CA-Clipper®

For DOS

Version 5.3

Quick Reference Guide

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

Chapter 6: Classes

Chapter 7: Command Line Utilities

Chapter 8: Menu Utilities

Chapter 9: The CA-Clipper Debugger

Chapter 10: Environment Variables

Chapter 11: Operators

Chapter 12: Comment Indicators

Chapter 1
Introduction

The brief definitions in this guide include the following syntactical conventions:

- ***<idMemvar list>*** is a list of private or public variables or arrays.
- ***<idVar>*** is a variable name.
- ***<lCondition>*** is an expression whose evaluation to a Boolean value determines the enabling or disabling of a process.

For clarity, the syntactical statements omit semicolon line-continuation markers. All items are listed as they might appear if on a single line.

Except in the case of array descriptors, square brackets ([]) indicate optional clauses within commands and are not part of the syntax.

The asterisk preceding an item name indicates that it is obsolete or that it exists for compatibility with previous releases of CA-Clipper. Obsolete items, in general, are inconsistent with the current CA-Clipper programming philosophy. We strongly discourage their use since they may not be supported in future releases of CA-Clipper.

Note: You may find more detailed information in the "Language Reference" chapter of the *Reference Guide, Volumes 1 and 2* under the descriptions of each language item.

Chapter 2
Statements

Below is a summary of the CA-Clipper statements covered in the "Language Reference" chapter of the *Reference Guide, Volumes 1 and 2.*

Statement Reference

ANNOUNCE *<idModule>*
Defines a module identifier which may later be used to satisfy pending module REQUESTs.

BEGIN SEQUENCE
    <statements>...
[BREAK [<*exp*>]]
    <statements>...
[RECOVER [USING *<idVar>***]]**
    <statements>...
END [SEQUENCE]
Defines a control structure for exception and runtime error handling.

DECLARE** *<identifier>* **[[:=** *<initializer>], ...]**
Creates one or more private arrays or variables and optionally initializes variables to the specified values at runtime. Private variables are visible within the creation as well as invoked procedures or user-defined functions. Private variables exist until either explicitly released or the creating procedure or user-defined function terminates.

DO** *<idProcedure>* **[WITH** *<argument list>]**
Executes the specified procedure with an optional list of actual arguments. Arguments are passed by reference as a default.

DO CASE
CASE *<lCondition1>*
    <statements>...
[CASE *<lCondition2>***]**
    <statements>...
[OTHERWISE]
    <statements>...
END[CASE]
Selects a path of program execution from a set of conditions and branches after the first true evaluation.

[DO] WHILE *<lCondition>*
    <statements>...
    [EXIT]
    <statements>...
    [LOOP]
    <statements>...
END[DO]
Executes a looping structure while a condition is true (.T.).

EXIT PROCEDURE *<idProcedure>*
    [FIELD *<idField list>* **[IN** *<idAlias>***]]**
    [LOCAL *<identifier>* **[[:=** *<initializer>***]]]**
    [MEMVAR *<identifer list>***]**
.
. *<executable statements>*
.
    [RETURN]
Declares a procedure that will be executed upon program termination. EXIT procedures are called after the last executable statement in a CA-Clipper application has completed.

***EXTERNAL** *<idProcedure list>*
Declares one or more symbols to the linker, allowing procedures and user-defined functions to be referenced using macro variables.

FIELD *<idField list>* **[IN** *<idAlias>***]**
Declares one or more identifiers as fields by adding the field alias (FIELD->) to undeclared and unaliased variable names in *<idField list>* during compilation.

FOR *<idCounter>* **:=** *<nStart>* **TO** *<nEnd>*
    [STEP *<nIncrement>***]**
    <statements>...
    [EXIT]
    <statements>...
    [LOOP]
NEXT
Executes a looping structure for a specified number of times.

[STATIC] FUNCTION *<idFunction>*
    [(<*idParam list*>**)]**
    [LOCAL *<identifier>* **[[:=** *<initializer>***]**, ... **]]**
    [STATIC *<identifier>* **[[:=** *<initializer>***]**, ... **]]**
    [FIELD *<identifier list>* **[IN** *<idAlias>***]]**
    [MEMVAR *<identifier list>***]**
    .
    . *<executable statements>*
    .
    RETURN *<exp>*
Declares a user-defined function written in CA-Clipper and a series of local variables (formal parameters) to receive passed values and references. If the function is declared STATIC, it is visible only to other procedures and functions declared within the same program (.prg) file; otherwise, it is visible anywhere in the program.

IF *<lCondition1>*
    <statements>...
[ELSEIF *<lCondition2>***]**
    <statements>...
[ELSE]
    <statements>...
END[IF]
Selects a path of program execution from a set of conditions and branches after the first true (.T.) evaluation.

INIT PROCEDURE *<idProcedure>*
    [(<*idParam list*>**)]**
    [FIELD *<idField list>* **[IN** *<idAlias>***]]**
    [LOCAL *<identifier>* **[[:=** *<initializer>***]]]**
    [MEMVAR *<identifier list>***]**
    .
    . *<executable statements>*
    .
    [RETURN]
Declares a procedure that will be executed at program startup.

LOCAL *<identifier>* **[[:=** *<initializer>***]**, ... **]**
Declares one or more local arrays or variables and optionally initializes variables to a specified value.

MEMVAR *<idMemvar list>*
Declares one or more identifiers as memory variables by adding the memory variable alias (MEMVAR->) to undeclared and unaliased variable names in *<idMemvar list>* during compilation.

PARAMETERS *<idPrivate list>*
At runtime, creates private variables to receive passed values or references. Private variables are visible within the creating as well as invoked procedures or user-defined functions. Private variables exist until either explicitly released or the creating procedure or user-defined function terminates.

PRIVATE *<identifier>* [[: = *<initializer>*], ...]
At runtime, creates one or more private arrays or variables and, optionally, initializes variables to a specified value. Private variables are visible within the creating as well as invoked procedures or user-defined functions. Private variables exist until either explicitly released or the creating procedure or user-defined function terminates.

[STATIC] PROCEDURE *<idProcedure>*
[(*<idParam list>***)]**
[FIELD *<idField list>* **[IN** *<idAlias>***]**
[LOCAL *<identifier>* [[:= *<initializer>*], ...]]
[MEMVAR *<identifier list>***]**
[STATIC *<identifier>* [[:= *<initializer>*], ...]]
.
. *<executable statements>*
.
[RETURN]
Declares a procedure written in CA-Clipper and a series of local variables (formal parameters) to receive passed values and references. If the procedure is declared STATIC, it is visible only to other procedures and functions declared within the same program (.prg) file; otherwise, it is visible anywhere in the program.

PUBLIC *<identifier>* [[:= *<initializer>*], ...]
At runtime, creates global variables and arrays and, optionally, initializes variables to the specified values. Public variables are visible everywhere within a program and exist until the program terminates or they are explicitly released.

REQUEST *<idModule list>*
Defines a list of module identifiers to the linker.

RETURN [*<exp>***]**
Terminates a procedure, a user-defined function, or a program by returning control to either the calling procedure or the operating system. For user-defined functions, this statement also defines the function return value.

STATIC *<identifier>* [[:= *<initializer>*], ...]
Declares one or more variables and arrays and, optionally, initializes variables with the specified constant values. Static variables are visible only within the procedure or program (.prg) file in which they are declared and have a lifetime of the entire program.

Chapter 3
Preprocessor Directives

Below is a summary of the CA-Clipper preprocessor directives covered in the "Language Reference" chapter of the *Reference Guide, Volume 1*.

Directives

#command <*matchPattern*>
**   => <*resultPattern*>**
#translate <*matchPattern*>
**   => <*resultPattern*>**

Defines a user-defined command or translation directive.

#define <*idConstant*> [<*resultText*>]
#define <*idFunction*>([<*arg list*>]) [<*exp*>]

Defines a manifest constant or pseudofunction.

#error [<*messageText*>]

Causes the compiler to generate error number C2074. If the *<messageText>* parameter is specified, an error message is displayed.

#ifdef <*identifier*>
**   <*statements*>...**
[#else]
**   <*statements*>...**
[#endif]

Compiles the section of code following #ifdef if the specified *<identifier>* is defined.

#ifndef <*identifier*>
**   <*statements*>...**
[#else]
**   <*statements*>...**
[#endif]

Compiles the section of code following #ifndef if the specified *<identifier>* is *not* defined.

#include "<*headerFileSpec*>"

Inserts the contents of the specified header file in place of the #include directive in the source file.

#stdout [<*messageText*>]

Sends literal text to the standard output device (stdout) during compilation. If *<messageText>* is not specified, a carriage return/line feed pair echoes to stdout.

#undef <*identifier*>

Removes a #define macro definition.

#xcommand <*matchPattern*>
**   => <*resultPattern*>**
#xtranslate <*matchPattern*>
**   => <*resultPattern*>**

Works like #command and #translate except that the four-letter abbreviation rules used by #command and #translate do not apply.

Chapter 4
Commands

Below is a summary of the CA-Clipper commands covered in the "Language Reference" chapter of the *Reference Guide, Volumes 1 and 2.*

Command Reference

? | ?? [<*exp list*>]

Displays the results of one or more expressions, separated by a space to the console. ? precedes the display with a carriage return/linefeed to begin on the next row, whereas ?? displays on the current row.

@ <*nTop*>, <*nLeft*>, <*nBottom*>, <*nRight*>
BOX <*cBoxString*> [COLOR<*cColorString*>]

Draws a box on the screen using the specified coordinates and a user-defined box frame defined by the *<cBoxString>* argument.

