foreground and background color. Foreground defines the color of characters displayed on the screen. Background defines the color displayed behind the character. Spaces and nondisplay characters display as background.

In CA-Clipper, the settings that define color behavior are:

Standard: The standard setting governs all console, full-screen, and interface commands and functions when displaying to the screen. This includes commands such as @...PROMPT, @...SAY, and ?, as well as functions such as ACHOICE(), DBEDIT(), and MEMOEDIT().

Enhanced: The enhanced setting governs highlighted displays. This includes GETs with INTENSITY ON, and the MENU TO, DBEDIT(), and ACHOICE() selection highlight.

Border: The border is an area around the screen that cannot be written to.

Background: The background is not supported.

Unselected: The unselected setting indicates input focus by displaying the current GET in the enhanced color while other GETs are displayed in the unselected color.

In addition to colors, foreground settings can have high intensity and/or blinking attributes. With a monochrome display, high intensity enhances brightness of painted text. With a color display, high intensity changes the hue of the specified color. For example, "N" displays foreground text as black where "N+" displays the same text as gray. High intensity is denoted by "+". The blinking attribute causes the foreground text to flash on and off at rapid intervals. Blinking is denoted with "*". The attribute character can occur anywhere in the setting string, but is always applied to the foreground color regardless where it occurs. See SETBLINK() for additional information.

The following colors are supported:

List of Colors

Color	Letter	Monochrome
Black	N, Space	Black
Blue	B	Underline
Green	G	White
Cyan	BG	White
Red	R	White
Magenta	RB	White
Brown	GR	White
White	W	White
Gray	N+	White
Bright Blue	B+	Bright Underline
Bright Green	G+	Bright White
Bright Cyan	BG+	Bright White
Bright Red	R+	Bright White
Bright Magenta	RB+	Bright White
Yellow	GR+	Bright White
Bright White	W+	Bright White
Black	U	Underline
Inverse Video	I	Inverse Video
Blank	X	Blank

Notes

- ***Arguments are not specified:*** Unlike SET COLOR TO, SETCOLOR() with no argument does not restore colors to their default values.
- ***Color numbers:*** SETCOLOR() supports color letter combinations, but not color number combinations.

Examples

- This example assigns the current color setting to the variable, cColor:

```
cColor:= SETCOLOR()
```

- This example uses SETCOLOR() to save the current color setting and set a new one.

```
cNewColor:= "BR+/N, R+/N"
cOldColor:= SETCOLOR(cNewColor)
```

- This example uses SET COLOR TO to reset the default colors:

```
SET COLOR TO
? SETCOLOR()          // Result: W/N, N/W, N, N, N/W
```

- These two examples specify SETCOLOR() with missing settings:

```
// Settings left off the end
SETCOLOR("W/N, BG+/B")
//
// Settings skipped within the list
SETCOLOR("W/N, BG+/B,,,W/N")
```

- This example uses SETCOLOR() with ISCOLOR() to set the colors, depending on the screen type:

```
FUNCTION DefaultColors
   IF ISCOLOR()
      cForm := "W+/N, BG+/B,,,W/N"
      cDialog := "N/N+, BG+/B,,,N/N+"
      cAlert := "W+/R, BG+/B,,,W+/R"
   ELSE
      cForm := "W+/N, N/W,,,W/N"
      cDialog := "W+/N, N/W,,,W/N"
      cAlert := "W+/N, N/W,,,W/N"
   ENDIF
   RETURN NIL
```

Files

Library is CLIPPER.LIB.

See Also

ISCOLOR(), SET COLOR*, SET INTENSITY

SETCURSOR() function

Set the cursor shape

Syntax

```
SETCURSOR([<nCursorShape>]) → nCurrentSetting
```

Arguments

<nCursorShape> is a number indicating the shape of the cursor. For simpler coding, the Setcurs.ch header file provides descriptive names for the various cursor shapes as shown in the table below:

Cursor Shapes

Shape	Value	Setcurs.ch
None	0	SC_NONE
Underline	1	SC_NORMAL
Lower half block	2	SC_INSERT
Full block	3	SC_SPECIAL1
Upper half block	4	SC_SPECIAL2

Returns

SETCURSOR() returns the current cursor shape as a numeric value.

Description

SETCURSOR() is an environment function that controls the shape of the screen cursor. The actual shape is dependent on the current screen driver. The specified shapes appear on IBM PC and compatible computers. On other computers, the appearance may differ for each value specified.

SETCURSOR(0) is the same as SET CURSOR OFF, and any positive integer value of *<nCursorShape>* less than 5 is the same as SET CURSOR ON. The cursor will display as the selected shape.

Examples

- This example uses SETCURSOR() to turn on a full block cursor for the subsequent READ. When the READ terminates, SETCURSOR() turns off the cursor:

```
#include "Setcurs.ch"
//
USE Customer NEW
@ 10, 10 GET Customer->Name
@ 11, 10 GET Customer->Phone
//
SETCURSOR(SC_SPECIAL1)          // Change cursor to a block
READ
SETCURSOR(SC_NONE)              // Turn off cursor
```

Files Library is CLIPPER.LIB, header file is Setcurs.ch.

See Also SET CONSOLE, SET CURSOR, SETPOS()

SETKEY() function

Assign an action block to a key

Syntax

```
SETKEY(<nInkeyCode>, [<bAction>]) → bCurrentAction
```

Arguments

<nInkeyCode> is the INKEY() value of the key to be associated or queried.

<bAction> specifies a code block that is automatically executed whenever the specified key is pressed during a wait state.

Returns

SETKEY() returns the action block currently associated with the specified key, or NIL if the specified key is not currently associated with a block.

Description

SETKEY() is a keyboard function that sets or queries the automatic action associated with a particular key during a wait state. A wait state is any mode that extracts keys from the keyboard except for INKEY(), but including ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT. Up to 32 keys may be assigned at any one time. At startup, the system automatically assigns the F1 key to execute a procedure or user-defined function named Help.

When an assigned key is pressed during a wait state, the EVAL() function evaluates the associated *<bAction>* and the parameters, PROCNAME(), PROCLINE(), and READVAR(). It is, however, not necessary to list arguments when specifying *<bAction>* if you do not plan to use them within the action block.

SETKEY() is like the SET KEY command which associates a procedure invocation with a key.

Examples

- This code fragment associates an action block with a key, and then, after getting a key using INKEY(), executes it with the EVAL() function:

```
#include "Inkey.ch"
SETKEY(K_DN, {|cProc, nLine, cVar| MyProc(cProc, ;
                    nLine, cVar)})
.
. <statements>
.
DO WHILE .T.
   nKey := INKEY(0)
   DO CASE
   CASE (bAction := SETKEY(nKey)) != NIL
      EVAL(bAction, PROCNAME(), PROCLINE(), READVAR())
   CASE nKey = K_PGUP
      Previous()
   CASE nKey = K_PGDN
      Next()
   CASE nKey = K_ESC
      EXIT
   ENDCASE
ENDDO
```

Files Library is CLIPPER.LIB, header is Inkey.ch.

See Also EVAL(), INKEY(), SET KEY

SETMODE() function

Change display mode to a specified number of rows and columns

Syntax

SETMODE(<*nRows*>, <*nCols*>) → *lSuccess*

Arguments

<nRows> is the number of rows in the desired display mode.

<nCols> is the number of columns in the desired display mode.

Returns

SETMODE() returns true (.T.) if the mode change was successful; otherwise, it returns false (.F.).

Description

SETMODE() is an environment function that attempts to change the mode of the display hardware to match the number of rows and columns specified. The change in screen size is reflected in the values returned by MAXROW() and MAXCOL().

Note: In LLG_VIDEO_TXT mode, and when a VESA driver is present, it is possible to use the following values :
25,80 | 43,80 | 50,80 | 60,80 | 25,132 | 43,132 | 50,132 | 60,132

Examples

- This example switches to a 43-line display mode:

```
IF SETMODE(43, 80)
   ? "43-line mode successfully set"
ELSE
   ? "43-line mode not available"
ENDIF
```

- This example switches the video mode to regular text mode with 60 rows and 132 columns:

```
// Switch to text mode
SET VIDEOMODE( LLG_VIDEO_TXT )
// Set the video mode to the largest number of characters
SETMODE( 60,132 )
```

Files Library is CLIPPER.LIB.

See Also GMODE(), MAXCOL(), MAXROW(), SET VIDEOMODE

SETPOS() function

Move the cursor to a new position

Syntax

SETPOS(<*nRow*>, <*nCol*>) → <*nRow*>

Arguments

<nRow> and ***<nCol>*** define the new screen position of the cursor. These values may range from 0, 0 to MAXROW(), MAXCOL().

Returns

SETPOS() always returns <*nRow*>

Description

SETPOS() is an environment function that moves the cursor to a new position on the screen. After the cursor is positioned, ROW() and COL() are updated accordingly. To control the shape and visibility of the cursor, use the SETCURSOR() function.

Examples

- This example moves the cursor to a new position then displays a string to the screen using a console command, ??:

```
SETPOS(1, 1)
?? "Hello world"
```

Files Library is CLIPPER.LIB.

See Also COL(), ROW(), SET CURSOR, SETCURSOR()

SETPRC() function

Set PROW() and PCOL() values

Syntax

```
SETPRC(<nRow>, <nCol>) → NIL
```

Arguments

<nRow> is the new PROW() value.

<nCol> is the new PCOL() value.

Returns

SETPRC() always returns NIL.

Description

SETPRC() is a printer function that sends control codes to the printer without changing the tracking of the printhead position. When CA-Clipper prints, it updates the PCOL() value with the number of characters sent to the printer. There is no discrimination between printable or nonprintable characters. If, for example, a string of ten characters sent to the printer contains two characters interpreted by the printer as a control code, the CA-Clipper PCOL() value is incremented by ten, while the true printhead position is moved only by eight. This can lead to alignment problems. Using SETPRC(), you can compensate for control codes by resetting PCOL() as shown in the example below.

SETPRC() also suppresses page ejects when printing with @...SAY. This is important when the next row position is smaller than the current row and an EJECT has not been issued. In this situation, CA-Clipper issues an automatic page eject if the next row print position is less than the current PROW() value. Using SETPRC(), you can set PROW() to a number less than the current row, thus suppressing the automatic EJECT.

Examples

- This user-defined function, PrintCodes(), uses SETPRC() to send control codes to the printer without affecting PROW() and PCOL() values:

```
#include "Set.ch"
#define ITALICS_ON   CHR(27) + "I"
#define ITALICS_OFF  CHR(27) + "E"
//
SET DEVICE TO PRINTER
@ 12, 10 SAY "This is an"
@ PROW(), PCOL() + 2 SAY PrintCodes(ITALICS_ON) + ;
         "important"
@ PROW(), PCOL() + 2 SAY PrintCodes(ITALICS_OFF) + ;
         "meeting"
SET DEVICE TO SCREEN
RETURN

FUNCTION PrintCodes( cCtrlCode )
   LOCAL nRow, nCol, lPrinter
   lPrinter := SET(_SET_PRINTER, .T.)// SET PRINTER ON
   nRow:= PROW()                     // Save printhead position
   nCol:= PCOL()
   //
   ?? cCtrlCode                      // Send control code
   //
   SETPRC(nRow, nCol)
   SET(_SET_PRINTER, lPrinter)       // Restore printer setting
   RETURN ""                         // Return a null string
```

Files Library is CLIPPER.LIB.

See Also PCOL(), PROW(), SET DEVICE, SET()

SKIP command

Move the record pointer to a new position

Syntax

```
SKIP [<nRecords>] [ALIAS <idAlias> | <nWorkArea>]
```

Arguments

<nRecords> is a numeric expression specifying the number of records to move the record pointer from the current position. A positive value moves the record pointer forward and a negative value moves the record pointer backward.

ALIAS <idAlias> | <nWorkArea> specifies the alias name as a literal identifier or the work area as a numeric expression.

SKIP specified with no arguments moves the record pointer forward one record.

Description

SKIP moves the record pointer to a new position relative to the current position in the current work area and within the current filter, if there is one. SKIP is generally used for operations, such as reporting, that need to go to the next record in a database file.

If the alias clause is specified, the pointer can be moved in another work area without SELECTing that work area. SKIP can move either forward or backward. If there is no active index, SKIP moves the record pointer relative to the current position in the target database file. If there is an active index, SKIP moves the pointer relative to the current position in the index instead of the database file.

Attempting to SKIP forward beyond the end of file positions the record pointer at LASTREC() + 1, and EOF() returns true (.T.). Attempting to SKIP backward beyond the beginning of file moves the pointer to the first record, and BOF() returns true (.T.).

In a network environment, any record movement command, including SKIP, makes changes to the current work area visible to other applications if the current file is shared and the changes were made during an RLOCK(). To force an update to become visible without changing the current record position, use SKIP 0. If, however, the changes were made during an FLOCK(), visibility is not guaranteed until the lock is released, a COMMIT is performed, or the file is closed. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Examples

- This example uses SKIP with various arguments and shows their results:

```
USE Customers NEW
SKIP
? RECNO()                // Result: 2
SKIP 10
? RECNO()                // Result: 12
SKIP -5
? RECNO()                // Result: 7
```

- This example moves the record pointer in a remote work area:

```
USE Customers NEW
USE Invoices NEW
SKIP ALIAS Customers
```

- This example prints a report using SKIP to move the record pointer sequentially through the Customer database file:

```
LOCAL nLine := 99
USE Customers NEW
SET PRINTER ON
DO WHILE !EOF()
    IF nLine > 55
       EJECT
       nLine := 1
    ENDIF
       ? Customer, Address, City, State, Zip
       nLine++
       SKIP
    ENDDO
SET PRINTER OFF
CLOSE Customers
```

Files Library is CLIPPER.LIB.

See Also BOF(), COMMIT, DBSKIP, EOF(), GO, LOCATE, RECNO(), SEEK,

SORT command

Copy to a database (.dbf) file in sorted order

Syntax

```
SORT TO <xcDatabase> ON <idField1> [/[A | D][C]]
   [, <idField2> [/[A | D][C]]...]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

TO <xcDatabase> is the name of the target file for the sorted records and can be specified either as a literal file name or as a character expression enclosed in parentheses. Unless otherwise specified, the new file is assigned a (.dbf) extension.

ON <idField> is the sort key and must be a field variable.

/[A | D][C] specifies how *<xcDatabase>* is to be sorted. /A sorts in ascending order. /D sorts in descending order. /C sorts in dictionary order by ignoring the case of the specified character field. The default SORT order is ascending.

<scope> is the portion of the current database file to SORT. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to SORT within the given scope.

Description

SORT is a database command that copies records from the current work area to another database file in sorted order. CA-Clipper SORTs character fields in accordance with the ASCII value of each character within the string unless the /C option is specified. This option causes the database file to be sorted in dictionary order—capitalization is ignored. Numeric fields are sorted in numeric order, date fields are sorted chronologically, and logical fields are sorted with true (.T.) as the high value. Memo fields cannot be sorted.

SORT performs as much of its operation as possible in memory, and then, it spools to a uniquely named temporary disk file. This temporary file can be as large as the size of the source database file. Note also that a SORT uses up three file handles: the source database file, the target database file, and the temporary file.

In a network environment, you must lock the database file to be SORTed with FLOCK() or USE it EXCLUSIVEly.

Notes

- ***Deleted source records:*** If DELETED is OFF, SORT copies deleted records to the target database file; however, the deleted records do not retain their deleted status. No record is marked for deletion in the target file regardless of its status in the source file.

  If DELETED is ON, deleted records are not copied to the target database file. Similarly, filtered records are ignored during a SORT and are not included in the target file.

Examples

- This example copies a sorted subset of a mailing list to a smaller list for printing:

```
USE Mailing INDEX Zip
SEEK "900"
SORT ON LastName, FirstName TO Invite WHILE Zip = "900"
USE Invite NEW
REPORT FORM RsvpList TO PRINTER
```

Files Library is CLIPPER.LIB.

See Also ASORT(), FLOCK(), INDEX, USE

SOUNDEX() function

Convert a character string to "soundex" form

Syntax

```
SOUNDEX(<cString>) → cSoundexString
```

Arguments

<cString> is the character string to convert.

Returns

SOUNDEX() returns a four-digit character string in the form A999.

Description

SOUNDEX() is a character function that indexes and searches for sound-alike or phonetic matches. It is used in applications where the precise spelling of character keys is not known or where there is a high probability of misspelled names. Misspelling is common in real-time transaction systems where the data entry operator is receiving information over the telephone. SOUNDEX() works by bringing sound-alikes together under the same key value. Note, however, the soundex method is not absolute. Keys that are quite different can result in the same soundex value.

Examples

- This example builds an index using SOUNDEX() to create the key values. It then searches for a value found in the Salesman field:

```
USE Sales
INDEX ON SOUNDEX(Salesman) TO Salesman
SEEK SOUNDEX("Smith")
? FOUND(), Salesman          // Result: .T. Smith
```

- Here, a search is made for the same key as above but with a different spelling:

```
SEEK SOUNDEX("Smythe")
? FOUND(), Salesman          // Result: .T. Smith
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\SOUNDEX.C.

See Also

INDEX, LOCATE, SEEK, SET SOFTSEEK

SPACE() function

Return a string of spaces

Syntax

SPACE(<*nCount*>) → *cSpaces*

Arguments

<nCount> is the number of spaces to be returned, up to a maximum of 65,535 (64 K).

Returns

SPACE() returns a character string. If <*nCount*> is zero, SPACE() returns a null string ("").

Description

SPACE() is a character function that returns a specified number of spaces. It is the same as REPLICATE("", <*nCount*>). SPACE() can initialize a character variable before associating it with a GET. SPACE() can also pad strings with leading or trailing spaces. Note, however, that the PADC(), PADL(), and PADR() functions are more effective for this purpose.

Examples

- This example uses SPACE() to initialize a variable for data input:

```
USE Customer NEW
MEMVAR->Name = SPACE(LEN(Customer->Name))
@ 10,10 SAY "Customer Name" GET MEMVAR->Name
READ
```

Files Library is CLIPPER.LIB.

See Also PAD(), REPLICATE()

SQRT() function

Return the square root of a positive number

Syntax

```
SQRT(<nNumber>) → nRoot
```

Arguments

<nNumber> is a positive number for which the square root is to be computed.

Returns

SQRT() returns a numeric value calculated to double precision. The number of decimal places displayed is determined solely by SET DECIMALS regardless of SET FIXED. A negative *<nNumber>* returns zero.

Description

SQRT() is a numeric function used anywhere in a numeric calculation to compute a square root (e.g., in an expression that calculates standard deviation).

Examples

- These examples show various results of SQRT():

```
SET DECIMALS TO 5
//
? SQRT(2)                  // Result: 1.41421
? SQRT(4)                  // Result: 2.00000
? SQRT(4) ** 2             // Result: 4.00000
? SQRT(2) ** 2             // Result: 2.00000
```

Files

Library is CLIPPER.LIB.

See Also

SET DECIMALS, SET FIXED

STATIC statement

Declare and initialize static variables and arrays

Syntax

```
STATIC <identifier> [[:= <initializer>], ... ]
```

Arguments

<identifier> is the name of the variable or array to declare static. If the <*identifier*> is followed by square brackets ([ ]), it is created as an array. If the <*identifier*> is an array, the syntax for specifying the number of elements for each dimension can be array[<*nElements*>, <*nElements2*>,...] or array[<*nElements*>] [<*nElements2*>]... The maximum number of elements is 4096. The maximum number of dimensions is limited only by available memory.

<initializer> is the optional assignment of a value to a new static variable. An <*initializer*> for a static variable consists of the inline assignment operator (:=) followed by a compile-time constant expression consisting entirely of constants and operators or a literal array. If no explicit <*initializer*> is specified, the variable is given an initial value of NIL. In the case of an array, each element is NIL. Array identifiers cannot be given values with an <*initializer*>.

Note: The macro operator (&) cannot be used in a STATIC declaration statement.

Description

The STATIC statement declares variables and arrays that have a lifetime of the entire program but are only visible within the entity that creates them. Static variables are visible only within a procedure or user-defined function if declared after a PROCEDURE or FUNCTION statement. Static variables are visible to all procedures and functions in a program (.prg) file (i.e., have filewide scope) if they are declared before the first procedure or user-defined function definition in the file. Use the /N compiler option to compile a program with filewide variable scoping.

All static variables in a program are created when the program is first invoked, and all values specified in a static <*initializer*> are assigned to the variable before the beginning of program execution.

Declarations of static variables within a procedure or user-defined function must occur before any executable statement including PRIVATE, PUBLIC, and PARAMETERS. If a variable of the same name is declared FIELD, LOCAL, or MEMVAR within the body of a procedure or user-defined function, a compiler error occurs and no object (.OBJ) file is generated.

The maximum number of static variables in a program is limited only by available memory.

Notes

- ***Inspecting static variables within the Debugger:*** To access static variable names within the CA-Clipper debugger, you must compile program (.prg) files using the /B option so that static variable information is included in the object (.OBJ) file.
- ***Macro expressions:*** You may not refer to static variables within macro expressions or variables. If a static variable is referred to within a macro expression or variable, a private or public variable of the same name will be accessed instead. If no such variable exists, a runtime error will be generated.
- ***Memory files:*** Static variables cannot be SAVED to or RESTOREd from memory (.mem) files.
- ***Type of a static local variable:*** Since TYPE() uses the macro operator (&) to evaluate its argument, you cannot use TYPE() to determine the type of a local or static variable or an expression containing a local or static variable reference. The VALTYPE() function provides this facility by evaluating the function argument and returning the data type of its return value.

Examples

- This example declares static variables both with and without initializers:

```
STATIC aArray1[20, 10], aArray2[20][10]
STATIC cVar, cVar2
STATIC cString := "my string", var
STATIC aArray := {1, 2, 3}
```

- This example manipulates a static variable within a user-defined function. In this example, a count variable increments itself each time the function is called:

```
FUNCTION MyCounter( nNewValue )
   STATIC nCounter := 0       // Initial value assigned once
   IF nNewValue != NIL
      nCounter:= nNewValue    // New value for nCounter
   ELSE
      nCounter++              // Increment nCounter
   ENDIF
   RETURN nCounter
```

- This example demonstrates a static variable declaration that has filewide scope. In this code fragment, aArray is visible to both procedures that follow the declaration:

```
STATIC aArray := {1, 2, 3, 4}

FUNCTION One
   ? aArray[1]               // Result: 1
   RETURN NIL

FUNCTION Two
   ? aArray[3]               // Result: 3
   RETURN NIL
```

See Also

FUNCTION, LOCAL, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC

STORE* command

Assign a value to one or more variables

Syntax

```
STORE <exp> TO <idVar list>

OR

<idVar> = <exp>

OR

<idVar> := [ <idVar2> := ...] <exp>
```

Arguments

<exp> is a value of any data type that is assigned to the specified variables.

TO <idVar list> defines a list of one or more local, static, public, private, or field variables that are assigned the value *<exp>*. If any *<idVar>* is not visible or does not exist, a private variable is created and assigned *<exp>*.