@ <nTop>, <nLeft>
[TO <nBottom>, <nRight>]]
(DOUBLE) (COLOR <*cColor*>)

Clears a rectangular region of the screen according to the specified coordinates.

@ <*nRow*>, <*nCol*>
[SAY <*exp*>
[PICTURE <*cSayPicture*>]]
[COLOR <*cColorString*>]
GET <*idVar*>
[PICTURE <*cGetPicture*>]
[COLOR <*cColorString*>]
[CAPTION <*cCaption*>]
[MESSAGE <*cMessage*>]
[WHEN <*lPreExpression*>]
[RANGE* <*dnLower*>, <*dnUpper*>] |
[VALID <*lPostExpression*>]
[SEND<*msg*>]
[GUISEND<*guimsg*>]

Creates a new Get object, displays its value, and adds it to the currently visible *GetList* array. A subsequent invocation of READ or READMODAL() activates the Get objects contained in the *GetList* array and allows the user to edit the Get object's editing buffer.

```
@ <nRow>, <nCol>
    GET <lVar>
    CHECKBOX
        [CAPTION<cCaption>]
        [MESSAGE <cMessage>]
        [WHEN <lPreExpression>]
        [VALID <lPostExpression>]
        [COLOR <cColorString>]
        [FOCUS <fBlock>]
        [STATE <bBlock>]
        [STYLE <cStyle>]
        [SEND <msg>]
        [GUISEND <guimsg>]
        [BITMAPS <aBitmaps>][
```

Creates a new check box Get object and displays it to the screen.

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
    GET <nVar| cVar>
    LISTBOX <aList>
        [CAPTION <cCaption>]
        [MESSAGE <cMessage>]
        [WHEN <lPreExpression>]
        [VALID <lPostExpression>]
        [COLOR <cColorString>]
        [FOCUS <fblock>]
        [STATE <bBlock>]
        [DROPDOWN]
        [SCROLLBAR]
        [SEND <msg>]
        [GUISEND <guimsg>]
        [BITMAP <cBitmap>]
```

Creates a new list box Get object and displays it to the screen.

```
@ <nRow>, <nCol>
    GET <lVar>
    PUSHBUTTON
        [CAPTION<cCaption>]
        [MESSAGE <cMessage>]
        [WHEN <lPreExpression>]
        [VALID <lPostExpression>]
        [COLOR <cColorString>]
        [FOCUS <fblock>]
        [STATE <bBlock>]
        [STYLE <cStyle>]
        [SEND<msg>]
        [GUISEND<guimsg>]
        [SIZE X <nSizeX> Y <nSizeY>]
        [CAPOFF X <nCapXOff> Y <nCapYOff>]
        [BITMAP <cBitmap>]
        [BMPOFF X <nBmpXOff> Y
<nBmpYOff>]
```

Creates a new push button Get object and displays it to the screen.

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
    GET <nVar| cVar>
    RADIOGROUP <aGroup>
        [CAPTION<cCaption>]
        [MESSAGE <cMessage>]
        [COLOR <cColorString>]
        [FOCUS <fblock>]
        [WHEN <lPreExpression>]
        [VALID <lPostExpression>]
        [SEND<msg>]
        [GUISEND<guimsg>]
```

Creates a new radio button group Get object and displays it to the screen.

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
    GET <idVar>
    TBROWSE <oBrowse>
        [MESSAGE <cMessage>]
        [WHEN <lPreExpression>]
        [VALID <lPostExpression>]
        [SEND<msg>]
        [GUISEND<guimsg>]
```

Creates a new TBrowse Get object and displays it to the screen.

@ *<nRow>*, *<nCol>
PROMPT *<cMenuItem>*
[MESSAGE *<cExpression>*]

Paints a lightbar menu item at the specified screen coordinates and defines an associated message for the item. Several @...PROMPT commands together define a lightbar menu activated with the MENU TO command.

@ *<nRow>*, *<nCol>*
SAY *<exp>* [PICTURE *<cSayPicture>*]
[COLOR *<cColorString>*]

Displays the result of the expression at the specified row and column positions on either the screen or the printer. PICTURE determines how the data is formatted when displayed.

@ *<nTop>*, *<nLeft>*
TO *<nBottom>*, *<nRight>*
[DOUBLE] [COLOR *<cColorString>*]

Draws a single- or double-line box on the screen using the specified coordinates.

ACCEPT [*<expPrompt>*] TO *<idVar>

Prompts the user for keyboard input and assigns the input to the specified variable as a character string. ACCEPT is a wait state command.

APPEND BLANK

Adds a blank record to the end of the current database file.

APPEND FROM *<xcFile>*
[FIELDS *<idField list>*]
[*<scope>*] [WHILE *<lCondition>*]
[FOR *<lCondition>*]
[SDF | DELIMITED
[WITH BLANK | *<xcDelimiter>*] |
[VIA *<cDriver>*]

Adds records to the current database file from another database file or from an ASCII text file if SDF or DELIMITED is specified.

AVERAGE *<nExp list>* TO *<idVar list>*
[*<scope>*] [WHILE *<lCondition>*]
[FOR *<lCondition>*]

Averages one or more numeric expressions for a range of records in the current work area and assigns the results to the corresponding variables as numeric values.

***CALL *<idProcedure>* [WITH *<exp list>*]**

Executes a separately compiled or assembled routine, passing parameters if specified.

***CANCEL | QUIT**

Terminates program execution by returning control to the operating system after closing all open files.

***CLEAR ALL**

Closes all open database (and related index, format, and memo) files, releases all memory variables, and then SELECTs work area 1.

CLEAR GETS

Deletes all Get objects contained in the current and visible *GetList* array, and terminates the current READ if executed within a SET KEY procedure or a user-defined function invoked by a VALID clause.

CLEAR MEMORY

Releases all public and private variables from memory.

CLEAR [SCREEN] | CLS

CLEAR SCREEN and CLS clear the screen and home the cursor. CLEAR performs the same action, but also CLEARs GETS.

CLEAR TYPEAHEAD

Clears the keyboard buffer of all pending characters.

CLOSE [<*idAlias*> | ALL | ALTERNATE | DATABASES | FORMAT | INDEXES]

Closes all files of the specified type. If <*idAlias*> is specified, files open in the specified work area are CLOSEd. If no arguments are specified, files in the current work area are CLOSEd.

COMMIT

Flushes CA-Clipper buffers for all work areas with open database and index files. Under DOS version 3.3 and higher, COMMIT performs a solid-disk write.

CONTINUE

Resumes the pending LOCATE search in the current work area, ignoring any scope or WHILE condition.

COPY FILE <*xcSourceFile*> TO <*xcTargetFile*> | <*xcDevice*>

Duplicates a file of any kind in the current CA-Clipper default drive and directory.

COPY STRUCTURE [FIELDS <*idField list*>] TO <*xcDatabase*>

Creates an empty database file with field definitions from the current database file.

COPY STRUCTURE EXTENDED TO <*xcExtendedDatabase*>

Creates a structure extended file with four fields: Field_name, Field_type, Field_len, and Field_dec. The records of this new database file are the field definitions of the current database file.

COPY [FIELDS <*idField list*>] TO <*xcFile*> [<*scope*>] [WHILE <*lCondition*>] [FOR <*lCondition*>] [SDF | DELIMITED [WITH BLANK | <*xcDelimiter*>] | [VIA <*xcDriver*>]]

Copies records from the current database file to another database file or to an ASCII text file if SDF or DELIMITED is specified.

COUNT TO <*idVar*> [<*scope*>] [WHILE <*lCondition*>] [FOR <*lCondition*>]

Tallies the number of records in the current work area for the specified scope and conditions. The result is assigned to the specified variable as a numeric value.

CREATE <*xcExtendedDatabase*>

Creates an empty structure extended file.

CREATE <*xcDatabase*> FROM <*xcExtendedDatabase*> [NEW] [ALIAS <*xcAlias*> [VIA <*cDriver*>]

Creates a new database file with a structure defined by the specified structure extended file.

DELETE [<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>]

Marks records for deletion in the current work area for the specified scope and conditions. Deleted records are not physically removed until a PACK is issued and can be reinstated with RECALL before they are physically removed.

DELETE FILE | ERASE <*xcFile*>

Deletes the specified file from disk.

DELETE TAG <*cOrderName*>
[IN <*xcOrderBagName*>]
[, <*cOrderName*>
[IN <*xcOrderBagName*>] list>]

Removes an order from an order bag in the current or specified work area.

***DIR [<*xcFileSpec*>]**

Displays files matching the given file specification. If no file specification is included on the command line, only database (.dbf) files in the current SET DEFAULT directory are displayed.

DISPLAY <*exp list*>
[TO PRINTER] [TO FILE <*xcFile*>]
[<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>] [OFF]

Displays the result of one or more expressions for a range of records in the current work area.

EJECT

Advances the printhead to the top-of-form and resets PROW() and PCOL() to zero.

ERASE | DELETE FILE <*xcFile*>

Deletes the specified file from disk. The file must include an extension if one is to be assumed, and it must be closed.

***FIND <*xcSearchString*>**

Searches the controlling index in the current work area for the first record with an index key that matches the value of the specified expression.

GO[TO] <*xIdentity*> | BOTTOM | TOP

Moves the record pointer to a specific record in the current work area or directly to the TOP or BOTTOM record. If index files are open, the record pointer moves relative to the controlling index.

INDEX ON <*expKey*> [TAG <*cOrderName*>]
[TO <*cOrderBagName*>]
[FOR <*lCondition*>] [ALL]
[WHILE <*lCondition*>] [NEXT <*nNumber*>]
[RECORD <*nRecord*>] [REST]
[EVAL <*bBlock*>] [EVERY <*nInterval*>]
[UNIQUE] [ASCENDING | DESCENDING]
[USECURRENT] [ADDITIVE]
[CUSTOM] [NOOPTIMIZE]

Adds a set of keyed pairs, ordered by <*expKey*> to a file specified by <*cOrderBagName*> using the database open in the current work area.

***INPUT [<*expPrompt*>] TO <*idVar*>**

Prompts the user for keyboard input, evaluates the input as an expression, and then assigns the result to the specified variable. INPUT is a wait state command.

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

Generates a new database file by joining the current work area and a second work area and eliminating all records that do not meet the specified FOR condition.

KEYBOARD <*cString*>

Stuffs the keyboard buffer with the keystrokes indicated by the specified character string. The keystrokes remain in the keyboard buffer until extracted by a wait state command or function, or INKEY().

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

Displays labels from a definition held in a label (.lbl) file for a range of records in the current work area. Label files are created with RL.EXE.

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

Displays the result of one or more expressions for a range of records in the current database file.

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

Positions the record pointer to the first record in the current work area that matches the specified condition within the given scope.

MENU TO <*idVar*>

Activates a lightbar menu for the currently defined @...PROMPT menu items and assigns the user's selection to the specified variable as a numeric value.

***NOTE [<*commentText*>]**

Places a single-line comment in a program (.prg) file.

PACK

Permanently removes records marked for deletion from the database file open in the current work area.

QUIT | CANCEL*

Terminates program execution by returning control to the operating system after closing all open files.

READ [SAVE] [MENU <*oMenu*>]
[MSG AT <*nRow*>, <*nLeft*>, <*nRight*>,] ,
[MSG COLOR <*cColorString*>]

Activates a full-screen editing mode for Get objects contained in the currently visible *GetList* array. READ is a wait state command.

RECALL
[<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>]

Restores records marked for deletion in the current database file. A range of records can be specified with a scope and conditions; otherwise, only the current record is reinstated.

REINDEX [EVAL <*lCondition*>]
[EVERY <*nRecords*>]

Rebuilds all open index files in the current work area.

RELEASE <*idMemvar list*>
RELEASE ALL [LIKE | EXCEPT <*skeleton*>]

Releases all or a specified subset of public and private variables from memory.

RENAME <*xcOldFile*> TO <*xcNewFile*>

Provides a new name for an existing file.

REPLACE <*idField*> WITH <*exp*>
[, <*idField2*> WITH <*exp2*> ...]
[<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>]

Replaces the contents of one or more fields in the current database file with corresponding values. A range of records can be specified with a scope and conditions; otherwise, only the current record is affected.

REPORT FORM <*xcReport*>
[TO PRINTER] [TO FILE <*xcFile*>]
[NOCONSOLE] [<*scope*>]
[WHILE <*lCondition*>]
[FOR <*lCondition*>]
[PLAIN | HEADING <*cHeading*>]
[NOEJECT] [SUMMARY]

Displays a tabular and optionally grouped report with page and column headings for a range of records in the current database file. The report definition is held in a report (.frm) file that may be created with RL.EXE.

RESTORE FROM <*xcMemFile*> [ADDITIVE]

Retrieves memory variables from a memory (.mem) file previously created with SAVE.

***RESTORE SCREEN [FROM <*cScreen*>]**

Redisplays a screen previously stored with SAVE SCREEN.

RUN | *! <*xcCommandLine*>

Executes a DOS command or program.

SAVE TO <*xcMemFile*>
[ALL [LIKE | EXCEPT <*skeleton*>]]

Saves all or a specified subset of the currently visible private and public variables containing values of simple data types to a memory (.mem) file. Complex data types such as arrays, code blocks, and objects as well as local and static variables also cannot be saved.