Description

STORE assigns a value to one or more variables of any storage class. The storage classes of CA-Clipper variables are local, static, field, private, and public. STORE is identical to the simple assignment operators (=) and (:=). In fact, a STORE statement is preprocessed into an assignment statement using the inline operator (:=). Like all of the assignment operators, STORE assigns to the most recently declared and visible variable referenced by *<idVar>*. If, however, the variable reference is ambiguous (i.e., not declared at compile time or not explicitly qualified with an alias), it is assumed to be MEMVAR. At runtime, if no private or public variable exists with the specified name, a private variable is created.

To override a declaration, you can specify the *<idVar>* prefaced by an alias. If *<idVar>* is a field variable, use the name of the work area. For private and public variables, you can use the memory variable alias (MEMVAR->). To assign to a field variable in the currently selected work area (as opposed to a particular named work area), you can use the field alias (FIELD->).

As a matter of principle, all variables other than field variables should be declared. Preface field variables with the alias. Use of private and public variables is discouraged since they violate basic principles of modular programming and are much slower than local and static variables.

Note that the STORE command is a compatibility command and not recommended for any assignment operation. CA-Clipper provides assignment operators that supersede the STORE command, including the inline assignment operator (:=), the increment and decrement operators (++) and (--), and the compound assignment operators (+=, -=, *=, /=). Refer to the Operators and Variables sections of the "Basic Concepts" chapter in the *Programming and Utilities Guide* for more information.

Notes

- ***Assigning a value to an entire array:*** In CA-Clipper, neither the STORE command nor the assignment operators can assign a single value to an entire array. Use the AFILL() function for this purpose.
- ***Memo fields:*** Assigning a memo field to a variable assigns a character value to that variable.

Examples

- These statements create and assign values to undeclared private variables:

```
STORE "string" TO cVar1, cVar2, cVar3
cVar1:= "string2"
cVar2:= MEMVAR->cVar1
```

- These statements assign multiple variables using both STORE and the inline assignment operator (:=). The methods produce identical code:

```
STORE "value" to cVar1, cVar2, cVar3
cVar1 := cVar2 := cVar3 := "value"
```

- These statements assign values to the same field referenced explicitly with an alias. The first assignment uses the field alias (FIELD->), where the second uses the actual alias name:

```
USE Sales NEW
FIELD->CustBal = 1200.98
Sales->CustBal = 1200.98
```

Files

Library is CLIPPER.LIB.

See Also

AFILL(), LOCAL, PRIVATE, PUBLIC, RELEASE, REPLACE, RESTORE, SAVE, STATIC

STR() function

Convert a numeric expression to a character string

Syntax

```
STR(<nNumber>, [<nLength>], [<nDecimals>]) → cNumber
```

Arguments

<nNumber> is the numeric expression to be converted to a character string.

<nLength> is the length of the character string to return, including decimal digits, decimal point, and sign.

<nDecimals> is the number of decimal places to return.

Returns

STR() returns *<nNumber>* formatted as a character string. If the optional length and decimal arguments are not specified, STR() returns the character string according to the following rules:

Results of STR() with No Optional Arguments

Expression	Return Value Length
Field Variable	Field length plus decimals
Expressions/constants	Minimum of 10 digits plus decimals
VAL()	Minimum of 3 digits
MONTH()/DAY()	3 digits
YEAR()	5 digits
RECNO()	7 digits

Description

STR() is a numeric conversion function that converts numeric values to character strings. It is commonly used to concatenate numeric values to character strings. STR() has applications displaying numbers, creating codes such as part numbers from numeric values, and creating index keys that combine numeric and character data.

STR() is like TRANSFORM(), which formats numeric values as character strings using a mask instead of length and decimal specifications.

The inverse of STR() is VAL(), which converts character numbers to numerics.

Notes

- If <*nLength*> is less than the number of whole number digits in <*nNumber*>, STR() returns asterisks instead of the number.
- If <*nLength*> is less than the number of decimal digits required for the decimal portion of the returned string, CA-Clipper rounds the number to the available number of decimal places.
- If <*nLength*> is specified but <*nDecimals*> is omitted (no decimal places), the return value is rounded to an integer.

Examples

- These examples demonstrate the range of values returned by STR(), depending on the arguments specified:

```
nNumber:= 123.45
? STR(nNumber)                    // Result:  123.45
? STR(nNumber, 4)                 // Result:  123
? STR(nNumber, 2)                 // Result:  **
? STR(nNumber * 10, 7, 2)         // Result:  1234.50
? STR(nNumber * 10, 12, 4)        // Result:  1234.5000
? STR(nNumber, 10, 1)             // Result:  1234.5
```

- This example uses STR() to create an index with a compound key of order numbers and customer names:

```
USE Customer NEW
INDEX ON STR(NumOrders, 9) + CustName TO CustOrd
```

Files Library is CLIPPER.LIB.

See Also TRANSFORM(), VAL()

STRTRAN() function

Search and replace characters within a character string or memo field

Syntax

```
STRTRAN(<cString>, <cSearch>,
   [<cReplace>], [<nStart>], [<nCount>]) → cNewString
```

Arguments

<cString> is the character string or memo field to be searched.

<cSearch> is the sequence of characters to be located.

<cReplace> is the sequence of characters with which to replace <*cSearch*>. If this argument is not specified, the specified instances of the search argument are replaced with a null string ("").

<nStart> is the first occurrence that will be replaced. If this argument is omitted, the default is one. If this argument is equal to or less than zero, STRTRAN() returns an empty string.

<nCount> is the number of occurrences to be replaced. If this argument is not specified, the default is all.

Returns

STRTRAN() returns a new character string with the specified instances of <*cSearch*> replaced with <*cReplace*>.

Description

STRTRAN() is a character function that performs a standard substring search within a character string. When it finds a match, it replaces the search string with the specified replacement string. All instances of <*cSearch*> are replaced unless <*nStart*> or <*nCount*> is specified. Note that STRTRAN() replaces substrings and, therefore, does not account for whole words.

Examples

- This example uses STRTRAN() to establish a postmodern analog to a famous quotation:

```
cString:= "To compute, or not to compute?"
? STRTRAN(cString, "compute", "be")
// Result: To be, or not to be?
```

Files Library is CLIPPER.LIB.

See Also AT(), HARDCR(), MEMOTRAN(), RAT(), STUFF(), SUBSTR()

STUFF() function

Delete and insert characters in a string

Syntax

```
STUFF(<cString>, <nStart>,
   <nDelete>, <cInsert>) → cNewString
```

Arguments

<cString> is the target character string into which characters are inserted and deleted.

<nStart> is the starting position in the target string where the insertion/deletion occurs.

<nDelete> is the number of characters to be deleted.

<cInsert> is the string to be inserted.

Returns

STUFF() returns a copy of *<cString>* with the specified characters deleted and with *<cInsert>* inserted.

Description

STUFF() is a character function that deletes *<nDelete>* characters from *<cString>* beginning at the *<nStart>* position. Then, it inserts *<cInsert>* into the resulting string beginning at *<nStart>* to form the return string. With this, STUFF() can perform the following six operations:

- ***Insert:*** If *<nDelete>* is zero, no characters are removed from *<cString>*. *<cInsert>* is then inserted at *<nStart>*, and the entire string is returned. For example, STUFF("My dog has fleas.", 12, 0, "no") returns "My dog has no fleas."

- ***Replace:*** If *<cInsert>* is the same length as *<nDelete>*, *<cInsert>* replaces characters beginning at *<nStart>*. The same number of characters are deleted as are inserted, and the resulting string is the same length as the original. For example, STUFF("My dog has fleas.", 12, 5, "bones") returns "My dog has bones."

- ***Delete:*** If *<cInsert>* is a null string (""), the number of characters specified by *<nDelete>* are removed from *<cString>*, and the string is returned without any added characters. For example, STUFF("My dog has fleas.", 1, 3, "") returns "dog has fleas."

- ***Replace and insert:*** If <*cInsert*> is longer than <*nDelete*>, all characters from <*nStart*> up to <*nDelete*> are replaced and the rest of <*cInsert*> is inserted. Since more characters are inserted than are deleted, the resulting string is always longer than the original. For example, STUFF("My dog has fleas.", 8, 3, "does not have") returns "My dog does not have fleas."
- ***Replace and delete:*** If the length of <*cInsert*> is less than <*nDelete*>, more characters are deleted than inserted. The resulting string, therefore, is shorter than the original. For example, STUFF("My dog has fleas.", 8, 3, "is") returns "My dog is fleas."
- ***Replace and delete rest:*** If <*nDelete*> is greater than or equal to the number of characters remaining in <*cString*> beginning with <*nStart*>, all remaining characters are deleted before <*cInsert*> is inserted. For example, STUFF("My dog has fleas.", 8, 10, "is.") returns "My dog is."

Examples

- These examples demonstrate the six basic operations of STUFF():

```
// Insert
? STUFF("ABCDEF", 2, 0, "xyz")      // Result: AxyzBCDEF
// Replace
? STUFF("ABCDEF", 2, 3, "xyz")      // Result: AxyzEF
// Delete
? STUFF("ABCDEF", 2, 2, "")         // Result: ADEF
// Replace and insert
? STUFF("ABCDEF", 2, 1, "xyz")      // Result: AxyzCDEF
// Replace and delete
? STUFF("ABCDEF", 2, 4, "xyz")      // Result: AxyzF
// Replace and delete rest
? STUFF("ABCDEF", 2, 10, "xyz")     // Result: Axyz
```

Files Library is EXTEND.LIB, source file is SOURCE\SAMPLE\STUFF.C.

See Also AT(), LEFT(), RAT(), RIGHT(), STRTRAN(), SUBSTR()

SUBSTR() function

Extract a substring from a character string

Syntax

```
SUBSTR(<cString>, <nStart>, [<nCount>]) → cSubstring
```

Arguments

<cString> is the character string from which to extract a substring. It can be up to 65,535 (64K) bytes, the maximum character string size in CA-Clipper.

<nStart> is the starting position in *<cString>*. If *<nStart>* is positive, it is relative to the leftmost character in *<cString>*. If *<nStart>* is negative, it is relative to the rightmost character in the *<cString>*.

<nCount> is the number of characters to be extracted. If omitted, the substring begins at *<nStart>* and continues to the end of the string. If *<nCount>* is greater than the number of characters from *<nStart>* to the end of *<cString>*, the excess numbers are ignored.

Returns

SUBSTR() returns a character string.

Description

SUBSTR() is a character function that extracts a substring from another character string or memo field. SUBSTR() is related to the LEFT() and RIGHT() functions which extract substrings beginning with leftmost and rightmost characters in *<cString>*, respectively.

The SUBSTR(), RIGHT(), and LEFT() functions are often used with both the AT() and RAT() functions to locate either the first and/or the last position of a substring before extracting it. They are also used to display or print only a portion of a character string.

Examples

- These examples extract the first and last name from a variable:

```
cName:= "Biff Styvesent"
? SUBSTR(cName, 1, 4)              // Result: Biff
? SUBSTR(cName, 6)                 // Result: Styvesent
? SUBSTR(cName, LEN(cName) + 2)    // Result: null string
? SUBSTR(cName, -9)                // Result: Styvesent
? SUBSTR(cName, -9, 3)             // Result: Sty
```

- This example uses SUBSTR() with AT() and RAT() to create a user-defined function to extract a file name from a file specification:

```
? FileBase("C:\PRG\MYFILE.OBJ")     // Result: MYFILE.OBJ

FUNCTION FileBase( cFile )
   LOCAL nPos
   IF (nPos := RAT("\", cFile)) != 0
      RETURN SUBSTR(cFile, nPos + 1)
   ELSEIF (nPos := AT(":", cFile)) != 0
      RETURN SUBSTR(cFile, nPos + 1)
   ELSE
      RETURN cFile
   ENDIF
```

Files Library is CLIPPER.LIB.

See Also AT(), LEFT(), RAT(), RIGHT(), STR()

SUM command

Sum numeric expressions and assign results to variables

Syntax

```
SUM <nExp list> TO <idVar list>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

<nExp list> is the list of numeric values to sum for each record processed.

TO <idVar list> identifies the receiving variables to be assigned assign the results of the sum. Variables that either do not exist or are not visible are created as private variables. *<idVar list>* must contain the same number of elements as *<nExp list>*.

<scope> is the portion of the current database file to SUM. The default scope is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to SUM within the given scope.

Description

SUM is a database command that totals a series of numeric expressions for a range of records in the current work area and assigns the results to a series of variables. The variables specified in *<idVar list>* can be field, local, private, public, or static.

Note that the *<nExp list>* is required and not optional as it is in other dialects.

Examples

- This example illustrates the use of SUM:

```
LOCAL nTotalPrice, nTotalAmount
USE Sales NEW
SUM Price * .10, Amount TO nTotalPrice, nTotalAmount
//
? nTotalPrice              // Result: 151515.00
? nTotalAmount             // Result: 150675.00
```

Files Library is CLIPPER.LIB.

See Also AVERAGE, DBEVAL(), TOTAL

TBColumn class

Provide column objects for TBrowse objects

Class Function

`TBColumnNew(<cHeading>, <bBlock>) → oTBColumn`

Returns

Returns a new TBColumn object with the specified heading and data retrieval block. Other elements of the TBColumn object can be assigned directly using the syntax for assigning exported instance variables.

Description

A TBColumn object is a simple object containing the information needed to fully define one data column of a TBrowse object.

Exported Instance Variables

`block` (Assignable)

Contains a code block that retrieves data for the column. Any code block is valid, and no block arguments are supplied when the block is evaluated. The code block must return the appropriate data value for the column.

`cargo` (Assignable)

Contains a value of any data type provided as a user-definable slot, allowing arbitrary information to be attached to a TBColumn and retrieved later.

`colorBlock` (Assignable)

Contains an optional code block that determines the color of data items as they are displayed. If present, this block is executed each time a new value is retrieved via the TBColumn:block (the data retrieval block). The newly retrieved data value is passed as an argument to the TBColumn:colorBlock, which must return an array containing four numeric values. The values returned are used as indexes into the color table of the TBrowse object as described in the TBColumn:defColor reference below.

The TBColumn:colorBlock allows display colors for data items based on the value of the data being displayed. For example, negative numbers may be displayed in a different color than positive numbers.

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Properties Workbench window for that item.

`colSep` (Assignable)

Contains an optional character string that draws a vertical separator to the left of this column if there is another column to the left of it. If no value is supplied for TBColumn:colSep, the value contained in TBrowse:colSep is used instead.

`defColor` (Assignable)

Contains an array of four numeric values used as indexes into the color table in the TBrowse object. The first value determines the unselected color which displays data values when the browse cursor is not on the data value being displayed. The second value determines the selected color. The selected color displays the current browse cell. The third color displays the heading. The fourth color displays the footing.

The default value for TBColumn:defColor is {1, 2, 1, 1}. This causes the first two colors in the TBrowse color table to be used for unselected and selected, respectively. Note that colors set using TBColumn:colorBlock override those set by TBColumn:defColor.

`footing` (Assignable)

Contains a character value that defines the footing for this data column.

`footSep` (Assignable)

Contains a character value that draws a horizontal line between the data values and the footing. If it does not contain a character value, TBrowse:footSep is used instead.

`heading` (Assignable)

Contains a character value that defines the heading for this data column.

`headSep` (Assignable)

Contains an optional character string that draws a horizontal separator between the heading and the data values. If it does not contain a character value, the TBrowse:headSep is used instead.

`picture` (Assignable)

Contains an optional character string that controls formatting and editing of the column. See the @...GET entry in the *Reference Guide, Volume 1* for more information on PICTURE strings.

`postBlock` (Assignable)

Contains an optional code block that validates a newly entered or modified value contained within a given cell.. If present, the TBColumn:postBlock should contain an expression that evaluates to true (.T.) for a legal value and false (.F.) for an illegal value.

The TBColumn object itself ignores this variable. It is typically used by the standard READ command.

During postvalidation, the TBColumn:postBlock is passed a reference to the current Get object as an argument.

`preBlock` (Assignable)

Contains an optional code block that decides whether editing should be permitted. If present, the TBColumn:preBlock should evaluate to true (.T.) if the cursor enters the editing buffer; otherwise, it should evaluate to false (.F.).

The TBColumn object itself ignores this variable. It is typically used by the standard READ command.

During prevalidation, the TBColumn:preBlock is passed a reference to the current Get object as an argument.

`width` (Assignable)

Contains a numeric value that defines the display width for the column. If TBColumn:width is not explicitly set, the width of the column will be the greater of the length of the heading, the length of the footing, or the length of the data at the first evaluation of TBColumn:block.

If this instance variable is explicitly set, the width of the column will be TBColumn:width. Displayed headings, footings, and data will all be truncated to this width when necessary. The width of the displayed data will be the length at the first evaluation of TBColumn:block for all data types other than character. Character data will be extended to TBColumn:width for display purposes.

Exported Methods

`setstyle( <nStyle>, [<lSetting>] ) → self`

TBColumn:setStyle() maintains a dictionary within a TBColumn object. This dictionary, which is simply an array, contains a set of logical values that determine behaviors associated with a TBrowse column. *<nStyle>* refers to the element in the dictionary that contains the style. *<lSetting>* indicates whether the style should be permitted or denied. Set to true (.T.) to allow the behavior to occur; otherwise, set to false (.F.) to prohibit it. CA-Clipper reserves the first three elements of the dictionary for predefined styles.

You may add custom styles to a TBColumn object by specifying any unused element of the dictionary. A maximum of 4096 definitions is available. When adding new styles to the dictionary, use the TBC_CUSTOM constant to ensure that the new styles will not interfere with the predefined ones. This guarantees that if more predefined styles are added in future releases of CA-Clipper, the positions of your styles in the dictionary will be adjusted automatically.

Styles are used by neither the TBColumn object nor the TBrowse object. The style dictionary is merely a convenient method of associating behaviors with a TBColumn object. The functions that query and implement these behaviors are external to the object. An example of this can be found in BrowSys.prg in the CLIP53\SAMPLES subdirectory.

TBColumn Styles

Number	TBrowse.ch	Meaning
1	TBC_READWRITE	Can the user modify the data in the column's cells?
2	TBC_MOVE	Can the user move the column to another position in the browse?
3	TBC_SIZE	Can the user modify the width of the column?
4	TBC_CUSTOM	First available element for custom styles.

Tbrowse.ch contains manifest constants for TBColumn:SetStyle().

Note: TBC_MOVE and TBC_SIZE are not implemented in CA-Clipper 5.3. They are reserved for future usage.

Examples

- This example is a code fragment that creates a TBrowse object and adds some TBColumn objects to it:

```
USE Customer NEW
//

// Create a new TBrowse object
objBrowse := TBrowseDB(1, 1, 23, 79)
//

// Create some new TBColumn objects and
// add them to the TBrowse object
objBrowse:addColumn(TBColumnNew( "Customer", ;
      {|| Customer->Name} ))
objBrowse:addColumn(TBColumnNew( "Address", ;
      {|| Customer->Address} ))
objBrowse:addColumn(TBColumnNew( "City", ;
      {|| Customer->City} ))
.
. <statements to actually browse the data>
.
CLOSE Customer
```

  For more information on TBrowse, refer to the "Introduction to TBrowse" chapter in the *Programming and Utilities Guide*. For a fully operational example of a TBrowse object, see TbDemo.prg located in \CLIP53\SOURCE\SAMPLE.

See Also BROWSE(), DBEDIT()*, TBrowse class

TBrowse class

Provide objects for browsing table-oriented data

Class Functions

```
TBrowseNew(<nTop>, <nLeft>, <nBottom>, <nRight>)
   → oTBrowse

TBrowseDB(<nTop>, <nLeft>, <nBottom>, <nRight>)
   → oTBrowse
```

Returns

TBrowseNew() returns a new TBrowse object with the specified coordinate settings defined by *<nTop>*, *<nLeft>*, *<nBottom>*, and *<nRight>*. The TBrowse object is created with no columns and no code blocks for data positioning. These must be provided before the TBrowse object can be used.

TBrowseDB() returns a new TBrowse object with the specified coordinate settings and default code blocks for data source positioning within database files. The coordinate settings are defined by *<nTop>*, *<nLeft>*, *<nBottom>*, and *<nRight>*. The default code blocks execute the GO TOP, GO BOTTOM, and SKIP operations. Note that TBrowseDB() creates an object with no column objects. To make the TBrowse object usable, you must add a column for each field to be displayed.

Description

A TBrowse object is a general purpose browsing mechanism for table-oriented data. TBrowse objects provide a sophisticated architecture for acquiring, formatting, and displaying data. Data retrieval and file positioning are performed via user-supplied code blocks, allowing a high degree of flexibility and interaction between the browsing mechanism and the underlying data source. The format of individual data items can be precisely controlled via the TBColumn data retrieval code blocks; overall display formatting and attributes can be controlled by sending appropriate messages to the TBrowse object.

Note: TBrowse has a memory limit of 200 fields.

A TBrowse object relies on one or more TBColumn objects. A TBColumn object contains the information necessary to define a single column of the browse table (see TBColumn class in this chapter).

During operation, a TBrowse object retrieves data by evaluating code blocks. The data is organized into rows and columns and displayed within a specified rectangular region of the screen. The TBrowse object maintains an internal browse cursor. The data item on which the browse cursor rests is displayed in a highlighted color. (The actual screen cursor is also positioned to the first character of this data item.)

Initially, the browse cursor is placed on the data item at the top left of the browse display. Messages can then be sent to the TBrowse object to navigate the displayed data, causing the browse cursor to move. These messages are normally sent in response to user keystrokes.

New data is automatically retrieved as required by navigation requests. When navigation proceeds past the edge of the visible rectangle, rows or columns beyond that edge are automatically brought into view. When new rows are brought into view, the underlying data source is repositioned by evaluating a code block.

Note: TBrowse objects do not clear the entire window before output during redisplay operations. Part of the window may still be cleared when data from the existing display is scrolled.

Exported Instance Variables

`autoLite` (Assignable)

Contains a logical value. When autoLite is set to true (.T.), the stabilize method automatically highlights the current cell as part of stabilization. The default for autoLite is true (.T.).

`border` (Assignable)

Contains a character value that defines the characters that comprise the box that is drawn around the TBrowse object on the screen. Its length must be either zero or eight characters. If not specified, the browse appears without a border. When present, the browse occupies the region of the screen specified by TBrowse:nTop + 1, TBrowse:nLeft + 1, TBrowse:nBottom - 1, TBrowse:nRight - 1. This effectively decreases the number of screen rows and columns that the browse occupies by two.

`cargo` (Assignable)

Contains a value of any data type provided as a user-definable slot. TBrowse:cargo allows arbitrary information to be attached to a TBrowse object and retrieved later.

`colCount`

Contains a numeric value indicating the total number of data columns in the browse. For each column, there is an associated TBColumn object.

`colorSpec` `(Assignable)`

Contains a character string defining a color table for the TBrowse display. As a default, the current SETCOLOR() value is copied into this variable when the TBrowse object is created.

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Properties Workbench window for that item.

`colPos` `(Assignable)`

Contains a numeric value indicating the data column where the browse cursor is currently located. Columns are numbered from 1, starting with the leftmost column.

`colSep` `(Assignable)`

Contains a character value that defines a column separator for TBColumns that do not contain a column separator of their own (see TBColumn class in this chapter for more information).

`footSep` `(Assignable)`

Contains a character value that defines a footing separator for TBColumn objects not containing a footing separator of their own (see TBColumn class in this chapter).

`freeze` `(Assignable)`

Contains a numeric value that defines the number of data columns frozen on the left side of the display. Frozen columns are always visible, even when other columns are panned off the display.

`goBottomBlock` `(Assignable)`

Contains a code block executed in response to the TBrowse:goBottom() message. This block is responsible for repositioning the data source to the last record displayable in the browse. Typically the data source is a database file, and this block contains a call to a user-defined function that executes a GO BOTTOM command.

`goTopBlock` (Assignable)

Contains a code block that is executed in response to the TBrowse:goTop() message. This block is responsible for repositioning the data source to the first record displayable in the browse. Typically the data source is a database file, and this block contains a call to a user-defined function that executes a GO TOP command.

`headSep` (Assignable)

Contains a character value that defines a heading separator for TBColumn objects not containing a heading separator of their own (see TBColumn class in this chapter).

`hitBottom` (Assignable)

Contains a logical value indicating whether an attempt was made to navigate beyond the end of the available data. TBrowse:hitBottom contains true (.T.) if an attempt was made; otherwise it contains false (.F.). During stabilization, the TBrowse object sets this variable if TBrowse:skipBlock indicates it was unable to skip forward as many records as requested.

`hitTop` (Assignable)

Contains a logical value indicating whether an attempt was made to navigate past the beginning of the available data. TBrowse:hitTop contains true (.T.) if an attempt was made; otherwise it contains false (.F.). During stabilization, the TBrowse object sets this variable if TBrowse:skipBlock indicates that it was unable to skip backward as many records as requested.

`leftVisible`

Contains a numeric value indicating the position of the leftmost unfrozen column visible in the browse display. If every column is frozen in the display, TBrowse:leftVisible contains zero.

`mColPos` (Assignable)

Contains a numeric value indicating the data column where the mouse cursor is currently located. Columns are numbered from 1, starting with the leftmost column.

`message` (Assignable)

Contains a character string that is displayed on the GET system's status bar line when the TBrowse is utilized within the GET system. Typically, it describes the anticipated contents of, or user response to, the browse. Refer to @ ...GET TBROWSE and the READ command for details pertaining to the GET system's status bar.

`mRowPos` (Assignable)

Contains a numeric value indicating the data row where the mouse cursor is currently located. Data rows are numbered from 1, starting with the topmost data row. Screen rows containing headings, footings, or separators are not considered data rows.

`nBottom` (Assignable)

Contains a numeric value defining the bottom screen row used for the TBrowse display.

`nLeft` (Assignable)

Contains a numeric value defining the leftmost screen column used for the TBrowse display.

`nRight` (Assignable)

Contains a numeric value defining the rightmost screen column used for the TBrowse display.

`nTop` (Assignable)

Contains a numeric value defining the top screen row used for the TBrowse display.

`rightVisible`

Contains a numeric value indicating the position of the rightmost unfrozen column visible in the browse display. If all columns visible in the display are frozen, TBrowse:rightVisible contains zero.

`rowCount`

Contains a numeric value indicating the number of data rows visible in the browse display. Only data rows are included in the count. Rows occupied by headings, footings, or separators are not included.

`rowPos` (Assignable)

Contains a numeric value indicating the data row where the browse cursor is currently located. Data rows are numbered from 1, starting with the topmost data row. Screen rows containing headings, footings, or separators are not considered data rows.

`skipBlock` (Assignable)

Contains a code block that repositions the data source. During stabilization, this code block is executed with a numeric argument when the TBrowse object needs to reposition the data source. The numeric argument passed to the block represents the number of records to be skipped. A positive value means skip forward, and a negative value means skip backward. A value of zero indicates that the data source does not need to be repositioned, but the current record may need to be refreshed. Typically, the data source is a database file, and this block calls a user-defined function that executes a SKIP command to reposition the record pointer.

The block must return the number of rows (positive, negative, or zero) actually skipped. If the value returned is not the same as the value requested, the TBrowse object assumes that the skip operation encountered the beginning or end of file.

`stable` (Assignable)

Contains a logical value indicating whether the TBrowse object is stable. It contains true (.T.) if the TBrowse object is stable; otherwise, it contains false (.F.). The browse is considered stable when all data has been retrieved and displayed, the data source has been repositioned to the record corresponding to the browse cursor, and the current cell has been highlighted. When navigation messages are sent to the TBrowse object, TBrowse:stable is set to false (.F.). After stabilization is performed using the TBrowse:stabilize() message, TBrowse:stable is set to true (.T.).

Exported Methods

Cursor Movement Methods

`down() → self`

Moves the browse cursor down one row. If the cursor is already on tl bottom row, the display is scrolled up and a new row is brought into If the data source is already at the logical end of file and the browse c is already on the bottom row, TBrowse:hitBottom is set true (.T.).

`end() → self`

Moves the browse cursor to the rightmost data column currently visible.

`goBottom() → self`

Repositions the data source to logical bottom of file (by evaluating the TBrowse:goBottomBlock), refills the display with the bottommost available data, and moves the browse cursor to the lowermost data row for which data is available. The pan position of the window is not changed.

`goTop() → self`

Repositions the data source to the logical beginning of file (by evaluating the TBrowse:goTopBlock), refills the display with the topmost available data, and moves the browse cursor to the uppermost data row for which data is available. The pan position of the window is not changed.

`home() → self`

Moves the browse cursor to the leftmost unfrozen column on the display.

`left() → self`

Moves the browse cursor left one data column. If the cursor is already on the leftmost displayed column, the display is panned and the previous data column (if there is one) is brought into view.

`pageDown() → self`

Repositions the data source downward and refills the display. If the data source is already at the logical end of file (i.e., the bottommost available record is already shown), the browse cursor is simply moved to the lowermost row containing data. If the data source is already at the logical end of file and the browse cursor is already on the bottom row, TBrowse:hitBottom is set true (.T.).

`pageUp() → self`

Repositions the data source upward and refills the display. If the data source is already at logical beginning of file (i.e., the topmost available record is already shown), the browse cursor is simply moved to the top data row. If the data source is already at logical beginning of file and the browse cursor is already on the first data row, TBrowse:hitTop is set true (.T.).

`panEnd() → self`

Moves the browse cursor to the rightmost data column, causing the display to be panned completely to the right.

`panHome() → self`

Moves the browse cursor to the leftmost data column, causing the display to be panned all the way to the left.

`panLeft() → self`

Pans the display without changing the browse cursor, if possible. When the screen is panned with TBrowse:panLeft(), at least one data column out of view to the left is brought into view, while one or more columns are panned off screen to the right.

`panRight() → self`

Pans the display without changing the browse cursor, if possible. When the screen is panned with TBrowse:panRight(), at least one data column out of view to the right is brought into view, while one or more columns are panned off the screen to the left.

`right() → self`

Moves the browse cursor right one data column. If the cursor is already at the right edge, the display is panned and the next data column (if there is one) is brought into view.

`up() → self`

Moves the browse cursor up one row. If the cursor is already on the top data row, the display is scrolled down and a new row is brought into view. If the data source is already at the logical beginning of file and the browse cursor is already on the top data row, TBrowse:hitTop is set true (.T.).

Miscellaneous Methods

`addColumn(<oColumn>) → self`

Adds a new TBColumn object to the TBrowse object and TBrowse:colCount is increased by one.

`applyKey(<nKey> ) → nResult`

Evaluates the code block associated with *<nKey>* that is contained within the TBrowse:setKey() dictionary. *<nResult>*, which is the code block's return value, specifies the manner in which the key was processed.

TBrowse:ApplyKey() Return Values

Constant	Value	Meaning
TBR_EXIT	-1	User request for the browse to lose input focus
TBR_CONTINUE	0	Code block associated with *<nKey>* was evaluated
TBR_EXCEPTION	1	Unable to locate *<nKey>* in the dictionary, key was not processed

Tbrowse.ch contains manifest constants for the TBrowse:applyKey() return values.

Previously, before TBrowse:applyKey() was available, a TBrowse was typically maintained by placing an INKEY() and a CASE structure within a WHILE loop. For example:

```
WHILE ( .T. )
   oTB:ForceStable()
   nKey := Inkey( 0 )
   DO CASE
   CASE ( nKey == K_UP )
      oTB:Up()
   CASE ( nKey == K_DOWN )
      oTB:Down()
   .
   .
   .
   CASE ( nKey := K_ESC )
      EXIT
   ENDCASE
ENDDO
```

Because this code was external to the browse, it did not encapsulate the relationship between a key press and its effect on the browse. TBrowse:applyKey() resolves this by placing the key and its associated code block in a dictionary within the TBrowse object. This effectively makes creating and managing a browse simple, yet flexible. For example:

```
WHILE ( .T. )
    oTB:ForceStable()
    IF ( oTB:ApplyKey( Inkey( 0 ) ) == TBR_EXIT )
       EXIT
    ENDIF
ENDDO
```

`colorRect(<aRect>, <aColors>)` → *self*

Directly alters the color of a rectangular group of cells. *<aRect>* is an array of four numbers (top, left, bottom, and right). The numbers refer to cells within the data area of the browse display, not to screen coordinates. *<aColors>* is an array of two numbers. The numbers are used as indexes into the color table for the browse. These colors will become the normal and highlighted colors for the cells within the specified rectangle.

Cells that are colored using colorRect retain their color until they are scrolled off the screen up or down. Horizontal panning has no affect on these colors and, in fact, cells that are currently off screen left or right can be colored even if they are not visible.

This example colors the entire virtual window (on and off screen):

```
aRect := {1, 1, browse:rowCount, browse:colCount}
browse:colorRect( aRect, {2, 1} )
```

`colWidth(<nColumn>)` → *nWidth*

Returns the display width of column number *<nColumn>* as known to the browse. If *<nColumn>* is out of bounds, not supplied, or not a number, the method returns zero.

`configure() → self`

Causes the TBrowse object to re-examine all instance variables and TBColumn objects, reconfiguring its internal settings as required. This message can force reconfiguration when a TBColumn object is modified directly.

`deHilite() → self`

Causes the current cell (the cell to which the browse cursor is positioned) to be "dehighlighted." This method is designed for use when TBrowse:autoLite is set to false (.F.).

`delColumn(<nPos>) → oColumn`

This new method allows a column to be deleted from a browse. The return value is a reference to the column object being deleted, so that the column object may be preserved.

`forceStable()`

Performs a full stabilization of the TBrowse. It is analogous to the following code, only slightly faster:

```
DO WHILE .NOT. oBrowse:stabilize()
ENDDO
```

`getColumn(<nColumn>) → oColumn`

Returns the TBColumn object specified by *<nColumn>*.

`hilite() → self`

Causes the current cell (the cell to which the browse cursor is positioned) to be highlighted. This method is designed for use when TBrowse:autoLite is set to false (.F.).

`hittest(<nRow>, <nColumn>)→ <nHitTest>`

Determines if the screen position specified by *<nRow>* and *<nColumn>* is on the TBrowse object.

Applicable Hit Test Return Values

Value	Constant	Description
0	HTNOWHERE	The screen position is not within the region of the screen that the TBrowse occupies
-1	HTTOPLEFT	The screen position is on the TBrowse border's top-left corner
-2	HTTOP	The screen position is on the TBrowse's top border
-3	HTTOPRIGHT	The screen position is on the TBrowse border's top-right corner
-4	HTRIGHT	The screen position is on the TBrowse's right border
-5	HTBOTTOMRIGHT	The screen position is on the TBrowse border's bottom-right corner
-6	HTBOTTOM	The screen position is on the TBrowse's bottom border
-7	HTBOTTOMLEFT	The screen position is on the TBrowse border's bottom-left corner
-8	HTLEFT	The screen position is on the TBrowse's left border
-5121	HTCELL	The screen position is on a cell
-5122	HTHEADING	The screen position is on a heading
-5123	HTFOOTING	The screen position is on a footing
-5124	HTHEADSEP	The screen position is on the heading separator line
-5125	HTFOOTSEP	The screen position is on the footing separator line
-5126	HTCOLSEP	The screen position is on a column separator line

Button.ch contains manifest constants for the TBrowse:hittest() return values.

`insColumn(<nPos>, <oColumn>)` → *oColumn*

This method allows a column object to be inserted into the middle of a browse. The return value is a reference to the column object being inserted.

`invalidate() → self`

TBrowse:invalidate() causes the next stabilization of the TBrowse object to redraw the entire TBrowse display, including headings, footings, and all data rows. Note that sending this message has no effect on the values in the data rows: it simply forces the display to be updated during the next stabilization. To force the data to be refreshed from the underlying data source, send the TBrowse:refreshAll() message.

`refreshAll() → self`

Internally marks all data rows as invalid, causing them to be refilled and redisplayed during the next stabilization loop.

`refreshCurrent() → self`

Internally marks the current data row as invalid, causing it to be refilled and redisplayed during the next stabilization loop.

`setColumn(<nColumn>, <oColumnNew>) → oColumnCurrent`

Replaces the TBColumn object indicated by *<nColumn>* with the TBColumn object specified by *<oColumnNew>*. The value returned is the current TBColumn object.

```
setKey( <nKey> [, <bBlock>]) → bPrevious
```

Gets and optionally sets the code block indicated by *<bBlock>* that is associated with the Inkey value specified by *<nKey>*. When replacing an existing keypress/code block definition, it returns the previous code block; otherwise, it returns the current one.

TBrowse:SetKey() Default Keypress/Code Block Definitions

Inkey Value	Associated Code Block
K_DOWN	{ | oTB, nKey | oTB:Down(), TBR_CONTINUE }
K_END	{ | oTB, nKey | oTB:End(), TBR_CONTINUE }
K_CTRL_PGDN	{ | oTB, nKey | oTB:GoBottom(), TBR_CONTINUE }
K_CTRL_PGUP	{ | oTB, nKey | oTB:GoTop(), TBR_CONTINUE }
K_HOME	{ | oTB, nKey | oTB:Home(), TBR_CONTINUE }
K_LEFT	{ | oTB, nKey | oTB:Left(), TBR_CONTINUE }
K_PGDN	{ | oTB, nKey | oTB:PageDown(), TBR_CONTINUE }
K_PGUP	{ | oTB, nKey | oTB:PageUp(), TBR_CONTINUE }
K_CTRL_END	{ | oTB, nKey | oTB:PanEnd(), TBR_CONTINUE }
K_CTRL_HOME	{ | oTB, nKey | oTB:PanHome(), TBR_CONTINUE }
K_CTRL_LEFT	{ | oTB, nKey | oTB:PanLeft(), TBR_CONTINUE }
K_CTRL_RIGHT	{ | oTB, nKey | oTB:PanRight(), TBR_CONTINUE }
K_RIGHT	{ | oTB, nKey | oTB:Right(), TBR_CONTINUE }
K_UP	{ | oTB, nKey | oTB:Up(), TBR_CONTINUE }
K_ESC	{ | oTB, nKey | TBR_EXIT}
K_LBUTTONDOWN	{ | oTB, nKey | tbMouse(oTB, MROW(), MCOL()) }

Key handlers may be queried, added, replaced, and removed from the dictionary. For example:

```
oTB:SetKey( K_ESC, { | oTB, nKey | TBR_EXIT } )
```

A default key handler may be declared by specifying a value of 0 for *<nKey>*. Its associated code block will be evaluated each time TBrowse:ApplyKey() is called with a key value that is not contained in the dictionary. For example:

```
oTB:SetKey( 0, { | oTB, nKey | DefTBProc( oTB, nKey ) }
```

The example above calls a function named DefTBProc() when nKey is not contained in the dictionary.

To remove a keypress/code block definition, specify NIL for *<bBlock>*. For example:

```
   oTB:SetKey( 0, NIL }
```

`setstyle( <nStyle>, [<lSetting>] ) → self`

TBrowse:setStyle() maintains a dictionary within a TBrowse object. This dictionary, which is simply an array, contains a set of logical values that determine behaviors associated with a TBrowse object. *<nStyle>* refers to the element in the dictionary that contains the style. *<lSetting>* indicates whether the style should be permitted or denied. Set to true (.T.) to allow the behavior to occur; otherwise, set to false (.F.) to prohibit it. CA-Clipper reserves the first four elements of the dictionary for predefined styles.

You may add custom styles to a TBrowse object by specifying any unused element of the dictionary. A maximum of 4096 definitions is available. When adding new styles to the dictionary, use the TBR_CUSTOM constant to ensure that the new styles will not interfere with the predefined ones. This guarantees that if more predefined styles are added in future releases of CA-Clipper, the positions of your styles in the dictionary will be adjusted automatically.

Styles are not utilized by the TBrowse object. The style dictionary is merely a convenient method of associating behaviors with a browse. The functions that query and implement these behaviors are external to the object. An example of this can be found in BrowSys.prg in the CLIP53\SAMPLES subdirectory.

TBrowse Styles

Number	TBrowse.ch	Meaning
1	TBR_APPEND	Can the user add new information?
2	TBR_MODIFY	Can the user modify the data in the browse's cells?
3	TBR_MOVE	Can the user move the column to another position in the browse?
4	TBR_SIZE	Can the user modify the width of the column?
5	TBR_CUSTOM	First available element for custom styles.

Tbrowse.ch contains manifest constants for TBrowse:setStyle().

Note: TBR_MOVE and TBR_SIZE are not implemented in CA-Clipper 5.3. They are reserved for future usage.

`stabilize() → lStable`

Performs incremental stabilization. Each time this message is sent, some part of the stabilization process is performed. Stabilization is performed in increments so that it can be interrupted by a keystroke or other asynchronous event.

If the TBrowse object is already stable, a value of true (.T.) is returned. Otherwise, a value of false (.F.) is returned indicating that further stabilization messages should be sent. The browse is considered stable when all data has been retrieved and displayed, the data source has been repositioned to the record corresponding to the browse cursor, and the current cell has been highlighted.

Examples

For fully operational examples of a TBrowse object, refer to "Introduction to TBrowse" in the *Programming and Utilities Guide* and to TbDemo.prg located in \CLIP53\SOURCE\SAMPLE.

See Also

BROWSE(), DBEDIT()*, SETCOLOR(), TBColumn class

TEXT* command

Display a literal block of text

Syntax

```
TEXT [TO PRINTER] [TO FILE <xcFile>]
   <text>...
ENDTEXT
```

Arguments

<text> is the block of literal characters to be displayed to the screen. Text is displayed exactly as formatted.

TO PRINTER echoes the display to the printer.

TO FILE <xcFile> echoes the display to the specified file. *<xcFile>* may be specified as a literal file name or as a character expression enclosed in parentheses. If no extension is specified, .txt is assumed.

Description

TEXT...ENDTEXT is a console command construct that displays a block of text to the screen, optionally echoing output to the printer and/or a text file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the TEXT command line.

Text within the TEXT construct displays exactly as formatted, including any indentation. Hard carriage returns are output as new lines, soft carriage returns as the character CHR(141). Macro variables found within TEXT...ENDTEXT are expanded. However, macro expressions are not.

TEXT...ENDTEXT is a compatibility command and not recommended. CA-Clipper has other facilities for text processing and output. For example, MEMOLINE() in combination with MLCOUNT() can word wrap long strings according to a specified line length. ? or @...SAY can display formatted text extracted from a long string with MEMOLINE().

Examples

- This example demonstrates how to use TEXT...ENDTEXT to print a form letter:

```
USE Sales NEW
DO WHILE !EOF()
    FormLetter()
    SKIP
ENDDO
RETURN

FUNCTION FormLetter
    LOCAL dDate := DTOC(DATE()), cSalesman := ;
              RTRIM(Salesman)
    TEXT TO PRINTER
    &dDate.
    Dear &cSalesman.,
    How are you!
    ENDTEXT
    EJECT
    RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also ? | ??, @...SAY, MEMOLINE(), MLCOUNT(), SET CONSOLE

TIME() function

Return the system time

Syntax

`TIME() → cTimeString`

Returns

TIME() returns the system time as a character string in the form *hh:mm:ss*. *hh* is hours in 24-hour format, *mm* is minutes, and *ss* is seconds.

Description

TIME() is a time function that displays the system time on the screen or prints it on a report. TIME() is related to SECONDS() which returns the integer value representing the number of seconds since midnight. SECONDS() is generally used in place of TIME() for time calculations.

Examples

- These examples show the results of TIME() used with SUBSTR() to extract the hour, minutes, and seconds digits:

```
? TIME()                        // Result: 10:37:17
? SUBSTR(TIME(), 1, 2)          // Result: 10
? SUBSTR(TIME(), 4, 2)          // Result: 37
? SUBSTR(TIME(), 7, 2)          // Result: 17
```

Files

Library is CLIPPER.LIB.

See Also

DATE(), SECONDS(), SUBSTR()

TONE() function

Sound a speaker tone for a specified frequency and duration

Syntax

```
TONE(<nFrequency>, <nDuration>) → NIL
```

Arguments

<nFrequency> is a positive numeric value indicating the frequency of the tone to be sounded.

<nDuration> is a positive numeric value indicating the duration of the tone measured in increments of 1/18 of a second. For example, an *<nDuration>* value of 18 represents one second.

For both arguments, non-integer values are truncated—not rounded—to their integer portion.

Returns

TONE() always returns NIL.

Description

TONE() is a sound function that indicates various program states to the user. These can be error states, boundary conditions, or the end of a time-consuming process. For example, an error state would sound an error tone before alerting the user with a message or interactive dialog box. A boundary condition might indicate that the user is attempting to move the cursor past the top or bottom of a column in a TBrowse object. A batch process also might indicate its completion with a sound to alert the user, in case the user has turned away from the screen.

TONE() works by sounding the speaker at the specified frequency for the specified duration. The duration is measured in increments of 1/18 of a second. The frequency is measured in hertz (cycles per second). Frequencies of less than 20 are inaudible. The table below shows the frequencies of standard musical notes.

Note: TONE() works only on IBM PC and 100 percent compatible computers.

Table of Musical Notes

Pitch	Frequency	Pitch	Frequency
C	130.80	mid C	261.70
C#	138.60	C#	277.20
D	146.80	D	293.70
D#	155.60	D#	311.10
E	164.80	E	329.60
F	174.60	F	349.20
F#	185.00	F#	370.00
G	196.00	G	392.00
G#	207.70	G#	415.30
A	220.00	A	440.00
A#	233.10	A#	466.20
B	246.90	B	493.90
		C	523.30

Examples

- This example is a beep function that indicates that a batch operation has completed:

```
FUNCTION DoneBeep
    TONE(300, 1)
    TONE(100, 1)
    TONE(300, 1)
    TONE(100, 1)
    RETURN NIL
```

- This example is a tone sequence that indicates invalid keystrokes or boundary conditions:

```
FUNCTION ErrorBeep
    TONE(100, 3)
    RETURN NIL
```

Files

Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also

CHR(), SET BELL

TopBarMenu class

Create a top bar menu

Class Function

`TopBar(<nRow>, <nLeft>, <nRight>) → oTopBar`

Arguments

<nRow> is a numeric value that indicates the screen row of the top bar menu. This value is assigned to the TopBarMenu:row instance variable.

<nLeft> is a numeric value that indicates the left screen column of the top bar menu. This value is assigned to the TopBarMenu:left instance variable.

<nRight> is a numeric value that indicates the right screen column of the top bar menu. This value is assigned to the TopBarMenu:right instance variable.

Returns

Returns a TopBarMenu object when all of the required arguments are present; otherwise, TopBar() returns NIL.