***SAVE SCREEN [TO <*idVar*>]**

Stores the current screen to a buffer or assigns it to a variable if TO <*idVar*> is specified. The saved screen can later be redisplayed with RESTORE SCREEN.

SEEK <*expSearch*> [SOFTSEEK]

Searches the controlling index in the current work area for the first record with an index key that matches the value of the specified expression or is greater than <*expSearch*>.

SELECT <*xnWorkArea*> | <*idAlias*>

Changes work areas. If the work area has a database file open, it can be referred to by its alias or its work area number. SELECT 0 changes to the first unoccupied work area.

SET ALTERNATE TO [<*xcFile*> [ADDITIVE]]
SET ALTERNATE on | OFF | <*xlToggle*>

Creates and opens a text file for capturing output from console commands and controls writing to that file.

SET BELL on | OFF | <*xlToggle*>

Determines whether or not the bell rings during data entry operations.

SET CENTURY on | OFF | <*xlToggle*>

Includes or omits the century in date displays.

***SET COLOR | COLOUR TO [[*<standard>*] [, *<enhanced>*] [,*<border>*] [, *<background>*] [,*<unselected>*]] | (*<cColorString>*)**

Defines the screen colors and display attributes.

SET CONFIRM on | OFF | *<xlToggle>*

Determines whether an exit key is required to terminate an @...GET.

SET CONSOLE ON | off | *<xlToggle>*

Determines whether or not the output of console commands is displayed on the screen. Full-screen commands are not affected by SET CONSOLE.

SET CURSOR ON | off | *<xlToggle>*

Controls the display of the screen cursor.

SET DATE FORMAT [TO] *<cDateFormat>*
SET DATE [TO] AMERICAN | Ansi | British | French | German | Italian | Japan | USA

Sets the default format in which dates are entered and displayed.

SET DECIMALS TO [*<nDecimals>*]

Sets the number of decimal places displayed for the results of numeric functions and calculations.

SET DEFAULT TO [*<xcPathspec>*]

Specifies the default drive and directory for creating and saving files.

SET DELETED on | OFF | *<xlToggle>*

Determines whether records that are marked for deletion are hidden or processed.

SET DELIMITERS on | OFF | *<xlToggle>*
SET DELIMITERS TO [*<cDelimiters>* | DEFAULT]

Determines whether or not delimiters display for @...GETs, and defines the delimiter characters.

SET DESCENDING ON | OFF | (*<lToggle>*)

Changes the descending flag of the controlling order.

SET DEVICE TO SCREEN | printer

Determines whether the results of @...SAY are displayed to the screen or the printer.

SET EPOCH TO *<nYear>*

Controls the interpretation of dates with no century digits.

SET ESCAPE ON | off | *<xlToggle>*

Determines whether or not Esc may be used as a READ exit key.

SET EVENTMASK TO *<nEventMask>*

Specifies mouse events to be returned by the INKEY() function.

SET EXACT on | OFF | *<xlToggle>

Determines whether or not exact matches are required in order for two character strings to compare as equal.

SET EXCLUSIVE ON | off | *<xlToggle>

Determines the default open mode for database files in a network environment as either shared or exclusive (nonshared).

SET FILTER TO [*<lCondition>*]

Causes the database file in the current work area to appear as if it contains only records that meet the specified condition.

SET FIXED on | OFF | *<xlToggle>*

Determines whether or not the current SET DECIMALS value is used as a fixed setting.

***SET FORMAT TO [<idProcedure>[.<ext>]]**

Activates a format that executes whenever a READ is encountered.

SET FUNCTION *<nFunctionKey>* TO *<cString>*

Defines a character string to be stuffed into the keyboard buffer during a wait state when the user presses the specified function key.

SET INDEX TO [*<xcOrderBagName list>*]
[ADDITIVE]

Opens one or more index files in the current work area. SET INDEX TO with no files specified closes all open index files in the current work area.

SET INTENSITY ON | off | *<xlToggle>*

Determines whether Get objects are displayed using the enhanced or the standard color setting.

SET KEY *<nInkeyCode>* TO [*<idProcedure>*]

Assigns a procedure to execute from any wait state when the designated key is pressed. A wait state is any command or function (except INKEY()) that pauses program execution (e.g., ACCEPT and ACHOICE()).

SET MARGIN TO [*<nPageOffset>*]

Sets the left margin, or page offset, for all printed output.

SET MEMOBLOCK TO *<nSize>*

Changes the block size for the memo file associated with the database.

SET MESSAGE TO
[*<nRow>* [CENTER | CENTRE]]

Sets the row where messages from the MenuItem class and GUI objects are displayed.

SET OPTIMIZE ON | OFF | [*<lToggle>*]

Changes the setting that determines whether to optimize using the open orders when processing a filtered database file.

SET ORDER TO [*<nOrder>* |
[TAG*<cOrderName>*]
[IN *<xcOrderBagName>*]]

Specifies which open index file in the current work area is the controlling index.

SET PATH TO [*<xcPathspec list>*]

Specifies the path that CA-Clipper searches when attempting to open files.

SET PRINTER on | OFF | *<xlToggle>*
SET PRINTER TO [*<xcDevice>* | *<xcFile>*
[ADDITIVE]]

Determines whether the output of console commands is directed to the printer, and additionally specifies the destination of the printed output.

***SET PROCEDURE TO [*<idProgramFile>* [.*<ext>*]]**

Compiles all procedures and functions in the named program file into the current .OBJ file.

SET RELATION TO
[*<expKey>* | *<nRecord>* INTO *<xcAlias>*]
[, [TO] *<expKey2>* | *<nRecord2>* INTO
***<xcAlias2>* ...] [ADDITIVE]**

Relates the database file in the current work area to one or more database files by a key value or record number.

SET SCOPE TO [<*expNewTop*>
[, <*expNewBottom*>]]

Changes the top and bottom boundaries for scoping key values in the controlling order.

SET SCOPEBOTTOM TO [<*expNewBottom*>]

Changes the bottom boundary for scoping key values in the controlling order.

SET SCOPETOP TO [<*expNewTop*>]

Changes the top boundary for scoping key values in the controlling order.

SET SCOREBOARD ON | off | <*xlToggle*>

Determines whether READ and MEMOEDIT() display messages that appear on line zero.

SET SOFTSEEK on | OFF | <*xlToggle*>

Determines whether relative seeking is performed. If SET SOFTSEEK ON and a key value is not found, the record pointer is positioned to the record with the next higher index key value. SET SOFTSEEK OFF moves the record pointer to EOF().

SET TYPEAHEAD TO <*nKeyboardSize*>

Sets the size of the keyboard buffer.

***SET UNIQUE on | OFF | <*xlToggle*>**

Determines whether or not UNIQUE is the default attribute when creating index files with INDEX ON.

SET VIDEOMODE TO <*nVideoMode*>

Changes the current display to text mode and other graphic modes.

***SET WRAP on | OFF | <*xlToggle*>**

Determines whether or not an @...PROMPT lightbar menu highlight wraps around when the menu is activated with MENU TO.

SKIP [<*nRecords*>]
[ALIAS <*idAlias*> | <*nWorkArea*>]

Moves the record pointer either forward or backward the specified number of records in the indicated work area.

SORT TO <*xcDatabase*>
ON <*idField1*> [/[A | D] [C]]
[, <*idField2*> [/[A | D] [C]] ...]
[<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>]

Copies records within the specified scope and condition from the current work area to another database file sorted according to the specified fields.

***STORE <*exp*> TO <*idVar list*>**
OR
<*idVar*> = <*exp*>
OR
<*idVar*> := [<*idVar2*> := ...] <*exp*>

Stores the result of an expression to one or more variables.

SUM <*nExp list*> TO <*idVar list*>
[<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>]

Totals one or more numeric expressions for a range of records in the current work area, and assigns the results to the corresponding variables.

***TEXT [TO PRINTER] [TO FILE <*xcFile*>]**
<*text*>...
ENDTEXT

Displays a block of text. Macro variables found within the text are expanded, macro expressions are not.

TOTAL ON <*expKey*>
[FIELDS <*idField list*>] TO <*xcDatabase*>
[<*scope*>] [WHILE <*lCondition*>]
[FOR <*lCondition*>]

Summarizes records in the current work area by key value, summing the specified numeric fields and then copying summary records to a second database file.

TYPE <*xcFile*>
[TO PRINTER] [TO FILE <*xcOutFile*>]

Displays the contents of a text file. The file must include an extension if one is to be assumed, and the file must be closed.

UNLOCK [ALL]

Releases file and record locks set previously by FLOCK() or RLOCK().

UPDATE FROM <*xcAlias*>
ON <*expKey*> [RANDOM]
REPLACE <*idField*> WITH <*exp*>
[, <*idField2*> WITH <*exp2*> ...]

Updates the database file in the current work area from another database file based on the specified key expression.

USE [<*xcDatabase*>
[INDEX <*xcIndex list*>]
[ALIAS <*xcAlias*>]
[EXCLUSIVE | SHARED]
[NEW] [READONLY]
[VIA <*cDriver*>]]

Opens an existing database (.dbf) file, its associated memo (.dbt) file, and optional index (.ntx/.ndx) file(s) in the current work area. If no arguments are specified, USE closes the current database file.

***WAIT [<*expPrompt*>] [TO <*idVar*>]**

Prompts for single-character keyboard input and optionally assigns the key pressed to the specified variable. WAIT is a wait state command.

ZAP

Permanently removes all records from the database file open in the current work area.

Chapter 5
Functions

Below is a summary of the CA-Clipper functions covered in the "Language Reference" chapter of the *Reference Guide, Volumes 1 and 2.*

Function Reference

AADD(<*aTarget*>, <*expValue*>) → *Value*

Increases the size of <*aTarget*> by one and initializes the new element to <*expValue*>.

ABS(<*nExp*>) → *nPositive*

Returns the absolute value of <*nExp*> as a numeric value.

ACHOICE(<*nTop*>, <*nLeft*>, <*nBottom*>, <*nRight*>, <*acMenuItems*>, [<*alSelectableItems*>| <*lSelectableItems*>], [<*cUserFunction*>], [<*nInitialItem*>], [<*nWindowRow*>]) → *nPosition*

Executes a pop-up menu using an array of character strings as choices and returns the selection as a numeric value. ACHOICE() is a wait state function.

ACLONE(<*aSource*>) → *aDuplicate*

Duplicates <*aSource*> including subarrays.

ACOPY(<*aSource*>, <*aTarget*>, [<*nStart*>], [<*nCount*>], [<*nTargetPos*>]) → *aTarget*

Copies elements from <*aSource*> to <*aTarget*>.

ADEL(<*aTarget*>, <*nPosition*>) → *aTarget*

Deletes array element <*nPosition*> from <*aTarget*> and moves the elements below up one position.

ADIR([<*cFilespec*>], [<*aFilenames*>], [<*aSizes*>], [<*aDates*>], [<*aTimes*>], [<*aAttributes*>]) → *nFiles

Fills a series of arrays with file information from the disk directory and returns the number of files matching the skeleton.

AEVAL(<*aArray*>, <*bBlock*>, [<*nStart*>], [<*nCount*>]) → *aArray*

Executes a code block for each element in <*aArray*>. The code block is passed the value of each array element as an argument.

AFIELDS([<*aFieldNames*>], [<*aTypes*>], [<*aWidths*>], [<*aDecimals*>]) → *nFields

Fills a series of arrays with field definition information and returns the number of fields.

AFILL(<*aTarget*>, <*expValue*>, [<*nStart*>], [<*nCount*>]) → *aTarget*

Fills <*aTarget*> with <*expValue*> starting at element <*nStart*>.

AINS(<*aTarget*>, <*nPosition*>) → *aTarget*

Inserts a NIL element into <*aTarget*> at position <*nPosition*> and moves all elements below down one position.

ALERT(<*cMessage*>, [<*aOptions*>])
→ *nChoice*

Creates a simple modal dialog displaying <*cMessage*> as a message centered within the box and <*aOptions*> as a list of all possible options. The user can respond by moving a highlight bar and pressing the Return key or Space bar, or by pressing the key corresponding to the first letter of the option. If <*aOptions*> is not supplied, a single "OK" option is presented.

ALIAS([<*nWorkArea*>]) → *cAlias*

Returns the alias of the work area specified by <*nWorkArea*> as a character string.

ALLTRIM(<*cString*>) → *cTrimString*

Returns <*cString*> with leading and trailing spaces removed.

ALTD([<*nAction*>]) → *NIL*

Executes the CA-Clipper debugger or enables/disables the use of Alt-D to invoke it.

ARRAY(<*nElements*>
[, <*nElements*> ...]) → *aArray*

Creates an uninitialized array with the specified dimensions.

ASC(<*cExp*>) → *nCode*

Returns the ASCII code value of the leftmost character in <*cExp*> as a numeric value.

ASCAN(<*aTarget*>, <*expSearch*>, [<*nStart*>],
[<*nCount*>]) → *nStoppedAt*

Searches <*aTarget*> for <*expSearch*> starting with element <*nStart*> and returns the array position as a numeric value. If <*expSearch*> is a code block, it is evaluated with the value of the current element passed as an argument. If the code block returns true (.T.), the scanning operation terminates; otherwise, it continues.

ASIZE(<*aTarget*>, <*nLength*>) → *aTarget*

Changes the size of <*aTarget*> to <*nLength*>.

ASORT(<*aTarget*>, [<*nStart*>], [<*nCount*>],
[<*bOrder*>]) → *aTarget*

Sorts <*aTarget*> starting with element <*nStart*>. The sort criteria can be optionally specified as the code block, <*bOrder*>.

AT(<*cSearch*>, <*cTarget*>) → *nPosition*

Returns the starting position of <*cSearch*> within <*cTarget*> as a numeric value.

ATAIL(<*aArray*>) → *Element*

Returns the highest numbered element in <*aArray*>.

BIN2I(<*cSignedInt*>) → *nNumber*

Converts <*cSignedInt*> formatted as a 16-bit signed integer to a CA-Clipper numeric value.

BIN2L(<*cSignedInt*>) → *nNumber*

Converts <*cSignedInt*> formatted as a 32-bit signed integer to a CA-Clipper numeric value.

BIN2W(<*cUnsignedInt*>) → *nNumber*

Converts <*cUnsignedInt*> formatted as a 16-bit unsigned integer to a CA-Clipper numeric value.

BLOBDIRECTEXPORT(<*nPointer*>, <*cTargetFile*>, [<*nMode*>]) → *lSuccess*

Exports the contents of a binary large object (BLOB) pointer to a file.

BLOBDIRECTGET(<*nPointer*>, [<*nStart*>],[<*nCount*>]) → *expBLOB*

Retrieves data stored in a BLOB file without referencing a specific field.

BLOBDIRECTIMPORT(<*nOldPointer*>, <*cSourceFile*>) → *nNewPointer*

Provides a mechanism for copying the contents of a file into a BLOB file.

BLOBDIRECTPUT(<*nOldPointer*>, <*uBLOB*>) → *nNewPointer*

Stores variable length BLOB data without creating a link with a particular memo field in a database file.

BLOBEXPORT(<*nFieldPos*>, <*cTargetFile*>, [<*nMode*>]) → *lSuccess*

Copies the contents of a BLOB, identified by its memo field number, to a file.

BLOBGET(<*nFieldPos*>, [<*nStart*>], [<*nCount*>]) → *uBLOB*

Gets the contents of a BLOB, identified by its memo field number.

BLOBIMPORT(<*nFieldPos*>, <*cSourceFile*>) → *lSuccess*

Reads the contents of a file as a BLOB, identified by a memo field number.

BLOBROOTGET() → *uBLOB*

Allows the retrieval of a BLOB from the root of a BLOB file in a work area.

BLOBROOTLOCK() → *lSuccess*

Accesses the database file in shared mode to obtain a lock on the root area of a BLOB file for reading from or writing to the root area.

BLOBROOTPUT(<*uBLOB*>) → *lSuccess*

Allows the storage of one piece of data to a BLOB file's root area and releases the space associated with any data previously stored in the BLOB file's root area.

BLOBROOTUNLOCK() → *NIL*

Releases the lock on a BLOB file's root area.

BOF() → *lBoundary*

Returns true (.T.) when the beginning of the file is encountered in the current work area.

BREAK(<*exp*>) → *NIL*

Branches out of the current SEQUENCE exactly like a BREAK statement. <*exp*> is the value passed to the RECOVER clause. BREAK() has the advantage that, as an expression, it can be executed from a code block.

BROWSE([<*nTop*>], [<*nLeft*>], [<*nBottom*>], [<*nRight*>]) → *lSuccess

Invokes a general purpose table-oriented record browser and editor for records in the current work area. If screen coordinates are specified, the editor operates in the specified window size. Otherwise, BROWSE() uses the default coordinates 0, 0 to MAXROW(), MAXCOL().

CDOW(<*dExp*>) → *cDayName*

Returns the day of the week name from <*dExp*> as a character string.

CHR(<*nCode*>) → *cChar*

Returns a character for the ASCII code specified by <*nCode*>.

CMONTH(<*dDate*>) → *cMonth*

Returns the month name from <*dDate*> as a character string.

COL() → *nCol*

Returns the column position of the screen cursor as a numeric value.

COLORSELECT(<*nColorIndex*>) → *NIL*

Activates the specified color pair from the current list of color attributes as established by SETCOLOR().

CTOD(<*cDate*>) → *dDate*

Converts <*cDate*> to a date value.

CURDIR([<*cDrivespec*>]) → *cDirectory*

Returns the current DOS directory path of the drive specified by <*cDrivespec*> as a character string.

DATE() → *dSystem*

Returns the system date as a date value.

DAY(<*dDate*>) → *nDay*

Returns the day of the month from <*dDate*> as a numeric value.

DBAPPEND([<*lReleaseRecLocks*>]) → *NIL*

Adds a new record to the database (.dbf) file associated with the current work area. If successfully added, each field in the record is set to the empty value for its data type and the new record becomes the current record.

DBCLEARFILTER() → *NIL*

Clears the logical filter condition, if any, for the current work area.

DBCLEARINDEX() → *NIL*

Closes any active indexes for the current work area. Any pending index updates are written and the index files are closed.

DBCLEARRELATION() → *NIL*

Clears any active relations for the current work area.

DBCLOSEALL() → *NIL*

Releases all occupied work areas from use.

DBCLOSEAREA() → *NIL*

Releases the current work area from use. Pending updates are written, pending locks are released, and any resources associated with the work area are closed or released.

DBCOMMIT() → *NIL*

Causes all updates to the current work area to be written to disk. All updated database and index buffers are written to DOS and a DOS COMMIT request is issued for the database (.dbf) file and any index files associated with the work area.

DBCOMMITALL() → *NIL*

Causes all pending updates to all work areas to be written to disk.

DBCREATE(<*cDatabase*>, <*aStruct*> [<*cDriver*>]) → *NIL*

Creates a database file from a database structure array.

DBCREATEINDEX(<*cIndexName*>, <*cKeyExpr*>, [<*bKeyExpr*>], [<*lUnique*>]) → *NIL*

Creates <*cIndexName*> as an index for the database file associated with the current work area. If the work area has active indexes, they are closed. After the new index is created, it becomes the controlling index for the work area and the work area is positioned to the first logical record.

DBDELETE() → *NIL*

Marks the current record as deleted.

DBEDIT([<*nTop*>], [<*nLeft*>], [<*nBottom*>], [<*nRight*>], [<*acColumns*>], [<*cUserFunction*>], [<*acColumnSayPictures*> | <*cColumnSayPicture*>], [<*acColumnHeaders*> | <*cColumnHeader*>], [<*acHeadingSeparators*>| <*cHeadingSeparator*>], [<*acColumnSeparators*> | <*cColumnSeparator*>], [<*acFootingSeparators*> | <*cFootingSeparator*>], [<*acColumnFootings*> | <*cColumnFooting*>]) → *NIL*

Displays and edits records from one or more work areas using a browse-style editor that executes within a window area defined by the specified screen coordinates. DBEDIT() is a wait state function.

DBEVAL(<*bBlock*>, [<*bForCondition*>], [<*bWhileCondition*>], [<*nNextRecords*>], [<*nRecord*>], [<*lRest*>]) → *NIL*

Evaluates the code block, <*bBlock*>, for each record matching a scope and condition.

DBF() → *cAlias

Returns the alias of the current work area as a character string.

DBFIELDINFO(<*nInfoType*>, <*nFieldPos*>, [<*expNewSetting*>]) → *uCurrentSetting*

Returns and optionally changes information about the state of a field.

DBFILTER() → *cFilter*

Returns the SET FILTER expression for the current work area as a character string.

DBGOBOTTOM() → *NIL*

Moves to last logical record in the current work area.

DBGOTO(<*xIdentity*>) → *NIL*

DBGOTO() is a database function that positions the record pointer in the current work area at the specified <*xIdentity*>. In an Xbase data structure, this identity is the record number because every record, even an empty record, has a record number. In non-Xbase data structures, identity may be defined as something other than record number.

DBGOTOP() → *NIL*

Moves to the first logical record in the current work area.

DBINFO(<*nInfoType*>, [<*expNewSetting*>]) → *uCurrentSetting*

Returns and optionally changes information about a database file opened in a work area.

DBORDERINFO(<*nInfoType*>, [<*cIndexFile*>], [<*cOrder*> | <*nPosition*>], [<*expNewSetting*>]) → *uCurrentSetting*

Returns and optionally changes information about orders and index files.

DBRECALL() → *NIL*

Causes the current record to be reinstated if it is marked for deletion.

DBRECORDINFO(<*nInfoType*>, [<*nRecord*>], [<*expNewSetting*>]) → *uCurrentSetting*

Returns and optionally changes information about a record and retrieves information about the state of a record (row). DBRECORDINFO() allows for additional <*nInfoType*> values that can be defined by third-party RDD developers.

DBREINDEX() → *NIL*

Rebuilds all active indexes associated with the current work area. After the indexes are recreated, the work area is moved to the first logical record in the controlling order.