Description

The top bar menu is used as a main menu in which pop-up menus reside.

Exported Instance Variables

`cargo` (Assignable)

Contains a value of any type that is ignored by the TopBarMenu object. TopBarMenu:cargo is provided as a user-definable slot allowing arbitrary information to be attached to a TopBarMenu object and retrieved later.

`colorSpec` (Assignable)

Contains a character string that indicates the color attributes that are used by the top bar menu's display() method. The string must contain six color specifiers.

TopBarMenu Color Attributes

Position in colorSpec	Applies To	Default Value from System Color Setting
1	The top bar menu items that are not selected	Unselected
2	The selected top bar menu item	Enhanced
3	The accelerator key for unselected top bar menu items	Background
4	The accelerator key for the selected top bar menu item	Enhanced
5	Disabled top bar menu items	Standard
6	The top bar menu's border	Border

Note: The colors available to a DOS application are more limited than those for a Windows application. The only colors available to you here are listed in the drop-down list box of the Properties Workbench window for that item.

`current` `(Assignable)`

Contains a numeric value that indicates which item is selected.

`itemCount`

Contains a numeric value that indicates the total number of items in the TopBarMenu object.

`left` `(Assignable)`

Contains a numeric value that indicates the top bar menu's leftmost column

`right` `(Assignable)`

Contains a numeric value that indicates the top bar menu's rightmost column

`row` `(Assignable)`

Contains a numeric value that indicates the row that the top bar menu appears on.

Exported Methods

`<oTopBar>:addItem(<oMenuItem>) → self`

<oMenuItem> is the MenuItem object to be added.

addItem() is a method of the TopBarMenu class that is used for appending a new item to a top bar menu.

`<oTopBar>:delItem(<nPosition>) → self`

<nPosition> is a numeric value that indicates the position in the top bar menu of the item to be deleted.

delItem() is a method of the TopBarMenu class that is used for removing an item from a top bar menu. When an item is deleted, the items which follow it move up a line.

`<oTopBar>:display() → self`

display() is a method of the TopBarMenu class that is used for showing a top bar menu and its items on the screen. It also shows the status bar description for menu items that contain one. display() uses the values of the following instance variables to correctly show the list in its current context, in addition to providing maximum flexibility in the manner a top bar menu appears on the screen: colorSpec, current, itemCount, left, right, and row.

Note: See the MENUMODAL() function in this guide for more information about displaying and activating a top menu bar.

`<oTopBar>:getFirst() → nPosition`

Returns a numeric value that indicates the position within the top bar menu of the first selectable item. getFirst() returns 0 in the event that the top bar menu does not contain a selectable item.

getFirst() is a method of the TopBarMenu class that is used for determining the position of the first selectable item in a top bar menu. The term selectable is defined as a menu item that is enabled and whose caption is not a menu separator.

Note: getFirst() does not change the currently selected menu item. In order to change the currently selected top bar menu item, you must call the TopBarMenu:select() method.

`<oTopBar>:getItem(<nPosition>) → self`

<nPosition> is a numeric value that indicates the position in the top bar menu of the item that is being retrieved.

getItem() is a method of the TopBarMenu class that is used for accessing a MenuItem object after it has been added to a top bar menu.

`<oTopBar>:getLast() → nPosition`

Returns a numeric value that indicates the position within the top bar menu of the last selectable item. getLast() returns 0 in the event that the top bar menu does not contain a selectable item.

getLast() is a method of the TopBarMenu class that is used for determining the position of the last selectable item in a top bar menu. The term selectable is defined as a menu item that is enabled and whose caption, is not a menu separator.

Note: getLast() does not change the currently selected menu item. In order to change the currently selected top bar menu item, you must call the TopBarMenu:select() method.

`<oTopBar>:getNext() → nPosition`

Returns a numeric value that indicates the position within the top bar menu of the next selectable item. getNext() returns 0 in the event that the current item is the last selectable item or the top bar menu does not contain a selectable item.

getNext() is a method of the TopBarMenu class that is used for determining the position of the next selectable item in a top bar menu. getNext() searches for the next selectable item starting at the item immediately after the current item. The term selectable is defined as a menu item that is enabled and whose caption is not a menu separator.

Note: getNext() does not change the currently selected menu item. In order to change the currently selected top bar menu item, you must call the TopBarMenu:select() method.

`<oTopBar>:getPrev() → nPosition`

Returns a numeric value that indicates the position within the top bar menu of the previous selectable item. getPrev() returns 0 in the event that the current item is the first selectable item or the top bar menu does not contain a selectable item.

getPrev() is a method of the TopBarMenu class that is used for determining the position of the previous selectable item in a top bar menu. getPrev() searches for the previous selectable item starting at the item immediately before the current item. The term selectable is defined as a menu item that is enabled and whose caption is not a menu separator.

Note: getPrev() does not change the currently selected menu item. In order to change the currently selected top bar menu item, you must call the TopBarMenu:select() method.

`<oTopBar>:getAccel(<nInkeyValue>) → nPosition`

<nInkeyValue> is a numeric value that indicates the inkey value to be checked.

Returns a numeric value that indicates the position in the top bar menu of the first item whose accelerator key matches that which is specified by *<nInkeyValue>*. The accelerator key is defined using the & character in MenuItem:caption.

getAccel() is a method of the TopBarMenu class that is used for determining whether a key press should be interpreted as a user request to evoke the data variable of a particular top bar menu item.

`<oTopBar>:hitTest(<nMouseRow>, <nMouseCol>)`
`→ nHitStatus`

<nMouseRow> is a numeric value that indicates the current screen row position of the mouse cursor.

<nMouseCol> is a numeric value that indicates the current screen column position of the mouse cursor.

Returns a numeric value that indicates the relationship of the mouse cursor with the top bar menu.

Hit Test Return Values

Value	Constant	Description
> 0	Not applicable	The position in the top bar menu of the item whose region the mouse is within
0	HTNOWHERE	The mouse cursor is not within the region of the screen that the top bar menu occupies
-1	HTTOPLEFT	The mouse cursor is on the top-left corner of the top bar menu's border
-2	HTTOP	The mouse cursor is on the top bar menu's top border
-3	HTTOPRIGHT	The mouse cursor is on the top-right corner of the top bar menu's border
-4	HTRIGHT	The mouse cursor is on the top bar menu's right border
-5	HTBOTTOMRIGHT	The mouse cursor is on the bottom-right corner of the top bar menu's border
-6	HTBOTTOM	The mouse cursor is on the top bar menu's bottom border
-7	HTBOTTOMLEFT	The mouse cursor is on the bottom-left corner of the top bar menu's border
-8	HTLEFT	The mouse cursor is on the top bar menu's left border

Button.ch contains manifest constants for the TopBarMenu:hitTest() return value.

hitTest() is a method of the TopBarMenu class that is used for determining if the mouse cursor is within the region of the screen that the Top bar menu occupies.

```
<oTopBar>:insItem(<nPosition>, <oMenuItem>) → self
```

<nPosition> is a numeric value that indicates the position at which the new menu item is inserted.

<oMenuItem> is the MenuItem object to be inserted.

insItem() is a method of the TopBarMenu class that is used for inserting a new item in a top bar menu.

`<oTopBar>:select(<nPosition>)` → *self*

<nPosition> indicates the position in the top bar menu of the item to be selected.

select() is a method of the TopBarMenu class that is used for changing the selected item. Its state is typically changed when one of the arrow keys is pressed or the mouse's left button is pressed when its cursor is within the top bar menu's screen region.

`<oTopBar>:setItem(<nPosition>, <oMenuItem>)` → *self*

<nPosition> is a numeric value that indicates the position in the top bar menu of the item that is being retrieved.

<oMenuItem> is the MenuItem object that replaces the one in the top bar menu specified by *<nPosition>*.

setItem() is a method of the TopBarMenu class that is used for replacing a MenuItem object after it has been added to a top bar menu. After the setItem() method is called, the display() method needs to be called in order to refresh the menu.

Examples

See the Menu.prg sample file in the \CLIP53\SOURCE\SAMPLE directory. This example demonstrates combining TopBarMenu, PopUpMenu, and MenuItem objects to create a menu with a number of available choices. See "Introduction to the Menu System" in the *Programming and Utilities Guide* for more information about using this class.

See Also

MenuItem class, MENUMODAL(), PopUpMenu class

TOTAL command

Summarize records by key value to a database (.dbf) file

Syntax

```
TOTAL ON <expKey> [FIELDS <idField list>]
   TO <xcDatabase>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

ON <expKey> defines the group of records that produce a new record in the target database file. To make the summarizing operation accurate, the source database file should be INDEXed or SORTed on this expression.

FIELDS <idField list> specifies the list of numeric fields to TOTAL. If the FIELDS clause is not specified, no numeric fields are totaled. Instead each numeric field in the target file contains the value for the first record matching the key expression.

TO <xcDatabase> is the name of the target file that will contain the copy of the summarized records. Specify this argument as a literal file name or as a character expression enclosed in parentheses. Unless otherwise specified, TOTAL assumes a .dbf extension.

<scope> is the portion of the current database file to TOTAL. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to TOTAL within the given scope.

Description

TOTAL is a database command that sequentially processes the current database file, summarizing records by the specified key value and copying them to a new database file. TOTAL works by first copying the structure of the current database file to *<xcDatabase>*, except for memo fields. It then sequentially scans the current database file within the specified scope of records. As each record with a unique *<expKey>* value is encountered, that record is copied to the new database file. The values of numeric fields specified in *<idField list>* from successive records with the same *<expKey>* value are added to fields with the same names in *<xcDatabase>*. Summarization proceeds until a record with a new key value is encountered. The process is then repeated for this record.

Since TOTAL processes the source database file sequentially, it must be INDEXed or SORTed in *<expKey>* order for the summarization to be correct.

To successfully TOTAL numeric fields, the source numeric fields must be large enough to hold the largest total possible for that numeric field. If not, a runtime error is generated.

Notes

- ***Deleted source records:*** If DELETED is OFF, deleted records in the source file are TOTALed. Records in the target *<xcDatabase>* inherit the deleted status of the first matching record in the source file, just as nontotaled fields inherit their values. If DELETED is ON, however, none of the deleted source records are TOTALed.

Examples

- In this example, a database file is TOTALed ON the key expression of the controlling index using a macro expression. When the macro expression is encountered, the expression is evaluated and the resulting character string is substituted for the TOTAL *<expKey>* argument:

```
USE Sales INDEX Branch NEW
TOTAL ON &(INDEXKEY(0)) FIELDS Amount TO Summary
```

Files

Library is CLIPPER.LIB.

See Also

AVERAGE, INDEX, SORT, SUM

TRANSFORM() function

Convert any value into a formatted character string

Syntax

```
TRANSFORM(<exp>, <cSayPicture>) → cFormatString
```

Arguments

<exp> is the value to be formatted. This expression can be any valid CA-Clipper data type except array, code block, and NIL.

<cSayPicture> is a string of picture and template characters that describes the format of the returned character string.

Returns

TRANSFORM() converts *<exp>* to a formatted character string as defined by *<cSayPicture>*.

Description

TRANSFORM() is a conversion function that formats character, date, logical, and numeric values according to a specified picture string that includes a combination of picture function and template strings. TRANSFORM() formats data for output to the screen or the printer in the same manner as the PICTURE clause of the @...SAY command.

- ***Function string:*** A picture function string specifies formatting rules that apply to the TRANSFORM() return value as a whole, rather than to particular character positions within *<exp>*. The function string consists of the @ character, followed by one or more additional characters, each of which has a particular meaning (see table below). If a function string is present, the @ character must be the leftmost character of the picture string, and the function string must not contain spaces. A function string may be specified alone or with a template string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

TRANSFORM() Functions

Function	Action
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays date in SET DATE format
E	Displays date in British format
R	Nontemplate characters are inserted
X	Displays DB after negative numbers
Z	Displays zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

- ***Template string:*** A picture template string specifies formatting rules on a character-by-character basis. The template string consists of a series of characters, some of which have special meanings (see table below). Each position in the template string corresponds to a position in the value of the *<exp>* argument. Because TRANSFORM() uses a template, it can insert formatting characters such as commas, dollar signs, and parentheses.

  Characters in the template string that have no assigned meanings are copied literally into the return value. If the @R picture function is used, these characters are inserted between characters of the return value; otherwise, they overwrite the corresponding characters of the return value. A template string may be specified alone or with a function string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

TRANSFORM() Templates

Template	Action
A,N,X,9,#	Displays digits for any data type
L	Displays logicals as "T" or "F"
Y	Displays logicals as "Y" or "N"
!	Converts an alphabetic character to uppercase
$	Displays a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Specifies a decimal point position
,	Specifies a comma position

Examples

- This example formats a number into a currency format using a template:

```
? TRANSFORM(123456, "$999,999")  // Result: $123,456
```

- This example formats a character string using a function:

```
? TRANSFORM("to upper", "@!")    // Result: TO UPPER
```

Files Library is CLIPPER.LIB.

See Also @...SAY, LOWER(), PAD(), STR(), UPPER()

TRIM() function

Remove trailing spaces from a character string

Syntax

```
TRIM(<cString>) → cTrimString
```

Arguments

<cString> is the character string to be copied without trailing spaces.

Returns

TRIM() returns a copy of *<cString>* with the trailing spaces removed. If *<cString>* is a null string ("") or all spaces, TRIM() returns a null string ("").

Description

TRIM() is a character function that formats character strings. It is useful when you want to delete trailing spaces while concatenating strings. This is typically the case with database fields which are stored in fixed-width format. For example, you can use TRIM() to concatenate first and last name fields to form a name string.

TRIM() is related to LTRIM(), which removes leading spaces, and ALLTRIM(), which removes both leading and trailing spaces. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADR(), and PADL() functions which center, right-justify, or left-justify character strings by padding them with fill characters.

Notes

- ***Space characters:*** The TRIM() function treats carriage returns, line feeds, and tabs as space characters and removes these as well.

Examples

- This is a user-defined function in which TRIM() formats city, state, and zip code fields for labels or form letters:

```
FUNCTION CityState(cCity, cState, cZip)
   RETURN TRIM(cCity) + ", " ;
    + TRIM(cState) + "  " + cZip
```

- In this example the user-defined function, CityState(), displays a record from Customer.dbf:

```
USE Customer INDEX CustName NEW
SEEK "Kate"
? CityState(City, State, ZipCode)
// Result: Athens, GA 10066
```

Files Library is CLIPPER.LIB.

See Also ALLTRIM(), LTRIM(), PAD(), RTRIM(), SUBSTR()

TYPE command

Display the contents of a text file

Syntax

```
TYPE <xcFile> [TO PRINTER] [TO FILE <xcOutFile>]
```

Arguments

<xcFile> is the name of the file, including extension, to be displayed to the screen. This argument may be specified as a literal file name or as a character expression enclosed in parentheses. *<xcFile>* must be specified with an extension if it has one.

TO PRINTER echoes the display to the printer.

TO FILE <xcOutFile> echoes the display to the specified file. *<xcOutFile>* may be specified either as a literal file name or as a character expression enclosed in parentheses. If no extension is specified, .txt is added.

Description

TYPE is a console command that displays the contents of a text file to the screen, optionally echoing the display to the printer and/or another text file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the TYPE invocation.

If *<xcFile>* is specified without a path and/or drive designator, TYPE searches the current DEFAULT directory, and then, the current PATH. If *<xcOutFile>* is specified without a path and/or drive designator, TYPE creates the file in the current DEFAULT directory.

TYPE performs no special formatting on the listing. There are no special headings or pagination when the output is sent to the printer.

To pause output, use Ctrl-S. Note that you cannot interrupt a listing with Esc.

Examples

- This example illustrates the TYPE command:

```
TYPE Main.prg TO PRINTER
```

Files Library is EXTEND.LIB.

See Also COPY FILE, SET DEFAULT, SET PATH, SET PRINTER

TYPE() function

Determine the type of an expression

Syntax

`TYPE(<cExp>) → cType`

Arguments

<*cExp*> is a character expression whose type is to be determined. <*cExp*> can be a field, with or without the alias, a private or public variable, or an expression of any type.

Returns

TYPE() returns one of the following characters:

TYPE() Return Values

Returns	Meaning
A	Array
B	Block
C	Character
D	Date
L	Logical
M	Memo
N	Numeric
O	Object
U	NIL, local, or static
UE	Error syntactical
UI	Error indeterminate

Description

TYPE() is a system function that returns the type of the specified expression. It can test expression validity as long as the expression uses CLIPPER.LIB functions and does not reference local or static variables, user-defined functions, or built-in functions supplied in EXTEND.LIB.

TYPE() is like VALTYPE() but uses the macro operator (&) to determine the type of the argument. This precludes the use of TYPE() to determine the type of local and static variables. VALTYPE(), by contrast, evaluates an expression and determines the data type of the return value. This lets you determine the type of user-defined functions as well as local and static variables.

Notes

- ***Array references:*** References to private and public arrays return "A." References to array elements return the type of the element.
- ***IF():*** To return the appropriate data type for an IF() expression, TYPE() evaluates the condition, and then, returns the type of the evaluated path. If either the IF() condition or the evaluated path are invalid, TYPE() returns "UE."
- ***Testing parameters:*** TYPE() can only test the validity of parameters received using the PARAMETERS statement. Testing a parameter declared as part of a FUNCTION or PROCEDURE declaration always returns "U" because local parameters do not have a symbol in the symbol table. To determine whether an argument was skipped or left off the end of the argument list, compare the parameter to NIL or use VALTYPE().
- ***User-defined and EXTEND.LIB functions:*** If a reference is made anywhere in an expression to a function not found in CLIPPER.LIB (a user-defined or EXTEND.LIB function), TYPE() returns "UI." If the user-defined function is not linked into the current program, TYPE() returns "U."

Examples

- These examples demonstrate various results from invocations of TYPE():

```
? TYPE('SUBSTR("Hi There", 4, 5)')      // Result: C
? TYPE("UDF()")                         // Result: UI
? TYPE('IF(.T., "true", 12)')           // Result: C
```

- This example shows two methods for testing for the existence and type of declared parameters:

```
FUNCTION TestParams
   PARAMETERS cParam1, nParam2
   IF cParam1 = NIL
      ? "First parameter was not passed"
      cParam1 := "Default value"
   ENDIF

   IF TYPE('nParam2') == "U"
      ? "Second parameter was not passed"
   ENDIF
   .
   . <statements>
   .
   RETURN NIL
```

Files Library is CLIPPER.LIB.

See Also VALTYPE()

UNLOCK command

Release file/record locks set by the current user

Syntax

```
UNLOCK [ALL]
```

Arguments

ALL releases all current locks in all work areas. If not specified, only the lock in the current work area is released.

Description

UNLOCK is a network command that releases file or record locks set by the current user. Use it when you want to release the current lock without setting a new lock. Both FLOCK() and RLOCK() release the current lock before setting a new one.

After an UNLOCK, an update to a shared database file and associated index and memo files becomes visible to DOS and other applications, but is not guaranteed to appear on disk until you perform a COMMIT or close the file.

Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information on the principles of locking and update visibility.

Notes

- ***SET RELATION:*** UNLOCK does not automatically release a record lock along a RELATION chain unless you UNLOCK ALL.

Examples

- This example attempts an update operation that requires a record lock. If the RLOCK() is successful, the record is updated with a user-defined function and the RLOCK() is released with UNLOCK:

```
USE Sales INDEX Salesman SHARED NEW
IF RLOCK()
    UpdateRecord()
    UNLOCK
ELSE
    ? "Record update failed"
    BREAK
ENDIF
```

Files Library is CLIPPER.LIB.

See Also DBUNLOCK(), DBUNLOCKALL(), FLOCK(), RLOCK(), SET RELATION, USE

UPDATE command

Update current database file from another database file

Syntax

```
UPDATE FROM <xcAlias>
   ON <expKey> [RANDOM]
   REPLACE <idField> WITH <exp>
   [, <idField2> WITH   <exp2>...]
```

Arguments

FROM <xcAlias> specifies the alias of the work area used to update records in the current work area. This argument may be specified either as a literal file name or as a character expression enclosed in parentheses.

ON <expKey> specifies the expression that defines matching records in the FROM work area.

REPLACE <idField> specifies a field in the current work area to replace with a new value.

WITH <exp> specifies the value to replace into the current field. You must reference any field contained in the FROM work area with the correct alias.

RANDOM allows records in the FROM database file to be in any order. If this option is specified, the current database file must be indexed on *<expKey>*.

Description

UPDATE is a database command that replaces fields in the current work area with values from another work area based on the specified key expression. UPDATE is designed to update only current work area records based on a one-to-one or one-to-many relation with the FROM work area. This means that UPDATE can only update records in the current work area with unique key values. When there is more than one instance of a key value, only the first record with the key value is updated. The FROM work area, however, can have duplicate key values.

There are two formulations of the command depending on whether the FROM work area records are sorted or indexed on *<expKey>* or not. If RANDOM is not specified, both the current work area and the FROM work area must be indexed or sorted in *<expKey>* order. If RANDOM is specified, the current work area must be indexed by *<expKey>*, but the FROM work area records can be in any order.

To use UPDATE in a network environment, the current database file must be locked with FLOCK() or USEed EXCLUSIVEly. The FROM database file may be used in any mode. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Notes

- ***Deleted records:*** If DELETED is OFF, deleted records in both source files are processed. Records in the file being updated retain their deleted status and are not affected by the deleted status of records in the FROM file. If DELETED is ON, however, no deleted records are processed from either source file.

Examples

- This example UPDATEs the Customer database file with outstanding invoice amounts:

```
USE Invoices NEW
USE Customer INDEX Customer NEW
UPDATE FROM Invoices ON Last;
    REPLACE Owed WITH Owed + Invoices->Amount RANDOM
```

Files

Library is CLIPPER.LIB.

See Also

DBCREATEIND(), INDEX, JOIN, REPLACE, SET UNIQUE*, SORT, TOTAL

UPDATED() function

Determine whether a GET changed during a READ

Syntax

```
UPDATED() → lChange
```

Returns

UPDATED() returns true (.T.) if data in a GET is added or changed; otherwise, it returns false (.F.).

Description

UPDATED() determines whether characters were successfully entered into a GET from the keyboard during the most current READ. Each time READ executes, UPDATED() is set to false (.F.). Then, any change to a GET entered from the keyboard sets UPDATED() to true (.T.) after the user successfully exits the GET. If the user presses Esc before exiting the first GET edited, UPDATED() remains false (.F.). Once UPDATED() is set to true (.T.), it retains this value until the next READ is executed.

Within a SET KEY or VALID procedure, you can change the current GET variable using the KEYBOARD command or by assigning a new value with one of the many assignment operators. Changing the variable with KEYBOARD is the same as if the user had entered the change directly from the keyboard, and UPDATED() is set accordingly. However, since UPDATED() reflects only those changes made from the keyboard, an assignment to the GET variable does not affect UPDATED().

Examples

- This example assigns field values from Customer.dbf to variables and edits them. If the user changes any of the values, the field variables for the current record are updated with the new values:

```
USE Customer NEW
CLEAR
MEMVAR->Customer = Customer->Customer
MEMVAR->Address = Customer->Address
@ 1, 1 SAY "Name:" GET MEMVAR->Customer
@ 2, 1 SAY "Address:" GET MEMVAR->Address
READ
//
IF UPDATED()
    Customer->Customer := MEMVAR->Customer
    Customer->Address := MEMVAR->Address
ENDIF
```

Files Library is CLIPPER.LIB.

See Also @...GET, READ, SET KEY

UPPER() function

Convert lowercase characters to uppercase

Syntax

`UPPER(<cString>) → cUpperString`

Arguments

<cString> is the character string to be converted.

Returns

UPPER() returns a copy of *<cString>* with all alphabetical characters converted to uppercase. All other characters remain the same as in the original string.

Description

UPPER() is a character function that converts lowercase and mixed case strings to uppercase. It is related to LOWER() which converts uppercase and mixed case strings to lowercase. UPPER() is related to the ISUPPER() and ISLOWER() functions which determine whether a string begins with an uppercase or lowercase letter.

UPPER() is generally used to format character strings for display purposes. It can, however, be used to normalize strings for case-independent comparison or INDEXing purposes.

Examples

- These examples illustrate the effects of UPPER():

```
? UPPER("a string")           // Result: A STRING
? UPPER("123 char = <>")      // Result: 123 CHAR = <>
```

- This example uses UPPER() as part of a case-independent condition:

```
USE Customer INDEX CustName NEW
LIST CustName FOR "KATE" $ UPPER(Customer)
```

- UPPER() is also useful for creating case-independent index key expressions:

```
USE Customer NEW
INDEX ON UPPER(Last) TO CustLast
```

- Later, use the same expression to look up Customers:

```
MEMVAR->Last = SPACE(15)
@ 10, 10 GET MEMVAR->Last
READ
SEEK UPPER(MEMVAR->Last)
```

Files Library is CLIPPER.LIB.

See Also ISLOWER(), ISUPPER(), LOWER()

USE command

Open an existing database (.dbf) and its associated files

Syntax

```
USE [<xcDatabase>
   [INDEX <xcIndex list>]
   [ALIAS <xcAlias>] [EXCLUSIVE | SHARED]
   [NEW] [READONLY]
   [VIA <cDriver>]]
```

Arguments

<xcDatabase> is the name of the database file to be opened and may be specified either as a literal file name or as a character expression enclosed in parentheses.

INDEX <xcIndex list> specifies the names of 1 to 15 index files to be opened in the current work area. Specify each index as a literal file name or as a character expression enclosed in parentheses. The first index in the list becomes the controlling index. If you specify an *<xcIndex>* as an expression and the value returned is spaces or NIL, it is ignored.

ALIAS <xcAlias> specifies the name to associate with the work area when the database file is opened. You may specify the alias name as a literal name or as a character expression enclosed in parentheses. A valid *<xcAlias>* may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). Within a single application, CA-Clipper will not accept duplicate aliases. If this clause is omitted, the alias defaults to the database file name.

EXCLUSIVE opens the database file for nonshared use in a network environment. All other users are denied access until the database file is CLOSEd.

SHARED opens the database file for shared use in a network environment. Specifying this clause overrides the current EXCLUSIVE setting.

NEW opens <*xcDatabase*> in the next available work area making it the current work area. If this clause is not specified, <*xcDatabase*> is opened in the current work area.

READONLY opens <*xcDatabase*> with a read-only attribute. This lets you open database files marked read-only. If you cannot open the <*xcDatabase*> this way, a runtime error is generated. If this clause is not specified, <*xcDatabase*> is opened as read-write.

VIA <cDriver> specifies the replaceable database driver (RDD) with which to process the current work area. <*cDriver*> is the name of the RDD specified as a character expression. If <*cDriver*> is specified as a literal value, it must be enclosed in quotes.

If the VIA clause is omitted, the DBFNTX driver is used by default. Note that if the specified driver is not linked, an unrecoverable error occurs.

In no arguments are specified, the database file open in the current work area is closed.

Description

USE opens an existing database (.dbf) file, its associated memo (.dbt) file, and optionally associated index (.ntx or .ndx) file(s) in the current or the next available work area. In CA-Clipper, there are 250 work areas with a maximum of 255 total files open in DOS 3.3 and above. Before USE opens a database file and its associated files, it closes any active files already open in the work area. When a database file is first opened, the record pointer is positioned at the first logical record in the file (record one, if there is no index file specified).

In a network environment, you may open database files as EXCLUSIVE or SHARED. EXCLUSIVE precludes the USE of the database file by other users until the file is closed. SHARED allows other users to USE the database file for concurrent access. If the database file is SHARED, responsibility for data integrity falls upon the application program.

In CA-Clipper, FLOCK() and RLOCK() are the two basic means of denying other users access to a particular work area or record. If a USE is specified and neither EXCLUSIVE nor SHARED is specified, the database file is opened according to the current EXCLUSIVE setting. In CA-Clipper, all USE commands should explicitly specify how the database file is to be opened, EXCLUSIVE or SHARED. The implicit open mode specified by SET EXCLUSIVE is supplied for compatibility purposes only and not recommended.

Opening a database file in a network environment requires some special handling to be successful. First, attempt to USE the database file without specifying the INDEX list. Then, test for the success of the operation using NETERR(). If NETERR() returns false (.F.), the open operation succeeded and you can SET INDEX TO the index list. A USE will fail in a network environment if another user has EXCLUSIVE USE of the database file. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information on opening files in a network environment.

You can open index files with USE or SET INDEX. The first index in the list of indexes defines the current ordering of records when they are accessed. This index is referred to as the controlling index. You can change the current controlling index without closing any files by using the SET ORDER command.

To close a database and its associated files in the current work area, specify USE or CLOSE with no arguments. To close database files in all work areas, use CLOSE DATABASEs. To close index files in the current work area without closing the database file, use CLOSE INDEX or SET INDEX TO with no arguments.

Refer to the "Basic Concepts" chapter in the *Programming and Utilities Guide* for more information about the CA-Clipper database paradigm.

Notes

- ***Setting the maximum open files:*** Control of the number of file handles available to a CA-Clipper application is controlled by a combination of the CONFIG.SYS FILES command, and the F parameter of the CLIPPER environment variable. The F parameter specifies the maximum number of files that can be opened at any one time within the current CA-Clipper program. CA-Clipper determines the number of files that can be opened using the smaller of the two parameters. For example, if the FILES command is set to 120 and the F parameter is set to 50, the maximum number of files that can be opened is 50. In a network environment, file handles also need to be set in the network configuration file.

  The file limit is controlled by the operating system. Under DOS versions less than 3.3, the maximum number of files that can be opened at one time is 20 files. In DOS versions 3.3 and greater, the maximum limit is 255 files.

- ***Opening the same database file in more than one work area:*** Although opening a database file in more than one work area is possible in a network environment, this practice is strongly discouraged. If done, each file must be opened with a different alias, otherwise a runtime error will occur.

- ***Opening two database files with the same names, in different directories:*** Although opening two database files with the same names in different directories is possible, the database files MUST have unique alias names; otherwise, a runtime error will occur.

Examples

- This example opens a shared database file with associated index files in a network environment. If NETERR() returns false (.F.), indicating the USE was successful, the indexes are opened:

```
USE Accounts SHARED NEW
IF !NETERR()
   SET INDEX TO AcctNames, AcctZip
ELSE
   ? "File open failed"
   BREAK
ENDIF
```

- This example opens a database file with several indexes specified as extended expressions. Note how the array of index names is created as a constant array:

```
xcDatabase = "MyDbf"
xcIndex = {"MyIndex1", "MyIndex2", "MyIndex3"}
USE (xcDatabase) INDEX (xcIndex[1]), ;
      (xcIndex[2]), (xcIndex[3])
```

Files Library is CLIPPER.LIB.

See Also CLOSE, DBSELECT(), DBSETINDEX(), DBUSEAREA(), NETERR(), SELECT, SET INDEX, USED()

USED() function

Determine whether a database file is in USE

Syntax

```
USED() → lDbfOpen
```

Returns

USED() returns true (.T.) if there is a database file in USE; otherwise, it returns false (.F.).

Description

USED() is a database function that determines whether there is a database file in USE in a particular work area. By default, USED() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

- This example determines whether a database file is in USE in the current work area:

```
USE Customer NEW
? USED()              // Result: .T.
CLOSE
? USED()              // Result: .F.
```

Files Library is CLIPPER.LIB.

See Also ALIAS(), SELECT, SELECT(), USE

VAL() function

Convert a character number to numeric type

Syntax

VAL(<*cNumber*>) → *nNumber*

Arguments

<*cNumber*> is the character expression to be converted.

Returns

VAL() returns <*cNumber*> converted to a numeric value including decimal digits.

Description

VAL() is a character conversion function that converts a character string containing numeric digits to a numeric value. When VAL() is executed, it evaluates <*cNumber*> until a second decimal point, the first non-numeric character, or the end of the expression is encountered. Leading spaces are ignored. When SET FIXED is ON, VAL() returns the number of decimal places specified by SET DECIMALS, rounding <*cNumber*> if it is specified with more digits than the current DECIMALS value. As with all other functions that round, digits between zero and four are rounded down, and digits between five and nine are rounded up. When SET FIXED is OFF, VAL() returns the number of decimal places specified in <*cNumber*>.

VAL() is the opposite of STR() and TRANSFORM(), which convert numeric values to character strings.

Examples

- These examples illustrate VAL() with SET FIXED ON and SET DECIMALS TO 2:

```
SET DECIMALS TO 2
SET FIXED ON
//
? VAL("12.1234")        // Result:  12.12
? VAL("12.1256")        // Result:  12.13
? VAL("12A12")          // Result:  12
? VAL("A1212")          // Result:     0
? VAL(SPACE(0))         // Result:     0
? VAL(SPACE(1))         // Result:     0
? VAL(" 12.12")         // Result:  12.12
```

Files

Library is CLIPPER.LIB.

See Also

ROUND(), SET DECIMALS, SET FIXED, STR(), TRANSFORM()

VALTYPE() function

Determine the data type returned by an expression

Syntax

VALTYPE(<*exp*>) → *cType*

Arguments

<*exp*> is an expression of any type.

Returns

VALTYPE() returns a single character representing the data type returned by <*exp*>. VALTYPE() returns one of the following characters:

VALTYPE() Return Values

Returns	Meaning
A	Array
B	Block
C	Character
D	Date
L	Logical
M	Memo
N	Numeric
O	Object
U	NIL

Description

VALTYPE() is a system function that takes a single argument, evaluates it, and returns a one-character string describing the data type of the return value. It is similar to TYPE(), but differs by actually evaluating the specified argument and determining the type of the return value. For this reason, you can determine the type of local and static variables, user-defined functions, and EXTEND.LIB functions. TYPE(), by contrast, uses the macro operator (&) to evaluate the type of its argument. Note that if the argument does not exist, an error ("undefined error") will occur, unlike TYPE which will return "U."

Examples

- These examples show the return values for several data types:

```
? VALTYPE(1)              // Result: N
? VALTYPE("GOOB")         // Result: C
? VALTYPE(NIL)            // Result: U
? VALTYPE(array)          // Result: A
? VALTYPE(block)          // Result: B
```

Files Library is CLIPPER.LIB.

See Also TYPE()

VERSION() function

Returns CA-Clipper version

Syntax

```
VERSION() → cVersion
```

Returns

VERSION() returns the version number of the CA-Clipper library, EXTEND.LIB, as a character value.

Description

VERSION() is an environment function that returns the version of the CA-Clipper library, EXTEND.LIB.

Files

Library is EXTEND.LIB.

WAIT* command

Suspend program processing until a key is pressed

Syntax

```
WAIT [<expPrompt>] [TO <idVar>]
```

Arguments

<expPrompt> is an expression of any data type displayed as a prompt. If no *<expPrompt>* is specified, the default prompt displayed is: "Press any key to continue..."

TO <idVar> is the variable, of any storage class, that holds the value of the key pressed as a character value. If *<idVar>* does not exist or is not visible, it is created as a private variable and then assigned the character value.

Description

WAIT is a console command and wait state that displays a prompt after sending a carriage return/line feed to the screen. It then waits for the user to press a key. If the TO clause is specified, *<idVar>* is assigned the keystroke as a character value. If an Alt or Ctrl key is pressed, WAIT assigns CHR(0) to *<idVar>*. Non-alphanumeric values entered by pressing an Alt-keypad combination assign the specified character. If the character can be displayed, it is echoed to the screen. Function keys are ignored unless assigned with SET FUNCTION or SET KEY.

WAIT is a compatibility command and, therefore, is not recommended for general usage. It is superseded by both @...GET/READ and INKEY() for getting single character input.

Notes

- ***WAITing without a prompt:*** To pause execution without displaying a prompt, specify WAIT, null string (""), or INKEY(0). The latter is recommended since it does not disturb the current screen cursor position.

Examples

- This example illustrates how to store the WAIT keystroke as an array element:

```
aVar := ARRAY(6)

WAIT "Press a key..." TO aVar[1]
? aVar[1]                // Result: key pressed in
                         // response to WAIT
? aVar[2]                // Result: NIL
? VALTYPE(aVar)          // Result: A
? VALTYPE(aVar[1])       // Result: C
```

Files Library is CLIPPER.LIB.

See Also @...GET, ACCEPT, INKEY(), INPUT, MENU TO

WORD()* function

Convert CALL command numeric parameters from double to integer values

Syntax

```
WORD(<nNumber>) → NIL
```

Arguments

<nNumber> is the numeric value to be converted to an integer specified in the range of plus or minus 32,767, inclusive.

Returns

Used as an argument for the CALL command, WORD() returns an integer. In all other contexts, it returns NIL.

Description

WORD() is a numeric conversion function that converts numeric parameters of the CALL command from double to integer values. WORD() is a compatibility command and, therefore, not recommended. Both the CALL command and the WORD() function are superseded by facilities provided by the Extend System. Refer to the "Using the Extend System" chapter in the *Technical Reference Guide* for more information.

Examples

- This example uses WORD() as an argument of the CALL command:

```
CALL Cproc WITH WORD(30000), "Some text"
```

Files

Library is CLIPPER.LIB.

YEAR() function

Convert a date value to the year as a numeric value

Syntax

```
YEAR(<dDate>) → nYear
```

Arguments

<dDate> is the date value to be converted.

Returns

YEAR() returns the year of the specified date value including the century digits as a four-digit numeric value. The value returned is not affected by the current DATE or CENTURY format. Specifying a null date (CTOD("")) returns zero.

Description

YEAR() is a date conversion function that converts a date value to a numeric year value. Use it in calculations for things like periodic reports or for formatting date displays.

YEAR() is a member of a group of functions that return components of a date value as numeric values. The group includes DAY() and MONTH() which return the day and month values as numeric values.

Examples

- These examples illustrate YEAR() using the system date:

```
? DATE()                    // Result: 09/20/90
? YEAR(DATE())              // Result: 1990
? YEAR(DATE()) + 11         // Result: 2001
```

- This example creates a user-defined function using YEAR() to format a date value in the following form: month day, year.

```
? Mdy(DATE())               // Result: September 20, 1990

FUNCTION Mdy( dDate )
   RETURN CMONTH(dDate) + " " + ;
          LTRIM(STR(DAY(dDate))) ;
      + "," + STR(YEAR(dDate))
```

Files Library is CLIPPER.LIB.

See Also DAY(), MONTH()

ZAP command

Remove all records from the current database file

Syntax

```
ZAP
```

Description

ZAP is a database command that permanently removes all records from files open in the current work area. This includes the current database file, index files, and associated memo file. Disk space previously occupied by the ZAPped files is released to the operating system. ZAP performs the same operation as DELETE ALL followed by PACK, but is almost instantaneous in comparison.

To ZAP in a network environment, the current database file must be USEd EXCLUSIVEly. Refer to the "Network Programming" chapter in the *Programming and Utilities Guide* for more information.

Examples

- This example demonstrates a typical ZAP operation in a network environment:

```
USE Sales EXCLUSIVE NEW
IF !NETERR()
    SET INDEX TO Sales, Branch, Salesman
    ZAP
    CLOSE Sales
ELSE
    ? "Zap operation failed"
    BREAK
ENDIF
```

Files

Library is CLIPPER.LIB.

See Also

DELETE, PACK, USE

Glossary

Abbreviation
(preprocessor) A source token whose leftmost characters exactly match the leftmost characters of a keyword in a translation directive. Abbreviations must be at least four characters in length.

Activation
(procedure and function) The invocation and execution process for procedures. Each call to a procedure is referred to as an *activation* of that procedure. If the procedure in turn calls another procedure (or calls itself recursively), a *new activation* is said to have occurred. The earlier activation is then referred to as a *pending activation*.

Pending activations are often referred to as *higher level activations* or *higher level procedures*. When one procedure calls another (i.e., creates a new activation), the latter is often referred to as a *lower level activation* or a *lower level procedure*. (*See also:* Activation Record, Activation Stack, Function, Procedure)

Activation Record
(procedure and function) An internal data structure that contains information pertaining to an activation. (*See also:* Activation, Activation Stack)

Activation Stack
(procedure and function) An internal data structure that contains an activation record for the current activation and all pending activations. (*See also:* Activation, Activation Record, Stack)

Active Window
(debugger) The window to which all keystrokes (except those valid in the Command Window) apply. An active window is indicated by a highlighted border. The Tab and Shift+Tab keys are used to select the next and previous window, respectively.

Algorithm
(algorithm) A set of rules and/or a finite series of steps that will accomplish a particular task. (*See also:* Sequence, Selection, Iteration)

Alias
(database) The name of a work area; an alternate name given to a database file. Aliases are often used to give database files descriptive names and are assigned when the database file is opened. If no alias is specified when the database file is USEd, the name of the database file becomes the alias. (*See also:* Work Area)

An alias can be used to reference both fields and expressions (including user-defined functions). In order to reference an expression using an alias, the expression must be enclosed in parentheses.

Animate Mode
(debugger) The mode of execution in which an application runs one line at a time until a breakpoint or tracepoint is reached, with the execution bar moving to each line as it is executed.

Application
(configuration) A program designed to execute a set of interrelated tasks. Typically referring to a system designed to address a particular business purpose (e.g., Order Entry/Inventory/Invoicing, a document tracking database, or an insurance claims calculator).

Application Buttons
(Workbench) The buttons in the Application Browser pictorially represent each application in the repository. They visually convey the following information: name of the application and compilation status. (*See also:* Compilation Status)

Application Programming Interface (API)
(general) A set of functions that allow direct communication between extend functions and the CA-Clipper Virtual Memory Manager (VMM). It permits safe access to large amounts of memory.

The API enables programmers to interact with and control the features built into CA-Clipper 5.3 and to add specific functionality to C and Assembly language modules that call CA-Clipper routines.

Argument
(variable) Generally, a value or variable supplied in a function or procedure call, or an operand supplied to an operator. In function and procedure calls, arguments are often referred to as *actual parameters*. (*See also:* Parameter)

Array
(array) A data structure that contains an ordered series of values called *elements*. The elements of an array are referred to by ordinal number: the first element is number 1, the second is number 2, etc. A numeric expression used to specify an element of an array is referred to as a *subscript* or *index*. In CA-Clipper, the elements of an array may be values of any type, including references to other arrays. (*See also:* Array Reference, Nested Array, Subarray, Subscript)

Array Functions
(array) Those functions that specifically perform their tasks on arrays. (*See also:* Array, Function, Element, Subscript)

Array Iterator
(array) A function that traverses an array, performing an operation on each element. (*See also:* Array)

Array Reference
(array) A special data value that allows access to an array. In CA-Clipper, program variables and array elements cannot directly contain arrays; they may, however, contain array references. A variable that contains a reference to a particular array is said to *refer to* that array, and the array's elements may be accessed by applying a subscript to the variable.

If the value of a variable containing an array reference is assigned to a second variable, the second variable will contain a copy of the array reference; both variables then refer to the same array, and the array's elements may be accessed by applying a subscript to either variable. (*See also:* Array, Multi-dimensional Array, Nested Array, Subarray, Subscript)

ASCII
(general) An acronym for the American Standard Code for Information Interchange, the agreed upon standard for representing characters (alphabetic, symbolic, etc.) in the memory of the computer.

Assignment
(expression) The act of copying a new value into a variable. In CA-Clipper this is done with the simple assignment operators (=) and (:=), or the compound operators (+=, -=, *=, **=).

Attribute
(database) As a formal DBMS term, refers to a column or field in a table or database file. (*See also:* Column, Field)

Auto Layout
(Workbench) CA-Clipper's Auto Layout feature allows you to (*See also:* Form Editor, Menu Editor):

- Add predefined controls to a data window, based upon the fields in the associated data server using the Auto Layout toolbar button from within the Window Editor
- Add one or more standard, predefined menus to the current menu tree using the Auto Layout toolbar button from within the Menu Editor

Background Color
(user-interface) The color that appears behind displayed text of another color (the foreground color). Though background color and foreground color are usually different, you may render text invisible by choosing the same color for both. (*See also:* Foreground color)

Beginning of File
(database) The top of the database file. In CA-Clipper there is no beginning of file area or record. Instead, it is indicated by BOF() returning true (.T.) if an attempt is made to move the record pointer above the first record in the database file or the database file is empty.

Binary File
(file) A file that contains an unformatted sequence of bytes. Carriage-return, line-feed, or end of file characters have no special meaning in a binary file. Binary files include executable files, graphics files, or data files. (*See also:* Text File)

Binary Notation
(general) A coding system that represents values with the digits 1 or 0. The numerical base upon which computers are modeled. Also referred to as base 2, it is a mathematical representation of the state of a logic element. A 0 represents an *off* state, and a 1 indicates an *on* state. A sequence of eight binary digits, eight on's and off's, express one byte of program or data. (*See also:* Bit, Byte)

Binary Operator
(expression) An operator that operates on two operands. For example, the addition operator. (*See also:* Operator)

Bit
(general) One binary digit, the smallest element of code. Eight consecutive bits form one byte.

Blockify
(preprocessor) To change an expression in the source text into a code block definition. Blockifying is accomplished by surrounding the source text with braces and placing an empty block parameter list (a pair of vertical bars) just inside the braces. When the resulting code block is evaluated at runtime, the original expression will be evaluated.

Blockify Result Marker
(preprocessor) A result marker of the form *<{id}>*. *id* must correspond to the name of a match marker. A blockify result marker specifies that the corresponding source text is to be blockified. If the matched source text is a list of expressions, each expression in the list is individually blockified. If no source text matched the corresponding match marker, an empty result is produced.

Branching
(language) Changing the sequence of execution in a program. Execution normally proceeds in sequence from the top of a function or procedure to the bottom. When control is transferred to a statement that is not in sequence, execution is said to have *branched*.

Breakpoint
(debugger) A point at which an application pauses execution and returns control to the debugger.

Browser
(Workbench) A *browser* is a tool in the CA-Clipper Workbench that provides a convenient and organized way to view the data that is currently stored in your repository.

In CA-Clipper, you can browse:

- Applications
- Modules
- Entities
- Errors

Browse/Form View
(Workbench) Use the Browse/Form View toolbar button to toggle between browse view and form view. *Browse view* displays multiple records for each field in a data window, always maintaining one record as the "current" row, reflecting the data server's current position. *Form view*, on the other hand, displays only one record—the current one—at any given time.

Buffer
(general) A temporary data storage location in memory. As an example, a *disk input-output buffer* is an area of memory that stores data read from the disk to temporary locations while processing it.

Byte
(general) Eight bits of data, the smallest unit of information stored in the computer's memory. As an example, one byte is required to represent one ASCII character.

Calling Convention
(general) The Microsoft Mixed-Language Programming Guide defines calling convention as "the way a language implements a call."

Calling Program
(procedure and function) The procedure or user-defined function that transferred control to the currently executing procedure or function. When the current procedure or user-defined function terminates with a RETURN statement, the program regains control.

Call Stack
(debugger) A list containing the names of all pending activations at the current point in an application.

Callstack Window
(debugger) The window in which the call stack is displayed.

Cell
(database) In a table, a cell is the intersection of a Row and a Column. (*See also:* Column, Row, Table)

Character
(general) A letter, digit, punctuation mark, or special symbol stored and processed by the computer. (*See also:* ASCII)

(data type) A special data type consisting of one or more values in the IBM extended character set. Characters can be grouped together to form strings. The maximum size of a character string in CA-Clipper is 65,534 bytes. (*See also:* String)

Character Functions
(expression) Those functions that act upon individual characters or strings of ASCII characters in the performance of their tasks. (*See also:* ASCII, Function, String)

Child
(Workbench) A *child* is a submenu item within a menu structure. Use the Demote Item toolbar button to demote a menu item to the child level in a menu's hierarchy. (*See also:* Sibling)

Class
(object-oriented) A class defines the variables contained in an object and the operations applied when the object receives a message. Every object is an instance of a class and responds to messages of that class. Each object, however, has its own copy of the variables specified in the class definition. New objects are created in CA-Clipper by calling a special function that begins with the class name followed by the *New* suffix. (*See also:* Object, Instance Variable, Message, Method)

Clause
(command) An optional or required section of a CA-Clipper command beginning with a keyword that modifies or enhances the command.

Code Block
(data type) A special data type that refers to a piece of compiled program code. In a program, the source code that specifies the creation of a code block.

Code Window
(debugger) The window in which source code is displayed.

Collapse/Expand Data

(Workbench) To *collapse* an item when viewing information (for example, in the Source Code Editor) means to compress it, "hiding" any additional information that is subordinate to that item. At any time you can *expand* a collapsed item (thereby redisplaying its subordinate data).

In CA-Clipper, the following types of items can be collapsed and expanded:

- Branches in tree browsers (like Entity Browsers)
- Entities in the Source Code Editor
- Menus in the Menu Editor

Collision

(network) An attempt by more than one user (typically on a networked, multi-user system) to update a database simultaneously, usually resulting in data corruption. Generally due to poor implementation of the code to support multi-user operations. (*See also:* LAN, Local Area Network, Deadly Embrace)

Column

(database) A database term used to describe a field in a table or database file. (*See also:* Field)

(user-interface) A numeric value that represents a position on the display screen or on the printed page.

Command

(command) A statement to be translated by the CA-Clipper preprocessor into source code that will perform a particular operation. All CA-Clipper commands are defined in the standard header file, Std.ch, located in \CLIP53\INCLUDE. Also, the preprocessor directives that define a command. (*See also:* Statement, Header files, Std.ch)

Command Window

(debugger) The window in which commands are displayed and entered.

Comment

(language) Text in a source program that is ignored by the compiler. Usually used to make descriptive comments about the surrounding source code.

Compilation Status

(Workbench) The *compilation status* of items in the repository is denoted in the various browsers with either LED or symbol indicators:

- *Green* (or a check mark) indicates that the application (or module) has been compiled successfully, contains an executable file, and is ready to run provided that the .EXE has not been deleted from DOS
- *Red* (or X) indicates compilation errors or warnings
- *Blue* (or !) indicates that compilation has been successful, but there are compiler warnings
- *Yellow* (or ?) indicates that the application (or module) is in an indeterminate state: either it is new and needs to be compiled, or has been modified and needs to be recompiled

Compiler
(program) A program that translates source code output from the preprocessor into object code. The resulting object file can then be linked to produce an executable program using a linker. (*See also:* Linker, Object File, Program File)

Concatenate
(expression) To combine two groups of character data together by placing them in a sequence to form a new string of characters. (*See also:* Data Type)

Concurrency
(database) The degree to which data can be accessed by more than one user at the same time.

Condition
(command, database, expression) A logical expression that determines whether an operation will take place. With database commands, a logical expression that determines what records are included in an operation. Conditions are specified as arguments of the FOR or WHILE clause. (*See also:* Scope)

Conditional Compilation
(preprocessor) Selective exclusion by the preprocessor of certain source code. The affected source code is bracketed by the #ifdef and #endif directives. It is excluded if the argument in the #ifdef directive is not a #defined identifier.

Console Input/Output
(user-interface) Operation of the keyboard and display that emulates a simple typewriter-like interface. Console input echoes each key typed and provides processing for the Backspace and Return keys. Console output wraps to the next line when the output reaches the right edge of the visible display, and scrolls the display when the output reaches the bottom of the visible display. (*See also:* Full-screen Input/Output)

Constant
(variable) The representation of an actual value. For example, .T. is a logical constant, *string* is a character constant, 21 is a numeric constant. There are no date and memo constants.

Constant Array
(array) *See* Literal Array.

Control Structure
(language) Any program structure that alters the flow of program control. In CA-Clipper, these include:

- BEGIN SEQUENCE...END
- DO WHILE...ENDDO
- DO CASE...ENDCASE
- FOR...NEXT
- IF...ENDIF

Controlling Order
(database) The active order (index) for a particular work area. Only one order may control a work area at any time, and it controls the order in which the database is accessed during paging and searching.

Controlling/Master Index

(database) The index currently being used to refer to records by key value or sequential record movement commands. (*See also:* Index, Natural Order)

Conversion Functions

(expression) Generally referring to a category of functions whose purpose is to change one data type to another (e.g., to change a number or a date to a character string).

Cursor

(user-interface) An on-screen indicator used to show the current keyboard input focus, displayed as a block or underline character. The cursor moves in response to characters or control keys typed by the user. (*See also:* Highlight, Input Focus)

(debugger) The cursor indicates the current line and/or column position in the active window or dialog box. Note that some windows, such as the Monitor Window, do not utilize the cursor. When a window that does not utilize the cursor is active, the cursor appears in the Code Window.

Data Form

(user-interface) A data form is a form or data entry screen associated with a data server. It is *data-aware*—it "knows" about the data with which it is intended to operate via properties you specify for each control in the form. (*See also:* Auto Layout, Form Editor)

Data Independence

(modular programming) The technique of writing a program, function, or procedure in such a way that it will be able to perform its operation on data supplied to it without a *built-in* description of the data format.

Data Server

(Workbench) A data server is a high-level, abstract entity designed as a consistent GUI interface for Xbase database files. The data server acts as a database describer, defining its file name, its fields (or columns), the order in which it is accessed, etc. It provides a mechanism for extending field definitions beyond the basic name, type, and length information stored in the database file structure—this last part is accomplished using a *field spec*. (*See also:* Field Specification (Field Spec))

Data Store

(file, general) An area of memory or disk in which data is *stored* and from which it is retrieved while being processed.

Data Type

(data type) The category of a data value. A data type is distinguished by the set of allowable values for that type, the set of operators that can be applied, and the storage format used to represent these values. In CA-Clipper, the following data types are defined: character, numeric, date, logical, array, object, code block, and NIL. Program variables may contain values of any type. Database field variables are limited to character, numeric, date, logical, and a special type called memo which is treated the same as character.

Database
(database) An aggregation of related operational data used by an application system. A database can contain one or more data files or tables. (*See also:* Field, Record, Tuple, View)

Date Functions
(expression) Functions that operate on date values (as opposed to character, numeric or other values). (*See also:* Function)

Date Type
(data type) A special data type consisting of digits to store year, month, and day values. Operations on date values are based on chronological values.

DB Server Editor
(Workbench) Use the Source Code Editor to create a data server, import the structure of an existing database file, and specify its properties and fields. (*See also:* Data Server, FieldSpec Editor, Visual Editor)

DBMS
(database) An acronym for the term *database management system*. A DBMS is a software system that mediates access to a database through a data manipulation language.

DDL
(general) Data Description Language. A set of symbols and words, and the rules governing their use as meaningful descriptions of the data components and their relationships within a database. Besides describing the elements and their relationships, the DDL also specifies the sorting and search fields.

Deadly Embrace
(network) In a multi-user database management system, the result of an unforeseen flaw that has allowed a situation wherein two or more simultaneous operations cannot be completed because each has instigated locks on files or records that the other needs to complete its task.

Debugger
(debugger) A tool used to track down errors in a program.

Debugging
(error handling) A phase of software development where errors are identified and fixed.

Declaration
(variable) A statement used by the compiler to define a variable, procedure, or function identifier. The scope of the declaration is determined by the position of the declaration statement in the source file. (*See also:* Identifier, Scope)

Decrement
(expression) To decrease a value by a fixed amount, usually one. In CA-Clipper, the decrement operator (--) can be used to decrement a numeric value in a variable. (*See also:* Assignment, Increment)

Define
(preprocessor) To define an identifier to the preprocessor, using the #define directive, and optionally specify text to be substituted for occurrences of the identifier.

Delimited File
(file, database) A text file that contains variable-length database records with each record separated by a carriage-return/line feed pair (CHR(13) + CHR(10)) and terminated with an end of file mark (CHR(26)). Each field within a delimited file is variable length, not padded with either leading or trailing spaces, and separated by a comma. Character strings are optionally delimited to allow for embedded commas. (*See also:* Database File, Text File, SDF File)

Delimiter
(general) A character or other symbol that marks a boundary.

Desktop
(user-interface) The background area of the screen on which all windows, icons, and dialog boxes appear.

Destination
(expression) The variable or array element to receive data in an assignment. (*See also:* Assignment)

(general) The work area, file, or device to which data is sent.

Device
(general) Either an actual physical component of the computer system such as a printer or a DOS *handle* that refers to it (e.g., PRN:), or a *logical device* that behaves and is addressed the same way as a *physical device* (e.g., a print spooler).

Dialog box
(debugger) A box displayed from within the debugger whenever further input is required.

Dimension
(array) The maximum number of subscripts required to specify an array element. For example, a two-dimensional array must have two subscripts, a three-dimensional array must have three subscripts, and so on. (*See also:* Subscript)

Directive
(preprocessor) An instruction to the preprocessor. Preprocessor directives must appear on a separate line, and must begin with a hash mark (#). Their scope or effect extends from the point where they are encountered to the end of the source file in which they appear.

Directory
(file) The major operating system facility for cataloging files. A directory contains a list of files and references to child directories (subdirectories), and is identified by name. Directories can be nested forming a hierarchical tree structure. The operating system provides a number of facilities that allow users to create and delete directories. (*See also:* Disk, File, Path, Volume)

Disk
A magnetic storage medium designed for long-term storage. Disks come in two varieties: hard disks (fast but fixed) or floppy disks (slow but removable). A disk can be partitioned into multiple volumes, each containing a tree-structured directory system that holds files accessible by programs. (*See also:* Volume, Directory, File)

DLL
(general) Abbreviation for a Microsoft Windows dynamic link library.

DML
(database) Data Manipulation Language. The set of commands and functions that control change and movement operations like input/output, reporting and sorting on data elements in a database.

Drive
(file) A disk drive or a letter (normally followed by a colon) that designates a disk drive. On most computers, the letters A and B refer to floppy disk drives; other letters refer to fixed disk drives or *logical* drives (e.g., fixed disk partitions or network drives).

Dumb Stringify Result Marker
(preprocessor) A result marker of the form #<*{id}*>. *id* must correspond to the name of a match marker. A dumb stringify result marker specifies that the corresponding source text is to be enclosed in quotes. If the matched source text constitutes a list of expressions, each expression in the list is individually stringified. If no source text was matched, an empty pair of quotes is produced.

Dynamic
(general) Used generically to refer to data or algorithms that change with time. Often used specifically to describe algorithms that automatically adjust to prevailing conditions.

Dynamic Overlay
(linker) A portion (memory page) of program code that can be moved into and out of memory on a least recently used basis. Dynamic overlays are allocated in 1K increments and the movement of code is managed by the overlay manager at runtime.

Dynamic Scoping
(variable) A method of determining an item's existence or visibility based on the state of a program during execution. Example: A CA-Clipper public variable may or may not be visible within a particular function, depending on whether the variable has been created and whether a previously called function has obscured it by creating a private variable with the same name. (*See also:* Lexical Scoping, Scope of a Variable)

Element
(array) A component unit of an array, usually referred to by a numeric subscript or index. (*See also:* Array, Subscript)

EMM
(memory) Expanded Memory Manager. A driver that provides a hardware-independent interface between application software and the expanded memory hardware.

Empty Result
(preprocessor) An absence of result text; the effect of certain result markers when the corresponding match marker did not match any source text (but when the translation directive as a whole was matched). An empty result simply implies that no result text is written to output.

EMS Memory
(memory) Expanded Memory Specification, or formally, the Lotus/Intel/Microsoft Expanded Memory Specification. EMS Memory is a functional definition of a bank-switched memory-expansion subsystem. This system consists of hardware expansion modules and a resident driver program.

Encapsulation
(modular programming) Generically, the design of a function or program that obeys the principle of information hiding. A function or program is said to be encapsulated if other programs or functions have no knowledge of its inner workings. A data structure is said to be encapsulated if knowledge of its internal organization is limited to a single function or module. (*See also:* Information Hiding, Lexical Scoping, Modularity, Side Effect)

End of File
(database) The bottom of a database file. In CA-Clipper, this is LASTREC() + 1 and is indicated by EOF() returning true (.T.).

Enhanced Color
(user-interface) The color used to display GETs or PROMPTs (if INTENSITY is ON). (*See also:* Standard Color)

Entity
(Workbench) In CA-Clipper, *applications* consist of *modules*, which consist of *entities*.

An *entity* is a component that has a distinct name and can be edited. Some examples are: functions, procedures, globals, constants, forms, and menus. (*See also:* Module)

Environment Variables
(configuration) Operating system variables that can be used to communicate configuration information to executable programs. Environment variables are manipulated using the DOS SET command. The CA-Clipper compiler and linker respond to certain environment variables. CA-Clipper programs can inspect the settings of environment variables using the GETENV() function.

Error
(error handling) The presence of some element of an operation that does not satisfy the requirements of the operation. An error causes failure when encountered, which in turn raises an exception. (*See also:* Exception, Failure, Runtime Error)

Error Handling
(error handling) The concept of including code in a program so that exceptions to normal operational states that occur during the program execution can be anticipated and dispatched with the least possible detrimental consequences to the use of the program and the data being worked on. (*See also:* Exceptions, Runtime Errors)

Evaluate
(expression) To execute part of a program in order to produce a value. For an expression, to execute the program code associated with the expression and return the resulting value. For the macro operator, to compile the macro string, execute the resulting program code, and return the resulting value. (*See also:* Expression)

Exception
(error handling) An occurrence of an abnormal condition during the execution of an operation. An exception is said to be raised when an operation fails. (*See also:* Error, Failure, Runtime Error)

Exclusive
(network) In a network, to assure that no other user will write data to a file, it may be opened in an Exclusive mode. Only the user opening the file exclusively may then access it until exclusive use of the file (by closing it, or opening it SHARED) is relinquished.

Executable File
(configuration) A file output from the linker directly executable from the operating system command line. Executable files have an .EXE extension. (*See also:* Linker)

Execution Bar
(debugger) The highlight bar which is positioned on the line of code to be executed next.

Exported Instance Variables
(object-oriented) *See* Instance Variables.

Expression
(expression) A combination of constants, identifiers, operators, and functions that yield a single value when evaluated.

Extend Routine
(extend) A routine written in a language other than CA-Clipper that can be called from CA-Clipper with or without return values.

Extend System API
(extend) A set of C and Assembly-callable routines that provide interface and services to extend routines.

Extended Memory
(general) Extended Memory is the IBM designation for physical memory above one megabyte that can be accessed by an 80286 or greater CPU in protected mode. (*See also:* Protected Mode)

Extension
(file) A portion of a file name normally used for identifying the type or originating program of a file. (*See also:* Drive, File Name, Path)

Failure
(error handling) The inability of an operation to satisfy its purpose. When a failure occurs an exception is raised. Failures are due in large part to errors. (*See also:* Exception, Error, Retry, Runtime Error)

Far Pointer
(memory addressing) A PC memory address consisting of both a 64K segment selector and an offset value which expands to a 20-bit address.

Field
(database) The basic column unit of a database file. A field has four attributes: name, type, length, and decimals if the type is numeric. (*See also:* Database, Record, Tuple, Vector, View)

Field Specification (Field Spec)
(Workbench) A *field specification* (or *field spec*) is essentially a template, that is, a set of properties (such as picture clauses, validation and formatting rules, etc.) that are related to a field but are *independent* of any particular data server. One of its advantages is that any changes made to a field spec are automatically propagated to all data servers that use that field spec. (*See also:* Data Server)

Field Variable
(variable, database) A variable that refers to data in a database field, as opposed to data in memory. (*See also:* Local Variable, Memory Variable, Private Variable, Public Variable, Static Variable, Variable)

FieldSpec Editor
(Workbench) Use the FieldSpec Editor to create and modify field specs *independent of* any field or data server with which they may be associated. (See also: Data Server, Field Specification, Visual Editor)

File
(file) A file is an organized collection of bytes stored on disk, maintained by the operating system, and referenced by name. Its internal structure is solely determined by its creator. (See also: Binary File, Database File, Text File)

File Handle
(file) An integer numeric value returned from FOPEN() or FCREATE() when a file is opened or created. This value is used to identify the file for other operations until it is closed.

File Locking
(network) The process by which a user is guaranteed exclusive access to a database file. The file is only available to the user that applied the lock. (*See also:* Record Locking)

File Name
(file) The name of a disk file that may optionally include a drive designator, path, and extension. (*See also:* Drive, Extension, Path)

File Server