DBRELATION(<*nRelation*>) → *cLinkExp*

Returns the SET RELATION expression of the relation specified by <*nRelation*> for the current work area as a character string.

DBRLOCK([<*xIdentity*>]) → *lSuccess*

Lock the record at the current or specified identity.

DBRLOCKLIST() → *aRecordLocks*

Returns an array of the current lock list.

DBRSELECT(<*nRelation*>) → *nWorkArea*

Returns the target work area of the SET RELATION specified by <*nRelation*> for the current work area as a numeric value.

DBRUNLOCK([<*xIdentity*>]) → *NIL*

Releases all or specified record locks.

DBSEEK(<*expKey*>, [<*lSoftSeek*>], [<*lLast*>]) → *lFound*

Moves to the first logical record whose key value is equal to <*expKey*>. If such a record is found, it becomes the current record and DBSEEK() returns true (.T.) Otherwise, DBSEEK() returns false (.F.) and the positioning of the work area is as follows: for a normal seek (<*lSoftSeek*> is false (.F.) or omitted), the work area is positioned to LASTREC() + 1 and EOF() returns true (.T.); for a soft seek (<*lSoftSeek*> is true (.T.)), the work area is positioned to the first record whose key value is greater than the specified key value. If no such record exists, the work area is positioned to LASTREC() + 1 and EOF() returns true (.T.).

DBSELECTAREA(<*nArea*> | <*cAlias*>) → *NIL*

Causes the work area specified by <*cAlias*> or <*nArea*> to become the current work area.

DBSETDRIVER([<*cDriver*>]) → *cCurrentDriver*

Sets <*cDriver*> as the default database driver to use when activating new work areas and returns the name of the current default driver, if any. If the specified driver is not available to the application, the call has no effect.

DBSETFILTER(<*bCondition*>, [<*cCondition*>]) → *NIL*

Sets a logical filter condition for the current work area. The filter condition may be supplied either as a code block (<*bCondition*>) or as both a code block and equivalent text (<*cCondition*>). If both versions are supplied, they must express the same condition. If the text version is not supplied, DBFILTER() will return an empty string for the work area.

DBSETINDEX(<*cOrderBagName*>) → *NIL*

Adds the contents of an order bag into the order list of the current work area. Any index files already associated with the work area continue to be active. If the newly opened order bag is the only order associated with the work area, it becomes the controlling order; otherwise, the controlling order remains unchanged.

DBSETORDER(<*nOrderNum*>) → *NIL*

Changes the controlling index in the active work area to the index specified by <*nOrderNum*>.

DBSETRELATION(<*nArea*> | <*cAlias*>, <*bExpr*>, <*cExpr*>) → *NIL*

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.

DBSKIP([<*nRecords*>]) → *NIL*

Moves the record pointer in the current work area <*nRecords*> relative to the current record. If <*nRecords*> is positive, the record pointer moves forward. If it is negative, the record pointer moves backward.

DBSTRUCT() → *aStruct*

Creates an array containing the structure of a database file.

DBUNLOCK() → *NIL*

Releases any record or file locks obtained by the current process for the current work area. DBUNLOCK() is only meaningful on a shared database in a network environment.

DBUNLOCKALL() → *NIL*

Releases any record or file locks obtained by the current process for any work area. DBUNLOCKALL() is only meaningful on a shared database in a network environment.

DBUSEAREA([<*lNewArea*>], [<*cDriver*>], <*cName*>, [<*xcAlias*>], [<*lShared*>], [<*lReadonly*>]) → *NIL*

Opens the database (.dbf) file specified by <*cName*> in the current or next available work area.

DELETED() → *lDeleted*

Returns the deleted status of the current record as a logical value.

DESCEND(<*exp*>) → *ValueInverted*

Returns an inverted expression of the same data type as the <*exp*> to create (i.e., INDEX ON) and SEEK descending order indexes.

DEVOUT(<*exp*>, [<*colorString*>]) → *NIL*

Writes the value of <*exp*> to the current device. <*cColorString*> defines the display color of <*exp*>.

DEVOUTPICT(<*exp*>, <*cPicture*>, [<*cColorString*>]) → *NIL*

Writes the value of a single expression to the current device at the current cursor or printhead position.

DEVPOS(<*nRow*>, <*nCol*>) → *NIL*

Moves the printhead to the specified row and column position.

DIRCHANGE(<*cDir*>) → *nSuccess*

Changes the current DOS directory and determines whether or not a directory exists.

DIRECTORY(<*cDirSpec*>, [<*cAttributes*>]) → *aDirectory*

Creates an array of directory and file information.

DIRMAKE(<*cNewDir*>) → *nSuccess*

Creates a specified directory, creating each subdirectory separately, starting from the top-level directory that is to be created.

DIRREMOVE(<*cDirName*>) → *nSuccess*

Removes a specified directory provided it is empty.

DISKCHANGE(<*cDrive*>) → *lSuccess*

Changes the current DOS disk drive.

DISKNAME() → *cDrive*

Returns the current DOS drive.

DISKSPACE([<*nDrive*>]) → *nBytes*

Returns the number of bytes available on the disk drive specified by <*nDrive*>.

DISPBEGIN() → *NIL*

Informs the CA-Clipper display output system that the application is about to perform a series of display operations.

DISPBOX(<*nTop*>, <*nLeft*>, <*nBottom*>, <*nRight*>, [<*cnBoxString*>], [<*cColorString*>]) → *NIL*

Draws a box using the specified screen coordinates and the specified color. If <*cnBoxString*> is specified as a character string, DISPBOX() uses its characters to define the box border characters starting from the upper left-hand corner and proceeding clockwise. The ninth character is used as a fill character for the box. If the ninth character is not specified, the screen region within the box is not painted.

DISPCOUNT() → *nDispCount*

Returns the number of DISPEND() calls required to restore the original display context.

DISPEND() → *NIL*

Informs the CA-Clipper display output system that the application has finished performing a series of display operations.

DISPOUT(<*exp*>, [<*cColorString*>]) → *NIL*

Writes the value of <*exp*> to the display at the current cursor position and can be used only within a procedure or function.

DOSERROR([<*nNewOsCode*>]) → *nOsCode*

Returns the number of the last DOS error and is set to the current one if the operation has an associated DOS error. <*nNewOsCode*> specifies the value returned by DOSERROR().

DOW(<*dDate*>) → *nDay*

Returns the day of the week from <*dDate*> as a numeric value between zero and seven.

DTOC(<*dDate*>) → *cDate*

Converts <*dDate*> to a character string in the current SET DATE and SET CENTURY format.

DTOS(<*dDate*>) → *cDate*

Converts <*dDate*> to a character string in the form *yyyymmdd* in order to INDEX ON a date and a character expression.

EMPTY(<*exp*>) → *lEmpty*

Returns true (.T.) if the result of <*exp*> is blank and determines if a user entered a value into a Get object before committing changes to a database file. It can also determine whether a formal parameter is NIL or unsupplied. In addition, it can test an array for zero-length.

EOF() → *lBoundary*

Returns true (.T.) if end of file is encountered in the current work area.

ERRORBLOCK([<*bErrorHandler*>]) → *bCurrentErrorHandler*

Defines a code block to execute when a runtime error occurs.

ERRORLEVEL([<*nNewReturnCode*>]) → *nCurrentReturnCode*

Returns the current DOS error level setting as a numeric value and optionally sets the current DOS error level to <*nNewReturnCode*>.

EVAL(<*bBlock*>, [<*BlockArg list*>]) → *LastBlockValue*

Evaluates the specified code block, optionally passing arguments, and returning the value of the last expression within the block.

EXP(<*nExponent*>) → *nAntilogarithm*

Returns *e****x*, where *e* is the base of natural logarithms and *x* is the <*nExponent*> argument.

FCLOSE(<*nHandle*>) → *lError*

Closes the binary file that corresponds to the file handle specified by <*nHandle*>, and returns false (.F.) if there is a write error.

FCOUNT() → *nFields*

Returns the number of fields in the database file of the current work area.

FCREATE(<*cFile*>, [<*nAttribute*>]) → *nHandle*

Creates a new binary file specified by <*cFile*> and returns the file handle as a numeric value.

FERASE(<*cFile*>) → *nSuccess*

Deletes <*cFile*> from disk.

FERROR() → *nErrorCode*

Returns the DOS error from the last binary file operation as a numeric value.

FIELDBLOCK(<*cFieldName*>) → *bFieldBlock*

Returns a code block that can assign a new value to or retrieve the current value of <*cFieldName*>. Note that <*cFieldName*> is not required to exist when the code block is created, but must exist before the code block is executed.

FIELDGET(<*nField*>) → *ValueField*

Retrieves the value of the field whose ordinal position in the current database structure is designated by <*nField*>.

FIELDNAME/FIELD(<*nPosition*>) → *cFieldName*

Returns the field name corresponding to <*nPosition*> in the current database file structure as a character string.

FIELDPOS(<*cFieldName*>) → *nFieldPos*

Given the database field name <*cFieldName*>, returns the numeric position of the field in the database file structure.

FIELDPUT(<*nField*>, <*expAssign*>) → *ValueAssigned*

Sets the field whose ordinal position in the current database structure is designated by <*nField*> to the value indicated by <*expAssign*>.

FIELDWBLOCK(<*cFieldName*>, <*nWorkArea*>) → *bFieldWBlock*

Returns a code block that can assign a new value to or retrieve the current value of <*cFieldName*> in the work area designated by <*nWorkArea*>. Note that <*cFieldName*> is not required to exist when the code block is created, but must exist before the code block is executed.

FILE(<*cFilespec*>) → *lExists*

Returns true (.T.) if the filename specified by <*cFilespec*> exists in the CA-Clipper SET DEFAULT drive and directory or anywhere in the SET PATH list.

FKLABEL(<*nFunctionKey*>) → *cKeyLabel

Returns the name of the specified function key number as a character string.

FKMAX() → *nFunctionKeys

Returns the maximum number of function keys (i.e., 40) that can be used with SET FUNCTION.

FLOCK() → *lSuccess*

Places a file lock on the current shared database file which includes open memo and index files in the same work area. The function returns true (.T.) if the lock succeeds.

FOPEN(<*cFile*>, [<*nMode*>]) → *nHandle*

Opens the binary file specified by <*cFile*> with an open mode of <*nMode*>, and returns the file handle as a numeric value.

FOUND() → *lSuccess*

Returns true (.T.) if the last search (i.e., SEEK or LOCATE) was successful.