(network) A computer on a network dedicated to providing data storage to other computers (i.e., workstations) for the purpose of sharing information among multiple users. File servers tend to provide other services such as E-Mail service and shared printer support as well.

File-wide Declaration
(variable) A variable declaration statement that has the scope of the entire source file. File-wide declarations are specified before the first procedure or function declaration in a program file, and the program file must be compiled with the /N option. (*See also:* Scope, Storage Class)

Foreground Color
(user-interface) The color of text appearing on the screen, usually on a different colored background. Though foreground color and background color are usually different, you may render text invisible by choosing the same color for both. (*See also:* Background Color)

Form Editor
(Workbench) Use the Form Editor to create forms, such as dialog boxes and data forms; define their properties and controls; and modify existing forms using standard editing techniques. (See also: Data Form, Visual Editor)

Form Feed
(general) A special character (CHR(12)) that by convention causes most printers to move the printhead to the top of the next page. (*See also:* Hard Carriage Return, Line Feed)

Full-Screen Input/Output

(user-interface) A style of operation of the keyboard and display used for complex data entry and display tasks. Full-screen input and output are generally performed using the @..SAY, @..GET and READ commands. Full-screen output is distinguished from console-style output by the fact that control characters (e.g., backspace, carriage-return) are not processed, and wrapping and scrolling do not occur at the boundaries of the visible display area. (*See also:* Console Input/Output)

Function

(procedure and function) An executable block of code with an assigned name. Alternately, the collection of source code statements that define a function. Certain functions are supplied as part of CA-Clipper; others are defined by the programmer using the FUNCTION or PROCEDURE declaration statements. The latter are referred to as *user-defined* functions.

The terms *procedure* and *function* are generally interchangeable. By convention, a function returns a value, while a procedure does not. (*See also:* Activation, Parameter, Procedure)

Group

(linker) An Intel 8086 addressing classification defining a collection of segments to be addressed using the same segment register. Note that a group is not a section but rather a logical concept used only for addressing.

GUI

(user-interface) Abbreviation for Graphical User Interface. Used generically by CA-Clipper to refer to graphical user interfaces like Windows.

Hard Carriage Return

(general) An explicit carriage-return character at the end of a line in a text file, as opposed to a soft carriage return that might be inserted into text by a program designed to handle word-wrapping. A hard carriage-return character is generated by the expression (CHR(13)) where a soft carriage-return character is generated with the expression (CHR(141)).

Header File

(configuration, preprocessor) A source file containing manifest constant definitions; command or pseudofunctions; and/or program statements merged into another source file using the #include preprocessor directive. (*See also:* Program File, Source File, Std.ch)

Help Window

(debugger) The window in which on-line help is displayed.

Hexadecimal

(general) A representation of a value in base-16 rather than decimal, which is base-10. Hexadecimal values are easily converted to and from binary (base-2), which is the form of data the computer actually uses. Hexadecimal values are represented by digits zero through nine and A through F for values between 10 and 15.

High-Level Wrapper Functions

(database) A CA-Clipper-callable function that consists of calls to lower-level internals. Specifically, in the RDD subsystem, a high-level wrapper function contains calls to a work area's RDD Virtual Jump Table.

Hidden
(modular programming) The resulting state of a program module written to conform to the principals of information hiding. (*See also:* Information Hiding)

Highlight
(user-interface) Indicates input focus for menus, browsers, or GETs. With menus and browsers, the currently selected item or cell has input focus and is displayed in the current enhanced color or inverse video. With GETs, the current GET is highlighted in the current enhanced color or inverse video while the other GETs are displayed in the current standard color if an unselected color setting is active. (*See also:* Cell, Standard Color, Enhanced Color, Input focus, Unselected Color)

IBM Extended Character Set
(general) The character set built into the ROM of the IBM-PC. This character set is a superset of ASCII, containing additional special characters (such as a line drawing character set) that may be used to enhance your program screens.

IDE
(user-interface) Abbreviation for Integrated Development Environment. Used generically by CA-Clipper to refer to Windows-based application development environments such as the CA-Clipper Workbench.

Identifier
(preprocessor, procedure and function, variable) A name that identifies a function, procedure, variable, constant or other named entity in a source program. In CA-Clipper, identifiers must begin with an alphabetic character and may contain alphabetic characters, numeric characters, and the underscore character.

Identity
(database) A unique value guaranteed by the structure of the data file to reference a specific record in a database even if the record is empty. In the Xbase file (.dbf), the identity is the record number; but it could be the value of a unique primary key or even the offset of an array in memory.

Include File
(preprocessor) *See* Header File.

Increment
(expression) To increase a value by a fixed amount, usually one. In CA-Clipper the increment operator (++) can be used to increment a numeric value in a variable.

Incremental Linking
(linker) The ability to link only the modules of an application that have been changed, greatly increasing the speed in which the link occurs. (*See also:* Linking, Module)

Index
(database) An ordered set of key values that provides a logical ordering of the records in an associated database file. Each key in an index is associated with a particular record in the database file. The records can be processed sequentially in key order, and any record can be located by performing a SEEK operation with the associated key value. (*See also:* Controlling/Master Index, Key Value, Natural Order)

Information Hiding
(procedure and function) A fundamental programming principle that states that functions and programs should conceal their inner workings from other functions and programs. Stated simply: a function should possess only the knowledge necessary for it to accomplish its task. When one function calls another, the calling function should possess only the knowledge explicitly required to call the other function. (*See also:* Encapsulation, Lexical Scoping, Modularity, Side Effect)

Initialize
(variable) To assign a starting value to a variable. If initialization is specified as part of a declaration or variable creation statement, the value to be assigned is called an *initializer*. (*See also:* Assignment)

Input Focus
(user-interface) The GET, browse cell, or menu item where user interaction can take place is said to have input focus. The item with input focus usually is displayed in enhanced color or inverse video.

Insert Mode
(user-interface) A data entry mode entered when the user presses the insert key. When this mode is active, characters are inserted at the cursor position. Text to the right of the cursor is shifted right. (*See also:* Overstrike Mode)

Inspecting
(debugger) The process of examining work areas, variables, expressions and activations inside the debugger.

Instance Variable
(object-oriented) Instance variables are the attributes or the data portion of an object as defined by the object's class. Each object, when created, is given its own unique set of instance variables initialized to their default values. Instance variables that are accessible are called *exported* instance variables. Exported instance variables can be inspected and—in some cases—assigned using the send operator. The instance variables of an object persist as long as the object it belongs to. (*See also:* Class, Object)

Instantiation
(object-oriented) Creation of an *instance* of a class, i.e., an object.

Integer
(data type) A number with no decimal digits. Note that CA-Clipper does not provide a separate data type for integer values.

Iteration
(algorithm) One of the three basic building blocks of algorithm development (the others are sequence and selection). Iteration refers to operations that are performed repeatedly, usually until some condition is satisfied. (*See also:* Selection, Sequence)

Join
(database) An operation that takes two tables as operands and produces one table as a result. It is, in fact, a combination of other operations including selection and projection. (*See also:* Selection, Projection)

Key Expression
(database) An expression, typically based on one or more database fields, that when evaluated, yields a key value for a database record. Key expressions are most often used to create indexes or for summarization operations. (*See also:* Index, Key Value)

Key Value
(database) The value produced by evaluating a key expression. When placed in an index, a key value identifies the logical position of the associated record in its database file. (*See also:* Index, Key Expression)

Keyboard Buffer
(user-interface) An area of memory dedicated to storing input from the keyboard while a program is unable to process the input. When the program is able to accept the input, the keyboard buffer is emptied.

Keyboard Polling
(user interface) The periodic system access of the keyboard buffer as one of the system services.

(system) That part of the system services that seeks and reports activity at the keyboard. It is one Input element of the DOS I/O channels.

Keyed Pair
(database) A pair consisting of a key value and an identity.

Keyword
(command, language) A word that has a special meaning to a compiler or other utility program. Commands, directives, or options are often recognized by examining supplied text to see if it contains keywords.

LAN
(network) An acronym for Local Area Network. Generally used to describe a system by which microcomputers are connected together to perform such functions as file and peripheral sharing, electronic mail, and centralized backup of data. (*See also:* Local Area Network)

Lexical Scoping
(variable) A method of determining an item's existence, visibility, or applicability (i.e., the item's *scope*) by it's position within the text of a program. (*See also:* Local, Variable, Scope of a Variable)

Lexically Scoped Variable
(variable) A variable that is only accessible in a particular section of a program, where that section is defined using simple textual rules. For example, a local variable is only accessible within the procedure that declares it. (*See also:* Dynamic Scoping, Private Variable, Public Variable, Static Variable)

Library
(linker) A file containing one or more object modules. Modules are extracted by linker and combined with object files to form an executable (.EXE) file.

Library File
(configuration) A file containing one or more object modules. The linker searches specified libraries to resolve references to functions or procedures that were not defined in the object files being linked. (*See also:* Linker, Module, Object File)

Lifetime of a Variable
(variable) The period of time during which a variable retains its assigned value. The lifetime of a variable depends on its storage class. (*See also:* Scope of a Variable, Visibility)

Line Feed
(general) A special character (CHR(10)) that by convention causes the cursor or printhead to move to the next line or to terminate a line in a text file. It is usually used in combination with a hard carriage return. (*See also:* Form Feed, Hard Carriage Return)

Linker
(program) A program that combines object files created by a compiler to produce an executable program. The linker examines the supplied object files to resolve symbol references between modules. If a module refers to a symbol that is not defined by any of the modules, the linker searches one or more libraries to resolve the reference. (*See also:* Library File, Object File)

Linking
(linker) The process in which object files and libraries are combined and references are resolved to produce a relocatable memory image (generally, an executable).

List
(expression, command) A list of expressions, field names, or file names, separated by commas specified generally as command, procedure, or function arguments. Code blocks can also execute a list of expressions.

List Match Marker
(preprocessor) A match marker indicating a position that will successfully match a list of one or more arbitrarily complex expressions. A list match marker marks a part of a command that is expected to consist of a list of programmer-supplied expressions. A list match marker has the form <*id*,...>. *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Literal
(data type) A source code element interpreted literally (as encountered) and assumed to have no abstract meaning. Generally a constant. (*See also:* Constant).

Literal Array
(array) In CA-Clipper, an array specified by enclosing a series of expressions in curly ({}) braces. A literal array is an expression that evaluates to an array reference. (*See also:* Array, Array Reference)

Local Area Network
(network) A system by which microcomputers are connected (via coaxial cable, optical fiber, twisted pair phone wire, or other media), relying on sophisticated operating software to perform such functions as file and peripheral sharing, electronic mail, and centralized backup of data. (*See also:* LAN).

Local Variable
(variable) A variable that exists and retains its value only as long as the procedure in which it is declared is active (i.e., until the procedure returns control to a higher level procedure). Local variables are lexically scoped; they are accessible by name only within the procedure where they are declared. (*See also:* Dynamic Scoping, Lexical Scoping, Private Variable, Public Variable, Static Variable)

Locklist
(database) A list of the records that are currently locked in the work area.

Logical Type
(data type) A special data type consisting of true (.T.) or false (.F.) values. (*See also:* Condition)

Logify
(preprocessor) To change an expression in the source text into a logical value. Logifying is accomplished by surrounding the expression with periods.

Logify Result Marker
(preprocessor) A result marker of the form #<.*id*.>. *id* must correspond to the name of a match marker. This result marker writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. The input text itself is not written to the result text.

Macro
(expression) In CA-Clipper, an operation that allows source code to be compiled and executed at runtime. In CA-Clipper, the macro symbol (&) does not perform *text substitution* unless embedded within a character string. Instead, it is generally treated as a unary operator that operates on a character string. The text in the character string is compiled *on the fly* using a special runtime compiler. The resulting code is then executed, and the value obtained is returned as the result of the macro operation. (*See also:* Code Blocks, Unary Operator)

Maintainable Scoped Orders
(database) Scoped (filtered) orders created using the FOR clause. The FOR condition is stored in the index header. Orders of this type are correctly updated using the expression to reflect record updates, deletions and additions.

Make
(program) A program used to maintain multi-file program systems. A make program takes as its input a file (make file) specifying the relationships between files. When executed, the make program compares the date and time stamps of specified target files to the specified dependent files. If any of the dependent files have a more recent date and time stamp than the associated target files, a series of actions is performed. (*See also:* Make File)

Make File
(configuration) A text file used as input to a make utility containing the specifications and actions required to build a program or a system of programs. This file is often referred to as a description file. (*See also:* Make)

Manifest Constant
(preprocessor) An identifier specified in a #define directive. The preprocessor substitutes the specified result text whenever it encounters the identifier in the source text.

Map File (.MAP)
(linker) The map file (.MAP) contains information about symbol and segment addresses within the memory image created by a linker. It is generated when requested through the use of the appropriate command line switch.

Mask
(ASCII) A set of characters that represent the valid data set for some matching or selection process. A character pattern used as a filter in evaluation of a character string. An ASCII or binary filter.

(binary) Data in a specific storage location that replaces characters in the accumulator. Also, a modifier in a logical operation.

Master Index
(database) *See* Controlling/Master Index.

Match
(preprocessor) A successful comparison of source text with a match pattern (or part of a match pattern).

Match Marker
(preprocessor) A construct used in a match pattern to indicate a position that will successfully match a particular type of source text. There are several types of match markers, each of which will successfully match a particular type of source text.

Match Pattern
(preprocessor) The part of a translation directive that specifies the format of source text to be affected by the directive. A match pattern generally consists of words and match markers.

Memo Type
(data type, database) A special database field type consisting of one or more characters in the IBM extended character set. The maximum size of a memo field in CA-Clipper is 65,534 bytes. A memo field differs only from a character string by the fact it is stored in a separate memo (.dbt) file and the field length is variable length. (*See also:* Character, String)

Memory Variable
(variable) In general, a variable that resides in memory, as opposed to a database field variable. Sometimes used specifically to refer to variables of the MEMVAR storage class (private and public variables), as opposed to static or local variables. (*See also:* Field Variable, Local Variable, Private Variable, Public Variable, Static Variable, Variable)

Menu
(user-interface) An on-screen list of choices from which the user selects. Menus range from simple to elaborate forms. Two examples are menus that *pull-down* from the top of the screen (an elaborate type requiring more programming), or a simple list of numbered items from which the user selects by entering the appropriate number. (*See also:* Menu Editor)

Menu Bar
(debugger) The bar at the top of the debugger screen, on which the available menu choices are displayed.

Menu Editor

(Workbench) Use the Menu Editor to create menu structures, define their properties and menu items, and modify existing menus using standard editing techniques. (*See also:* Auto Layout, Menu, Visual Editor)

Messages

(object-oriented) A message is the way an object is requested to perform some action. Messages are sent to an object and composed of the object name, the send operator, and the selector name followed by arguments enclosed in parentheses. The selector has the same name as the method it is calling. Sending a message produces a return value, much like a function call, with the return value varying depending on the operation performed. (*See also:* Object, Method, Instance Variable)

Metasymbol

(language) Descriptive symbols used in syntax to represent information that must be supplied as part of a source code statement. A metasymbol is constructed using two information components: a data type prefix and a logical descriptor.

Metasymbol table

Prefix	Type
a	Array
b	Code block
c	Character
d	Date
exp	Expression
id	Literal identifier
l	Logical
m	Memo
n	Numeric
obj, o	Object
u	Undetermined
x	Extended expression

Within a syntax statement, metasymbols are generally delimited by square or pointed brackets ([], <>); the delimiters are not part of the syntax.

Method

(object-oriented) A method is the operation performed in response to a message sent to an object. (*See also:* Class, Message, Object)

Modularity

(modular programming) Roughly, a measure of a system's adherence to the principles of modular programming. The principles of modular programming are not precisely defined, but may be said to comprise these basic ideas: programs should be organized as well-defined *modules*; modules should correspond with syntactic units of the programming language (such as functions or source files); a module should accomplish a well-defined task; a module should interact with as few other modules as possible; interactions between modules should be explicitly specified in the source code for the modules; modules should obey the principle of information hiding. (*See also:* Encapsulation, Information Hiding, Lexical Scoping, Side Effect)

Module

(modular programming) Generically, a procedure or function (or a set of related procedures and functions) that can be treated as a unit. Sometimes used to refer specifically to the code in a single object file, normally the result of compiling a single source file. (*See also:* Object File, Source File)

Module
(linker) A portion of the object code that is a discrete unit. If any part of a module is linked, the entire module must be linked.

(Workbench) In CA-Clipper, *applications* consist of *modules*, which consist of *entities*.

Modules are "containers" for entities and can be one of three types in CA-Clipper:

- The Binary Objects module which contains the binary definitions of all form, menu, data server, and field spec entities created using the Workbench's four editors
- Program modules (usually .prg files)
- Header modules (usually .ch files)

Module Buttons
(Workbench) The buttons in a Module Browser pictorially represent each module in the current application. They are arranged in alphabetical order and visually convey the following information:

- The name of the module
- The type
- The compilation status
- The debug mode status
- The number of entities it contains

(*See also:* Compilation Status, Entity, Module)

Modulus
(mathematics) The correct term for the % operator, as used in CA-Clipper, is *modulo*. It produces the remainder of a division operation (e.g., 9 modulo 47 ==> 2). The CA-Clipper function, MOD(), performs a modulo operation on two numbers.

Traditionally, *modulus* is actually the term for an entirely different mathematical value (i.e., the absolute value of a complex number, computed by adding the squares of each and taking the positive square root of the sum). The same word also has a specific and different meaning in Physics.

Monitor Window
(debugger) The window in which monitored variables are displayed.

Monitored variable
(debugger) A variable which is selected by the options on the Monitor Menu and displayed in the Monitor Window.

Multi-tasking
(general) Computer operations in which one CPU handles several tasks. System operations in which a single CPU handles programs in a manner that seems simultaneous to one or more users while it interleaves the programs, executing them in segments.

Multidimensional Array
(array) In CA-Clipper, an array whose elements consist entirely of references to other arrays (called *subarrays*). The elements of the subarrays may, in turn, contain references to other arrays. Arrays organized in this fashion are said to be *nested*. Each level of nesting may be viewed as a *dimension* of the main array, and the elements of the subarrays may be accessed by applying multiple subscripts to the main array. (*See also:* Array, Array Reference, Nested Array, Subscript)

Multiple-Order Bag
(database) An order bag that can contain any number of orders; a multiple-tag index. The .cdx and .mdx files are examples of multiple-order bags.

Name
(general) A routine's public symbol. Often the words "symbol," "public symbol," and "name" will be used interchangeably.

Naming Convention
(general) The way a compiler alters the name of a routine before placing it in an object file. According to the Microsoft Mixed-Language Programming Guide, this term "refers to the way that a compiler alters the name of a routine before placing it in an object file."

Natural Order
(database) For a database file, the order determined by the sequence in which records were originally entered into the file. Also called unindexed order. (*See also:* Index)

Nested Array
(array) In CA-Clipper, two arrays are said to be *nested* if one of them contains a reference to the other. When an array contains a reference to a second array, the second array is sometimes called a *subarray* of the first array. (*See also:* Array, Array Reference, Multidimensional Array, Subscript)

NIL
(data type) A special data type that has only one allowable value. The special value (NIL) is automatically assigned to all uninitialized variables except publics and is also passed as a substitute when arguments are omitted in a procedure or function call.

Non-Maintainable/Temporary Orders
(database) Orders created using the WHILE or NEXT clauses. These orders are useful because they can be created quickly. However, the conditions in these clauses are *not* stored in the index header. Therefore, orders of this type are *not* correctly updated to reflect record updates, deletions and additions. They are only for temporary use.

Nondedicated Server
(equipment) A computer that serves as both a workstation and a network server.

Normal Stringify Result Marker
(preprocessor) A result marker of the form <*"id"*>. *id* must correspond to the name of a match marker. A normal stringify result marker specifies that the corresponding source text is to be enclosed in quotes. If the matched source text is a list, each element of the list is individually stringified. If no source text was matched, an empty result is produced.

Normalization
(database) The process of elimination and consolidation of redundant data elements in a database system.

Numeric Type
(data type) A special data type consisting of values that indicate magnitude. Numeric values consist of digits between zero and nine, a sign, and a decimal point.

Object
(object-oriented, data type) An object is an instance of a class. Each object has one or more attributes (called instance variables) and a series of operations (methods) that execute when a message is sent to the object. The object's instance variables can only be accessed or assigned by sending messages to the object. Objects are created by calling a special function associated with a class. (*See also:* Class, Instance Variable, Message, Method)

Object File
(configuration) A file that contains the output of a compiler or other language translator, generally the result of compiling a single source file. Object files are linked to create an executable program. (*See also:* Linking, Source File)

Operand
(expression) A value that is operated on by an operator, or the term in an expression that specifies such a value. For example, in the expression $x + 5$, x and 5 are operands. (*See also:* Operator)

Operating System
The basic software program that organizes and services the computer and its peripheral devices. The operating system supported by CA-Clipper, MS/PC-DOS, is organized into several layers as follows:

- *Loader* is the layer which brings the operating system software into memory.
- *BIOS* is the basic hardware interface layer that provides services to the kernel and consists of initialization code and device drivers.
- *Kernel* is the application interface layer and provides services for process control, memory management, peripheral support, and a file system.
- *User interface shell (COMMAND.COM)* provides basic services to the user including an interactive mode, directory management, and a service for loading and executing application programs.
- *Support programs* provide extended operating services not resident in the user interface shell.

Operator
(expression) A symbol that identifies a basic operation. For example, the multiplication operator (*) denotes that two values are to be multiplied. Operators are categorized as either unary or binary, depending on whether they require one or two operands, respectively. (*See also:* Binary Operator, Operand, Unary Operator)

Optional Clause
(command, preprocessor) A portion of a match pattern that is enclosed in square ([ ]) brackets. An optional clause specifies part of a match pattern that need not be present for source text to match the pattern. An optional clause may contain any of the components legal within a match pattern, including other optional clauses. When a match pattern contains a series of optional clauses that are immediately adjacent to each other, the matching portions of the source text are not required to appear in the same order as the clauses in the match pattern. If an optional clause is matched by more than one part of the source text, the multiple matches may be handled using a repeating clause in the result pattern.

Order
(database) A named mechanism (index) that provides logical access to a database according to the keyed-pairs. This term encompasses both single indices and the tags in multiple-tag indices.

Orders are not, themselves, data files. They provide access to data that gives the appearance of an ordering of the data in a specific way. This ordering is defined by the relationships between keyed-pairs. An order does not change the physical (the natural or entry) order of data in a database.

Order Bag
(database) A container that holds zero or more orders. Normally a disk or memory file. A traditional index like .ntx is an order bag that holds only one order. A multiple-tag index (.mdx or .cdx) is an order bag that holds zero or more orders. Though order bags may be a memory or disk file, CA-Clipper 5.3 only supports order bags as disk files.

Order List
(database) A list of all the orders available to the database in the specified work area.

Ordinal Position
(general) The numeric position within a sequence.

(programming) The position of an array or list element within its array or list. The number(s) that represent that position.

Overlay
(linker) A section of an executable program that shares memory with other sections of the same program. An overlay is read into memory when the code residing in it is requested by the root (non-overlaid) section or another overlay. (*See also:* Dynamic overlay)

Overstrike Mode
(user-interface) A data entry mode entered when the user presses the insert key. When this mode is active, characters are entered at the cursor position and text to the right of the cursor remains stationary. (*See also:* Insert Mode)

Page Frame
(memory) A 64 KB block of memory that is made available to applications as four contiguous 16 KB pages.

Pan
(user interface) Apparent horizontal movement of a cursor or view across an area. Sometimes mistakenly applied to apparent vertical movement which is, correctly, scrolling. (*See also:* Scrolling)

Parameter
(variable) A identifier that receives a value or reference passed to a procedure or user-defined function. A parameter is sometimes referred to as a *formal parameter*. (*See also:* Activation, Argument, Function, Procedure, Reference)

Parameter-passing Convention
(general) The specific format in which particular parameters are meant to be transferred.

Path
(file) A literal string that specifies the location of a disk directory in the tree structured directory system. A path specification consists of the following elements: an optional disk drive letter followed by a colon, an optional backslash indicating that the path starts at the root directory of the specified drive, the names of all the directories from the root directory to the target directory, separated by backslash (\) characters. Example: C:\CLIP53\INCLUDE. A path list is a series of path specifications separated by semicolons.

Picture
(user-interface) A string that defines the format for data entry or display in a GET, SAY, or the return value of TRANSFORM(). Picture strings are comprised of functions which affect the formatting as a whole and a series of template characters that affect formatting on a character by character basis. (*See also:* Template)

Pointer
(memory) A variable that *points* to another variable. Usually a pointer is a memory address. As one of C's strongest features, pointers provide the means by which functions can modify their calling arguments. They are also useful for optimizing routines.

Port
(general) A designation for the hardware that allows the processor to communicate with peripheral devices.

Postfix Notation
(general) Inline placement of decrement (--) or increment (++) operators after the variable so the operators are applied to the value after an assignment.

Example:

```
LOCAL x, y
x := 99
Y = x++    // postfix increment
? Y        // Y = 99
```

Precedence
(expression) The stature of an operator in the hierarchy that determines the order in which expressions are evaluated. For example, the expression *5 + 2 * 3* is interpreted as *5 + (2 * 3)* because the multiply operator (*) has a higher precedence than the addition operator (+). (*See also:* Expression)

Prefix Notation
(general, operators) Inline placement of decrement (--) or increment (++) operators before the variable so the operators are applied to the value before an assignment.

Example:

```
LOCAL x, y
x := 99
Y = ++x    // prefix increment
? Y        // Y = 100
```

Preprocessor
(program, preprocessor) A translation program that prepares source code for compilation by applying selective text replacements. The replacements to be made are specified by directives in the source file. In CA-Clipper, the preprocessor operates transparently as a part of the compiler program (CLIPPER.EXE). (*See also:* Compiler)

Preview Menu Bar
(Workbench) As you define a menu structure in the Menu Editor, each new entry is added to the *preview menu bar*, a prototypical menu bar at the top of the Menu Editor window. The preview menu bar is partially operational—showing description messages in the status bar and allowing submenus to be pulled down—but nothing actually happens when you make a selection. Its purpose is to give you visual feedback while you are designing a menu structure. (*See also:* Menu Editor)

Primitive
(procedure and function) A simple, low-level function used by other high-level functions or programs to perform a more complex task.

Print Spooler
A program running either on a local workstation or on the file server that captures print jobs to a file and then queues them for later printing. Print spoolers generally operate as background tasks in order to facilitate printing while other tasks are operating in the foreground.

Private Variable
(variable) A variable of the MEMVAR storage class. Private variables are created dynamically at runtime using the PRIVATE statement. They are accessible within the creating procedure and any lower level procedures unless obscured by another private variable with the same name. (*See also:* Activation, Dynamic Scoping, Function, Lexical Scoping, Local Variable, Procedure, Public Variable, Static Variable)

Procedure
(procedure and function) An executable block of code with an assigned name. Alternately, the collection of source code statements that define a procedure. In CA-Clipper, this can be a source (.prg) file, a format (.fmt) file, an explicitly declared procedure (PROCEDURE), or an explicitly declared function (FUNCTION).

The terms *procedure* and *function* are generally interchangeable. By convention, a function returns a value, while a procedure does not. (*See also:* Activation, Function, Parameter)

Procedure File
(configuration) An ASCII text file containing CA-Clipper procedure and function definitions, usually ending with a .prg extension; a program file. (*See* Program File)

Program Editor
(program) An executable program that allows the creation and editing of text files or programs from within DOS. (*See also:* File Server, Source Code Editor)

Program File
(configuration) An ASCII text file containing CA-Clipper source code. Program files usually end with a .prg extension. The compiler reads the program file, translates the source code, and produces an object file that is then linked to produce an executable program. (*See also:* Linking, Object File, Source Code)

Projection
(database) A DBMS term specifying a subset of fields. In CA-Clipper, the analogy is the FIELDS clause. (*See also:* Join, Selection)

Prompt
(user-interface) A series of characters displayed on the screen indicating that input from the keyboard is expected.

Protected Mode
(CPU) In this mode, an 80286 or higher processor retains instruction compatibility with the 8086 but can directly address 16 MB of memory.

Pseudofunction
(preprocessor) A function-like construct that is replaced with another expression via the #define directive, rather than compiled into a conventional function call. Pseudofunctions may contain parenthesized arguments that may be included in the substituted text.

Public Variable
(variable) A variable of the MEMVAR storage class. Public variables are created dynamically at runtime using the PUBLIC statement and are accessible from any procedure at any level unless obscured by a private variable with the same name. (*See also:* Activation, Dynamic Scoping, Function, Lexical Scoping, Local Variable, Private Variable, Procedure, Static Variable)

Query
(database) A request for information to be retrieved from a database. Alternately, a data structure in which such a request is encoded.

(general) A query is also a general term used when you want to interrogate a setting or an exported instance variable for its current value.

Queue
(general) A data structure of variable length where elements are added to one end and retrieved from the other. A queue is often described as *first in, first out*. (*See also:* Stack, Print Spooler)

Real Memory
(memory) Memory available to an application at time of execution.

Real Mode
(CPU) 8086 emulation mode. In this mode, a computer cannot directly address *extended* memory, and an 80286 or higher processor performs as if it were a fast 8088.

Record
(database) A record in the traditional database paradigm is a row of one or more related columns (fields) of data. In the expanded architecture of CA-Clipper, a record could be data that does not exactly fit this definition.

A record is, in this expanded context, data associated with a single *identity*. In an Xbase data structure, this corresponds to a row (fields associated with a record number); in other data structures, this may not be the case.

In this document we use "record" in the traditional sense, but you should be aware that CA-Clipper permits expansion of the meaning of record.

Record
(database) The basic row unit of a database file consisting of one or more field elements. (*See also:* Database, Field, Table, Tuple)

Record Locking
(network) The process by which one user obtains exclusive access to a record in a database to prevent another user from attempting to write data to it concurrently. A record lock must be applied prior to writing to a database in use by more than one user.

Recovery
(error handling) The process of attempting to handle an exception or runtime error. Generally, recovery consists of three possible actions: terminate processing, retry the failed operation, or resume processing with the next operation. In all cases, the environment of the program must be restored to a stable state. (*See also:* Exception, Error, Failure, Retry, Runtime Error)

Recursion
(expression) The calling of a procedure by a statement in that same procedure. When a procedure calls itself it is said to *recurse*. A recursive call causes a new activation of the procedure. If the source code for the procedure includes a declaration of local variables, a new set of local variables is created for each activation. A private variable created by the procedure is associated with the activation in which it is created and is visible in that activation and any lower level activations, unless obscured by a private variable created in a lower level activation. (*See also:* Activation, Function, Private Variable, Procedure)

Reference
(array, variable) A special value that refers indirectly to a variable or array. If one variable contains a reference to a second variable (achieved by passing the second variable by reference in a function or procedure call), operations on the first variable (including assignment) are *passed through* to the second variable. If a variable contains a reference to an array, the elements of the array can be accessed by applying a subscript to the variable. (*See also:* Array Reference, Parameter)

Regular Match Marker
(preprocessor) A match marker indicating a position that will successfully match an arbitrarily complex expression in the source text. A regular match marker generally marks a part of a command that is expected to consist of arbitrary programmer-supplied text, as opposed to a keyword or other restrictive component. In order for the source text to match, it must constitute a properly formed expression. A regular match marker has the form *<id>*. *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Regular Result Marker
(preprocessor) A result marker of the form *<id>*. *id* must correspond to the name of a match marker. This result marker writes the matched input text to the result text or writes nothing if no input text is matched.

Relation
(database) A link between database files that allows the record pointer to move in more than one database file based on the value of a common field or expression. This allows information to be accessed from more than one database file at a time.

Relational Database System
(database) A system that stores data in rows and columns, without system dependencies within the data. In other words, relationships between different databases are not stored in the actual database itself, as is the case in a system that uses record pointers.

Relative Addressing
(user-interface) To refer to a memory address, array element, screen location, or printer location with respect to another value, rather than referring to a specific address or element.

Repeating Clause
(preprocessor) A portion of a result pattern surrounded by square ([ ]) brackets. The text specified by the repeating clause is written to output once for each successfully matched match marker in the corresponding match pattern.

Repository
(Workbench) The *repository* is where the CA-Clipper Workbench stores all application components, and it automatically manages the relationships between the various components of an application.

Restricted Match Marker
(preprocessor) A match marker indicating a position that will successfully match one or more specified keywords. A restricted match marker marks a part of a command that is expected to be a keyword. A restricted match marker has the form *<id: wordList>* where *wordList* is a list of one or more keywords. Source text is successfully matched only if it matches one of the keywords (or is an acceptable abbreviation). *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Result Pattern
(preprocessor) The part of a translation directive that specifies the text to be substituted for source text that matches the match pattern. A result pattern generally consists of operators and result markers.

Result Text
(preprocessor) The text that results from formatting matched source text using a result pattern. If the result text matches a match pattern in another preprocessor directive, then it becomes the source text for that directive. Otherwise, the result text is passed as input to the compiler.

Retry
(network) Upon failing to lock a record in a database that is opened in shared mode, retry refers to attempting the write again based upon certain programmatic parameters. (*See also:* Shared, Exclusive)

(error handling) After an exception has been raised and the conditions of a failure corrected, an attempt is made to re-execute the failed operation. (*See also:* Exception, Error, Failure, Runtime Error)

Return Value
(procedure and function) The value or reference returned by a function or method from a function call or message send.

Robust
(general) Strength, durability. The quality of an application or language, or element that permits its effective use, without failure, in many environments and situations.

Routine
(general) Any procedure, function, or other complete lexical unit.

Row
(database) A group of related column or field values that are treated as a single entity. It is the same as a CA-Clipper record. (*See also:* Column, Field, Record)

(user-interface) A numeric expression that evaluates to an integer identifying a screen or printer row position.

Run Mode
(debugger) The mode of execution in which an application executes without pausing, until a breakpoint or tracepoint is reached.

Runtime Error
(error handling) An error that halts a program while it is executing.

Scope
(command, database) In a database command, a clause that specifies a range of database records to be addressed by the command. The scope clause uses the qualifiers ALL, NEXT, RECORD, and REST to define the record scope. (*See also:* Condition)

Scoreboard
(user-interface) An area of the display on line 0 beginning at column 60 that displays status information during certain data entry operations.

Screen Coordinates
(user-interface) The area of the screen beginning at the origin of the screen (0,0) and incrementing as X- and Y-coordinates. Note that the relation of coordinates to pixels depends on the mapping mode that Windows uses.

Script File
(configuration) A text file that contains command input to a compiler, linker, or other utility program. A script file is often used in lieu of equivalent keyboard input. For the CA-Clipper compiler, script files contain a list of source files to be compiled into a single object file.

(debugger) A file in which frequently used debugger commands are stored and from which those commands can be executed.

Scrolling
(user-interface) The action that takes place when the user attempts to move the cursor or highlight beyond the window boundary to access information not currently displayed. (*See also:* Window)

SDF File
(file, database) A text file that contains fixed-length database records with each record separated by a carriage return/line feed pair (CHR(13) + CHR(10)) and terminated with an end of file mark (CHR(26)). Each field within an SDF file is fixed-length with character strings padded with trailing spaces and numeric values padded with leading spaces. There are no field separators. (*See also:* Database, Text File, Delimited File)

Search Condition
(database) *See* Condition, Scope

Section
(linker) Load module portion of an .EXE or .OVL file loaded into memory as a single unit. In a program with overlays, the root section containing the main program module loads when the program is executed. Other sections are loaded as overlays when modules within them are invoked. (*See also:* Dynamic Overlay, Linking)

Segment
(linker) Code or data handled by the linker as a indivisible unit.

Segment Handle (VM API)
(linker) The identifier of a particular Virtual Memory segment.

Selection
- (algorithm) One of the three basic building blocks of algorithm development (the others are sequence and iteration). Selection allows control to flow along a number of possible paths, depending on the circumstance encountered.
- (database) A DBMS term that specifies a subset of records meeting a condition. The selection itself is obtained with a selection operator. In CA-Clipper, the analogy is the FOR clause.

(*See also:* Sequence, Iteration, Join, Projection)

Self
(object-oriented) An object-oriented term describing a reference to the object that received the current message. In CA-Clipper, this reference is often the return value of a message send.

Send Operator
(object-oriented) A new operator (:) in CA-Clipper used to send messages to user interface objects.

Separator
(file, database) The character or set of characters that differentiate fields or records from one another. In CA-Clipper, the DELIMITED and SDF file types have separators. The DELIMITED file uses a comma as the field separator and a carriage return/line feed pair as the record separator. The SDF file type has no field separator, but also uses a carriage return/line feed pair as the record separator. (*See also:* Delimiter)

Sequence
- (algorithm) One of the three basic building blocks of algorithm development (the others are selection and iteration). A sequence is a series of discrete steps that must be performed in a particular order.
- (language) In CA-Clipper, a series of statements enclosed in a BEGIN SEQUENCE control structure. (*See also:* Algorithm, Iteration, Selection)

Set Colors Window
(debugger) The window in which the Debugger color settings can be displayed and modified.

Shared
(network) A mode in which a file is opened that allows it to be accessed by more than one user at the same time. The inverse of Exclusive.

Shortcutting
(expression) A compiler optimization that causes expressions to be evaluated only to the extent required to determine their outcome. For example, in the expression *f*() .OR. *g*() function *g* need not be executed if function *f* returns true (.T.). CA-Clipper performs shortcutting on all logical operators (.OR. .AND. .NOT.).

Sibling
(Workbench) A *sibling* is a menu item at the same level as another within a menu structure. Use the Promote Item toolbar button to promote a menu item to the sibling level in a menu's hierarchy. (*See also:* Child)

Side Effect
(modular programming) An *unexpected effect* of executing a function or program. When a function changes the state of a system in a way that is not explicitly specified by the function's name or calling protocol, the change is called a *side effect*. Reliance on side effects is contrary to the principles of modular programming. (*See also:* Encapsulation, Information Hiding, Lexical Scoping, Modularity)

Single Step Mode
(debugger) The mode of execution in which only the line of code highlighted by the execution bar is executed, and its output displayed.

Single-dimensional Array
(array) In CA-Clipper, an array whose elements do not contain references to other arrays. (*See also:* Array, Array Reference, Multi-dimensional Array, Nested Array, Subarray, Subscript)

Single-Order Bag
(database) An order bag that can contain only one order. The .ntx and .ndx files are examples of single-order bags.

Skeleton
(command) A wildcard mask used to specify a group of file names or memory variables. The * is used to specify one or more characters and the ? to specify a single character.

Smart Stringify Result Marker
(preprocessor) A result marker of the form <*(id)*>. *id* must correspond to the name of a match marker. A smart stringify result marker specifies that the corresponding source text is to be enclosed in quotes unless the source text was enclosed in parentheses. If the matched source text is a list, each element of the list is individually processed. If no source text was matched, an empty result is produced. The smart stringify result marker is used to implement commands that allow extended expressions (a part of a command that may be either an unquoted literal or a character expression).

Soft Carriage Return
(general) A carriage return that is introduced into text usually in order to implement some sort of wrap operation, as opposed to a Hard Carriage Return that was specifically entered into the text when it was created.

Sort Order
(array, database) Describes the various ways database files and arrays are ordered.

- Ascending

  Causes the order of data in a sort to be from lowest value to highest value.

- Descending

  Causes the order of data in a sort to be from highest value to lowest value.

- Chronological

  Causes data in a sort to be ordered based on a date value, from earliest to most recent.

- ASCII

  Causes data in a sort to be ordered according to the ASCII Code values of the data to be sorted.

- Dictionary

  The data in a sort is ordered in the way it would appear if the items sorted were entries in a dictionary of the English language.

- Collating Sequence

  Data in a sort will be placed in sequence following the order of characters in the IBM Extended Character Set.

- Natural

  The order in which data was entered into the database.

Source Code
(procedure and function) The textual representation of a program or procedure. (*See also:* Source File, Object File)

Source Code Editor
(Workbench) Use the Source Code Editor to create new entities; edit existing entities using standard editing techniques—such as cut, copy, paste, delete, search for and replace text, and insert and delete lines of code. (*See also:* Program Editor)

Source File
(configuration) *See* Program File, Header File.

Source Text
(preprocessor) Text from a source file, processed by the preprocessor. Source text is examined to see if it matches a previously specified match pattern. If so, the corresponding result pattern is substituted for the matching source text.

Specialization
(Error class) The conditional processing performed by an error handler, usually in a series of case statements.

Spooler
(network) *See* Print Spooler.

Stabilization
(class) Stabilization is a TBrowse class process accomplished through the stabilize() method. The process includes a call to a browsing method and a testing of the instance variable.

After all the browse objects in a browsing area are established, the area is drawn on the screen, the internal browsing cursor is initialized, and the data is displayed in the proper locations. The process is incremental and may be interrupted at any time. It continues until all objects in a browse area are *stabilized.*

Stack
(general) A data structure of variable length whose elements are added and retrieved from the same end. A stack is often described as *first in, last out.* (*See also:* Queue)

Standard Color
(user-interface) The color pair definition that is used by all output options (such as SAY and ?), with the exception of GETs and PROMPTs, that use the enhanced color pair. (*See also:* Enhanced Color).

Statement
(language) In CA-Clipper, the basic unit of source code. A statement is normally a single line of text. Multiple statements can be placed on the same line by separating them with semicolons. A statement may be continued to another line by placing a semicolon at the end of the line to be continued. If the text of a statement matches a command definition (defined with a preprocessor directive), it is translated into the form specified by the command definition. (*See also:* Command)

Static Overlay
(linker) A section of the program that is not always resident in RAM and shares memory with other sections. The section that is currently in use is loaded into memory, allowing a larger program to execute in less available RAM.

Static Variable
(variable) A variable that exists and retains its value for the duration of execution. Static variables are lexically scoped; they are only accessible within the procedure that declares them, unless they are declared as *file-wide*, in which case they are accessible to any procedure in the source file that contains the declaration. (*See also:* Dynamic Scoping, Lexical Scoping, Local Variable, Private Variable, Public Variable)

Status Bar
(Workbench) Almost every window, browser, and editor in the CA-Clipper Workbench contains a status bar that displays helpful, informative text about the current window, a toolbar button, or selected menu command.

Std.ch
(preprocessor) The *standard header file* containing definitions for all CA-Clipper commands.

Storage Class
(variable) Defines the two characteristics of variables: lifetime and visibility. (*See also:* Lifetime, Scope, Visibility)

String
(data type) Generically, a value of type character. In source code, a series of characters enclosed in single or double quotes. (*See also:* Character)

Stringify
(preprocessor) To change source text into a literal character string by surrounding the text with quotes.

Stub
(error handling) A procedure used for debugging purposes that only simulates the intended actions of the real procedure. It may display an indicating message, return a constant value, or do nothing.

Subarray
(array) In CA-Clipper, an array that is referred to by an element of another array. (*See also:* Array, Array Reference, Multi-dimensional Array, Nested Array, Subscript)

Subclass (noun)
(class, API programming) A class that inherits from another class.

Subclass (verb)
(class, API programming) The act of extending or modifying the behavior of a class through inheritance.

Subdirectory
(file) *See* Directory.

Subscript
(array) A numeric value used to designate a particular element of an array. Applying a subscript to an array is called *subscripting* the array. In CA-Clipper programs, subscripting is specified by enclosing a numeric expression in square ([ ]) brackets after the name of a program variable. The variable is then said to be *subscripted*. (*See also:* Array, Array Reference, Multi-dimensional Array, Nested Array, Subarray)

Substring
(data type) A string within a string, usually to be specified as an argument of a function or command.

Swap Space (VM API)
(linker) Region of real memory in which the Virtual Memory Manager loads virtual memory segments.

Swapfile
(linker) Also known as the *workfile*, used by a linker to swap data and code in and out of memory during the linking process.

Symbol
(linker) An assigned name for a value representing a constant or the address of code or data. There are four types of symbols used by the linker defined as follows:

- *Absolute symbol:* a constant
- *Relative symbol:* address of code or data
- *Public symbol:* accessed by modules other than the module in which they are defined. Public symbols are used to share procedures and variables between modules. As such, the relative address of a public symbol is assigned by the compiler during compilation.
- *External symbol:* a public symbol not defined in the current module. Generally, these are references into CLIPPER.LIB or EXTEND.LIB, but the compiler generates them whenever there is a procedure or user-defined function referenced but not compiled into the current module.

Syntax
(language) The rules that dictate the form of statements or commands as defined by the implementors of the language. Also, a complete description of the forms that a statement or command can take.

Table
(database) A DBMS term defining a collection of column definitions and row values. In CA-Clipper, it is represented and referred to as a database file.

Tag
(database) A set of keyed-pairs that provides ordered access to the table based on a key value. Usually, an order in a multiple-order index (order).

Template
(user-interface) A mask that specifies the format in which data should be displayed. For example, you might want to store phone numbers as "9999999999" to save space, but use a template to display the number to the user as "(999) 999-9999."

Text File
(file) A file consisting entirely of ASCII characters. Each line is separated by a carriage return/line feed pair (CHR(13) + CHR(10)) and the file is terminated with a end of file mark (CHR(26)). (*See also:* Delimited File, Program File, SDF File)

Text Replacement
(preprocessor) The process of removing portions of input text and substituting different text in its place.

Toggle
(command) As a verb, to choose between an *on* or *off* state. As a noun, a value or setting that can be either on or off. A toggle is often represented using a logical value, with true (.T.) representing on, and false (.F.) representing off.

Token
(preprocessor) An elemental sequence of characters having a collective meaning. The preprocessor groups characters into tokens as it reads the input text stream. Tokens include identifiers, keywords, constants, and operators. White space, and certain special characters, serve to mark the end of a token to the preprocessor.

Tool Palette
(Workbench) The tool palette is a feature of the Form Editor that contains a set of icons that provide for drag-and-drop placement of GUI controls on a form.

Toolbar
(Workbench) Almost every window, browser, and editor in the CA-Clipper Workbench contains a customized toolbar that provides buttons as shortcuts to commonly used menu commands. Most toolbars have the same set of common buttons on the left, and buttons specific to the particular editor or browser on the right.

Trace Mode
(debugger) A mode of execution similar to Single Step Mode, the difference being that Trace Mode traces over function and procedure calls.

Tracepoint
(debugger) A variable or expression whose value is displayed in the Watch Window, and which causes an application to pause whenever that value changes.

Translation Directive
(preprocessor) A preprocessor instruction containing a translation rule. The two translation directives are #command and #translate.

Translation Rule
(preprocessor) The portion of a translation directive containing a match pattern followed by the special symbol (=>) followed by a result pattern.

Truncate
(expression) To remove insignificant information from the end of an item of data. With numerics, to ignore any part of the number that falls outside of the specified precision.

Tuple
(database) A formal DBMS term that refers to a row in a table or a record in a database file. In DIF files, tuple also refers to the equivalent of a CA-Clipper record. (*See also:* Database, Field, Record)

Two-dimensional Array
(array) An array that has two dimensions. In CA-Clipper, an array whose elements contain references to other arrays, all of which have the same length and do not refer to other arrays. (*See also:* Array, Array Reference, Nested Array, Subscript)

Typeahead Buffer
(user-interface) *See* Keyboard Buffer.

Unary Operator
(expression) An operator that operates on a single operand. For example, the .NOT. operator. (*See also:* Binary Operator, Operator)

Undefine
(preprocessor) To remove an identifier from the preprocessor's list of defined identifiers via the #undefine directive.

Undefined Symbol
(linker) An *unresolved symbol* that was never declared public by a module, but which is referenced by another module. After the public symbol definition is encountered, the symbol becomes defined (resolved). When a symbol is referenced, but not defined, it is said to be undefined.

Unselected Color
(user-interface) The color pair definition used to display all but the current GET, or the GET that has input focus. If this color setting is specified, the current GET is displayed using the current enhanced color. (*See also:* Enhanced Color).

Update
(database) The process of changing the value of fields in one or more records. Database fields are updated by various commands and the assignment operator.

User Function
(user-interface) A user-defined function called by ACHOICE(), DBEDIT(), or MEMOEDIT() to handle key exceptions. A user function is supplied to one of these functions by passing a parameter consisting of a string containing the function's name.

User Interface
(user-interface) The way a program interacts with its user (i.e., menu operation and selection, data input methods, etc.)

User-defined Function
(procedure and function) *See* Function.

Variable
(variable) An area of memory that contains a stored value. Also, the source code identifier that names a variable. (*See also:* Local Variable, Private Variable, Public Variable, Static Variable)

Vector
(database) In a DIF file, vector refers to the equivalent of a CA-Clipper field. (*See also:* Database, Field, Record, Tuple)

Verb
(command) The first word of a command that describes the action to perform. (*See also:* Command)

View
(database) A DBMS term that defines a virtual table. A virtual table does not actually exist but is derived from existing tables and maintained as a definition. The definition in turn is maintained in a separate file or as an entry in a system dictionary file. In CA-Clipper, views are supported only by DBU.EXE and are maintained in .vew files. (*See also:* Database, Field, Record)

View Sets Window
(debugger) The window in which CA-Clipper status settings can be inspected.

View Workareas Window
(debugger) The window in which work area information is displayed.

Virtual Memory
(general) Disk space, RAM drive, or expanded memory used as random access memory.

(VM API) The Virtual Memory Application Programmer Interface is a group of functions that permits a limited amount of real memory to emulate a much larger *virtual* memory space. Extend routines call these functions, directly communicating with the Virtual Memory Manager.

Visibility
(variable) The set of conditions under which a variable is accessible by name. A variable's visibility depends on its storage class. (*See also:* Dynamic Scoping, Lexical Scoping)

Visual Editor
(Workbench) A *visual editor* is a WYSIWYG environment that provides for visual development of applications using standard Windows techniques such as point-and-click and drag-and-drop placement of controls on a window.

The CA-Clipper Workbench's Menu Editor, Form Editor, DB Server Editor, and FieldSpec Editor are visual editors. (*See also:* DB Server Editor, FieldSpec Editor, Form Editor, Menu Editor)

Volume
(file) A unit of disk storage uniquely identified by a *label* and of fixed size. A hard disk can be partitioned into one or more volumes by an operation system utility. Volumes are subdivided into one or more directories organized in tree structure. (*See also:* Directory, Disk)

Wait State
(user-interface) A wait state is any mode that extracts keys from the keyboard except for INKEY(). These modes include ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT.

Watch Window
(debugger) The window in which watchpoints and tracepoints are displayed.

Watchpoint
(debugger) A variable or expression whose value is displayed in the Watch Window and updated as an application executes.

Wild Match Marker
(preprocessor) A match marker indicating a position that will successfully match any source text. A wild match marker matches all source text from the current position to the end of the source line. A wild match marker has the form <**id**>. *id* associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Window
(user-interface) A rectangular screen region used for display. A window may be the same size or smaller than the physical screen. Attempting to display information that extends beyond the specified boundaries of the window clips the output at the window edge.

Word
(preprocessor) A series of characters in a match pattern or result pattern. Source text matches a word in a match pattern if the text is identical to the word or is an acceptable abbreviation of it. A word that appears in a result pattern is copied unmodified into the result text.

Word Wrapping
(user-interface) The process of continuing the current text on the next line of a display when a boundary is reached and breaking the text on a word boundary.

Work Area
(database) The basic containment area of a database file and its associated indexes. Work areas can be referred to by alias name, number, or a letter designator. (*See also:* Alias)

Workfile
(linker) *See* Swapfile.

Workstation
(network) A personal computer connected to a network used to run applications and front end processes. (*See also:* File Server)

Index

!

#

$

%

&

(

*

+

-

.

/

:

<

=

>

?

@

[

^

{

|

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z