FREAD(<*nHandle*>, @<*cBufferVar*>, <*nBytes*>) → *nBytes*

Reads characters from the binary file specified by <*nHandle*> into <*cBufferVar*> beginning at the current file pointer position, and returns the number of bytes successfully read.

FREADSTR(<*nHandle*>, <*nBytes*>) → *cString*

Reads <*nBytes*> characters from the binary file specified by <*nHandle*> beginning at the current file pointer position, and returns the value read as a character string.

FRENAME(<*cOldFile*>, <*cNewFile*>) → *nSuccess*

Changes the name of <*cOldFile*> to <*cNewFile*> and is identical to the RENAME command.

FSEEK(<*nHandle*>, <*nOffset*>, [<*nOrigin*>]) → *nPosition*

Moves the file pointer in the binary file specified by <*nHandle*> to a new position offset <*nOffset*> bytes from the position defined by <*nOrigin*>. The new file pointer position is then returned as a numeric value.

FWRITE(<*nHandle*>, <*cBuffer*>, [<*nBytes*>]) → *nBytesWritten*

Writes <*nBytes*> bytes from <*cBuffer*> to a binary file specified by <*nHandle*>, and returns the number of bytes written.

GBMPDISP(<*aBmpArray*> | <*cFile*>, <*nX*>, <*nY*>, [<*nTransColor*>]) → *NIL*

Displays a bitmap (.BMP) or icon (.ICO) file previously loaded into memory or display a BMP directly from disk. This function can be used only if you have set the screen to a graphic mode using SET VIDEOMODE. GBMPDISP() respects the constraints defined by GSETCLIP().

GBMPLOAD(<*cFileName*>) → *aBmpArray*

Allows you to load one or more .BMP or .ICO files into memory without having to display them. It avoids having to load the .BMP file each time you display it.

GELLIPSE(<*nXc*>, <*nYc*>, <*nRadiusX*>, <*nRadiusY*>, [<*nDegStart*>], [<*nDegEnd*>], [<*nStyle*>], [<*nColor*>], [<*nMode*>], [<*nOutLineColor*>], [<*nHeight3D*>]) → *NIL*

Draws an ellipse or circle and respects the constraints defined by GSETCLIP().

GETACTIVE([<*oGet*>]) → *oGet*

Provides access to the active Get object during a READ.

GETAPPLYKEY(<*oGet*>, <*nKey*>, <*GetList*>, <*oMenu*>, <*nMsgRow*>, <*nMsgLeft*>, <*nMsgRight*>, <*cMsgcolor*>) → *NIL*

Applies a key to a Get object from within a GET reader. If the key supplied to GETAPPLYKEY() is a SET KEY, GETAPPLYKEY() executes the set key and return; the key is not applied to the Get object.

GETDOSETKEY(<*bKeyBlock*>, <*oGet*>) → *NIL*

Processes SET KEY code block during GET editing. The procedure name and line number passed to the SET KEY block are based on the most recent call to READMODAL().

GETENV(<*cEnvironmentVariable*>) → *cString*

Returns the contents of a DOS environmental variable specified by <*cEnvironmentVariable*> as a character string.

GETPOSTVALIDATE(<*oGet*>) → *lSuccess*

Postvalidates the current Get object and evaluates Get:postBlock (the VALID clause) if present. The return value indicates whether the GET has been postvalidated successfully.

GETPREVALIDATE(<*oGet*>) → *lSuccess*

Prevalidates a Get object and evaluates Get:preBlock (the WHEN clause) if it is present. The logical return value indicates whether the GET has been prevalidated successfully.

GETREADER(<*oGet*>, <*GetList*>, <*oMenu*>, <*nMsgRow*>, <*nMsgLeft*>, <*nMsgRight*>, <*cMsgcolor*>) → *NIL*

Executes standard READ behavior for GETs. By default, READMODAL() uses the GETREADER() function to read Get objects. GETREADER() in turn uses other functions in Getsys.prg to do the work of reading the Get object.

GFNTERASE(<*aFont*>) → *NIL*

Erases a specified font from memory.

GFNTLOAD(<*cFontFile*>) → *aFont*

Loads a font file into memory and returns a pointer to the VMM region where the font is loaded.

GFNTSET(<*aFont*>, [<*nClipTop*>, <*nClipBottom*>]) → *aClipInfo*

Sets an already loaded font as active. When GFNTSET() is called, all subsequent DEVOUT() calls will use the new font. This means that you can use multiple .FND fonts at the same time.

GFRAME(<*nXStart*>, <*nYStart*>, <*nXEnd*>, <*nYEnd*>, <*nColorBack*>, <*nColorLight*>, <*nColorDark*>, <*nWidthUp*>, <*nWidthRight*>, <*nWidthDown*>, <*nWidthLeft*>, [<*nMode*>], [<*nStyle*>]) → *NIL*

Draws box frames with a 3-D look using three appropriately selected colors and can be used only if you have set the screen to a graphic mode using GMODE(). This function respects the constraints defined by GSETCLIP().

GGETPIXEL(<*nX*>, <*nY*>) → *nColor*

Gets the color information for a specific pixel.

GLINE(<*nXStart*>, <*nYStart*>, <*nXEnd*>, <*nYEnd*>, [<*nColor*>], [<*nMode*>]) → NIL

Draws lines on the screen and respects the constraints defined by GSETCLIP(). This function can be used only if you have set the screen to a graphic mode using SET VIDEOMODE.

GMODE([<*nMode*>]) → *aOldState*

Changes the video mode or retrieves information about the current video mode.

GPOLYGON(<*aVertex*>, <*lFilled*>, <*nMode*>, <*nOutLine*>, <*nFillColor*>) → *NIL*

Creates a polygon if you pass an array of coordinates which make up the polygon.

GPUTPIXEL(<*nX*>, <*nY*>, <*nColor*>, <*nMode*>) → *NIL*

Draws and changes the color of a specific pixel on the screen.

GRECT(<*nXStart*>, <*nYStart*>, <*nXEnd*>, <*nYEnd*>, [<*nStyle*>], [<*nColor*>], [<*nMode*>]) → *NIL*

Draws filled or empty rectangles on the screen and can be used only if you have set the screen to a graphic mode using GMODE(). It respects the constraints defined by GSETCLIP().

GSETCLIP([<*nX1*>, <*nY1*>, <*nX2*>, <*nY2*>]) → *aOldRegion*

Limits the active display to a portion of the screen and applies only to CA-Clipper graphic mode functions beginning with "G."

GSETEXCL([<*nExclArea*>] | [<*nTop*>, <*nLeft*>, <*nBottom*>, <*nRight*>, <*nType*>]) → NIL

Prevents output from being displayed in a defined region of the screen.

GSETPAL(<*nColor*>, <*nRedValue*>, <*nGreenValue*>, <*nBlueValue*>) → *aOldPalette*

GSETPAL() → *aOldPalette*

GSETPAL(<*aPalette*>) → *aOldPalette*

GSETPAL(<*nColor*>, <*aRGB*>) → *aOldPalette*

Changes a color's basic component values and must be called for each color if you want to change the palette.

GWRITEAT(<*nColStart*>, <*nLnStart*>, <*cString*>, [<*nColor*>], [<*nMode*>], [<*aFont*>]) → *nWidth*

Displays text in graphic mode without affecting the background and allows you to display text pixel by pixel as GWRITEAT() receives graphic coordinates only.

HARDCR(<*cString*>) → *cConvertedString*

Replaces all soft carriage returns in <*cString*> with hard carriage returns, and returns the result as a character string.

HEADER() → *nBytes*

Returns the number of bytes in the current database file header.

I2BIN(<*nInteger*>) → *cBinaryInteger*

Converts a CA-Clipper numeric to a character string formatted as a 16-bit integer.

IF(<*lCondition*>, <*expTrue*>, <*expFalse*>) → *Value*

Returns the result of <*expTrue*> if <*lCondition*> is true (.T.), or the result of <*expFalse*> if <*lCondition*> is false (.F.).

[I]IF(<*lCondition*>, <*expTrue*>, <*expFalse*>) → *Value*

Returns the result of <*expTrue*> if <*lCondition*> is true (.T.), or the result of <*expFalse*> if <*lCondition*> is false (.F.).

INDEXEXT() → *cExtension*

Returns the default index extension based on the database driver associated with the current work area.

INDEXKEY(<*nOrder*>) → *cKeyExp*

Returns the key expression of an index specified by its USE...INDEX or SET INDEX list position, <*nOrder*>, as a character string.

INDEXORD() → *nOrder*

Indicates which index file is the controlling index by returning its USE...INDEX or SET INDEX list position as a numeric value.

INKEY([<*nSeconds*>] [, <*nEventMask*>]) → *nInkeyCode*

Extracts a key from the keyboard buffer, or a mouse event and returns its INKEY() code as a numeric value representing the appropriate event.

INT(<*nExp*>) → *nInteger*

Converts <*nExp*> to an integer value by truncating all digits to the right of the decimal point.

ISALPHA(<*cString*>) → *lBoolean*

Determines if the specified string begins with an alphabetic character that consists of any uppercase or lowercase letter from A to Z. This function returns false (.F.) if the string begins with a digit or any other character.

ISCOLOR() | ISCOLOUR() → *lBoolean*

Returns true (.T.) if a color display is installed and false (.F) if a monochrome display is installed.

ISDIGIT(<*cString*>) → *lBoolean*

Determines whether the first character in a string is a numeric digit between zero and nine. If any other character is the first character of the <*cString*>, ISDIGIT() returns false (.F.).

ISDISK(<*cDrive*>) → *lAvailable*

Checks if a disk drive is available.

ISLOWER(<*cString*>) → *lBoolean*

Returns true (.T.) if the first character in <*cString*> is a lowercase letter. It is the inverse of ISUPPER() which determines whether a character begins with an uppercase character.

ISPRINTER() → *lReady*

Returns true (.T.) if the LPT1 print device is ready. This function is hardware-dependent and, therefore, only works on IBM BIOS compatible systems.

ISUPPER(<*cString*>) → *lBoolean*

Returns true (.T.) if the first character in <*cString*> is an uppercase letter.

L2BIN(<*nExp*>) → *cBinaryInteger*

Converts a CA-Clipper numeric to a character string formatted as a 32-bit signed integer.

LASTKEY() → *nInkeyCode*

Returns the INKEY() code of the last key extracted from the keyboard or the last mouse event.

LASTREC() | *RECCOUNT() → *nRecords*

Returns the number of records in the current work area.

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

Returns <*nCount*> characters from the left of <*cString*>.

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

Returns the number of characters in <*cString*> or the number of elements in <*aTarget*>.

LOG(<*nExp*>) → *nNaturalLog*

Returns the natural, base *e*, logarithm of <*nExp*> as a numeric value.

LOWER(<*cString*>) → *cLowerString*

Converts <*cString*> to lowercase.

LTRIM(<*cString*>) → *cTrimString*

Returns <*cString*> with the leading spaces removed.

LUPDATE() → *dModification*

Returns the date the current database file was last modified and CLOSEd.

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

Returns the greater of two numeric or date values.

MAXCOL() → *nColumn*

Returns the column number of the rightmost visible screen column.

MAXROW() → *nRow*

Returns the row number of the bottommost visible screen row.

MCOL() → *nCurrentMouseColumn*

Determines the mouse cursor's screen column position. This is useful when implementing a hit testing routine, which determines if the mouse cursor is on pertinent information when the left mouse button is pressed.

MDBLCLK([<*nNewSpeed*>]) → *nSpeed*

Determines and optionally changes the mouse's double-click speed threshold. This is useful when the mouse's double-click sensitivity needs to be adjusted.

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

Displays and edits memo fields and character strings and returns the edited string. MEMOEDIT() is a wait state function.

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

Returns a formatted line of text from *<cString>*.

MEMOREAD(<*cFile*>) → *cString*

Returns the contents of a disk file specified by *<cFile>* as a character string.

MEMORY(<*nExp*>) → *nKbytes*

Returns the number of Kbytes of available free pool memory.

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

Sets a RDD inheritance chain for the DBFMEMO database driver.

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

Replaces all hard and soft carriage returns in *<cString>* and returns the result as a character string. If not specified, *<cReplaceHardCR>* is assumed to be a semicolon (;) and *<cReplaceSoftCR>* is assumed to be a space(" ").

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

Writes *<cString>* to a disk file named *<cFile>*, and returns true (.T.) if it is successful.

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

Returns a code block that can assign a new value to or retrieve the current value of *<cMemvarName>*.

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

Activates a top bar menu and responds only to menu actions.

MHIDE() → *NIL*

Hides the mouse pointer and should be used in conjunction with MSHOW() when updating the screen.

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

Returns the smaller of two numeric or date values.

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

Returns the number of lines in a character string or memo field.

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

Determines the byte position that corresponds with a particular line and column within the <*cText*>. Note that <*nLine*> is one-relative while <*nCol*> is zero-relative. This is compatible with MEMOEDIT(). The return value is one-relative, compatible with AT(), RAT(), and other string functions.

MLEFTDOWN() →*lIsPressed*

Determines the press status of the left mouse button.

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

Returns the position of line <*nLine*> in <*cString*> with a line width of <*nLineLength*>.

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

Emulates the dBASE III PLUS MOD() function using the CA-Clipper modulus operator (%).

MONTH(<*dDate*>) → *nMonth*

Returns the month from <*dDate*> as a numeric value.

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

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

MPRESENT() → *lIsPresent*

Determines if a mouse is present.

MRESTSTATE(<*cSaveState*>) → *NIL*

Re-establishes the previous state of a mouse.

MRIGHTDOWN() → *lIsPressed*

Determines the status of the right mouse button.

MROW() → *nCurrentMouseRow*

Determines a mouse cursor's screen row position.

MSAVESTATE() → *cSaveState*

Saves the current state of a mouse.

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

Sets screen boundaries for the mouse cursor.

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

Controls mouse pointer movements and 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.

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

Determines a mouse cursor's visibility.

MSETPOS(<*nRow*>, <*nCol*>) → *NIL*

Sets a new position for the mouse cursor.

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

MSHOW([<*nCursorShape*>])
→ *nOldCursorShape*

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

Displays the mouse pointer and is generally used without parameters to simply redisplay the mouse pointer at the position where MHIDE() hid it.

MSTATE() → *aState* | *0*

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.

NETERR([<*lNewError*>]) → *lError*

Returns true (.T.) if a USE, USE...EXCLUSIVE, USE...SHARED, APPEND BLANK, FLOCK(), or RLOCK() fails in a network environment. <*lNewError*> specifies the value returned by NETERR().

NETNAME() → *cWorkstationName*

Returns the current workstation identification as a character string.

NEXTKEY() → *nInkeyCode*

Returns the INKEY() code of the next key pending in the keyboard or next mouse event as a numeric value.

NOSNOW(<*lToggle*>) → *NIL*

Toggles snow suppression. If <*lToggle*> is true (.T.), NOSNOW() enables snow suppression; otherwise, snow suppression is disabled.

ORDBAGEXT() → *cBagExt*

Returns the default order bag RDD extension of the current or aliased work area.

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

Returns a character string, the order bag name of a specific order.

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

Specifies conditions for ordering—i.e., creates a new order (using the ORDCREATE() function) or rebuilds an existing order (using the ORDREBUILD() function).

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

Sets the condition and scope for an order. ORDCONDSET() allows you to specify conditions and scoping rules that records must meet in order to be included in the next order created.

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

Creates an order in an order bag and creates orders in RDDs that recognize multiple-order bags.

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

Returns and optionally changes the descending flag of an order.

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

Removes a specified order from multiple-order bags.

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

Returns the FOR expression of an order.

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

Returns the status of the unique flag for a given order.

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

Returns the key expression of a specified order.

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

Adds a key to a custom-built order. ORDKEYADD() evaluates the key expression and then adds the key for the current record to the order.

**ORDKEYCOUNT([<*cOrder*> | <*nPosition*>],
[<*cIndexFile*>]) → *nKeys***

Counts the keys in the specified order and returns the result as a numeric value.

**ORDKEYDEL([<*cOrder*> | <*nPosition*>],
[<*cIndexFile*>],
[<*expKeyValue*>]) → *lSuccess***

Deletes a key from a custom-built order. ORDKEYDEL() evaluates the key expression, and then deletes the key for the current record from the order.

ORDKEYGOTO(<*nKeyNo*>) → *lSuccess*

Moves to a record specified by its logical record number in the controlling order.

**ORDKEYNO([<*cOrder*> | <*nPosition*>],
[<*cIndexFile*>]) → *nKeyNo***

Returns the logical record number of a key in an order.

ORDKEYVAL() → *uKeyValue*

Gets the key value of the current record from the controlling order.

**ORDLISTADD(<*cOrderBagName*>
[, <*cOrderName*>]) → *NIL***

Adds the contents of an order bag, or a single order in an order bag to the order list.

ORDLISTCLEAR() → *NIL*

Removes all orders from the order list for the current or aliased work area.

ORDLISTREBUILD() → *NIL*

Rebuilds all orders in the order list of the current work area.

ORDNAME(<*nOrder*>[,<cOrderBagName>])
→ *cOrderName*

Returns the name of an order in the order list.

ORDNUMBER(<*cOrderName*>
[, <*cOrderBagName*>]) → *nOrderNo*

Returns the position of an order in the current order list.

ORDSCOPE(<*nScope*>,[<*expNewValue*>])
→ *uCurrentValue*

Sets or clears the boundaries for scoping key values in the controlling order.

ORDSETFOCUS([<c*OrderName*> | <*nOrder*>]
[, <*cOrderBagName*>])
→ *cPrevOrderNameInFocus*

Returns the order name of the previous controlling order and optionally sets the focus to an new order in the order list.

ORDSETRELATION(<*nArea*> | <*cAlias*>,
<*bKey*>, [<*cKey*>]) → *NIL*

Relates a specified work area (the child work area) to the current work area (the parent work area).

ORDSKIPUNIQUE([<*nDirection*>]) → *lSuccess*

Moves the record pointer to the next or previous unique key in the controlling order.

OS() → *cOsName*

Returns the name and version number of the operating system the current workstation is operating under as a character string.

OUTERR(<*exp list*>) → *NIL*

Writes the values in *<exp list>* to the standard error device.

OUTSTD(<*exp list*>) → *NIL*

Writes the values in *<exp list>* to the standard output device.

PADL(<*exp*>, <*nLength*>,
[<*cFillChar*>]) → *cPaddedString*
PADC(<*exp*>, <*nLength*>,
[<*cFillChar*>]) → *cPaddedString*
PADR(<*exp*>, <*nLength*>,
[<*cFillChar*>]) → *cPaddedString*

Pads *<exp>* with a leading and/or trailing fill character until the resulting character string has a length of *<nLength>*.

PCOL() → *nColumn*

Returns the current column position of the printhead as a numeric value.

PCOUNT() → *nLastArgumentPos*

Returns the position of the last argument passed in the list of arguments specified by the calling procedure or user-defined function. Note that this includes arguments skipped within the list.

PROCLINE([<*nActivation*>]) → *nSourceLine*

Queries the CA-Clipper activation stack to determine the last line executed in a currently executing procedure, user-defined function, or code block. A line number is relative to the beginning of the original program (.prg) file.

PROCNAME([<*nActivation*>])
→ *cProcedureName*

Queries the CA-Clipper activation stack and returns the name of a currently executing procedure, user-defined function, or code block as a character string.

PROW() → *nRow*

Returns the current row position of the printhead as a numeric value.

QOUT([<*exp list*>]) → *NIL*
QQOUT([<*exp list*>]) → *NIL*

Displays a list of expressions to the console.

RAT(<*cSearch*>, <*cTarget*>) → *nPosition*

Returns the starting position of the last instance of <*cSearch*> in <*cTarget*> as a numeric value.

RDDLIST([<*nRDDType*>]) → *aRDDList*

Returns a one-dimensional array that lists the available Replaceable Database Drivers (RDDs). Returns all available RDDs, regardless of type, if <*nRDDType*> is not supplied.

RDDNAME() → *cRDDName*

Returns the name of the RDD active in the current or specified work area. A work area other than the currently active work area can be specified by aliasing the function.

RDDSETDEFAULT([<*cNewDefaultRDD*>])
→ *cPreviousDefaultRDD*

Sets or returns the 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.

READEXIT([<*lToggle*>]) → *lCurrentState*

Determines whether or not Up arrow and Down arrow can be used as READ exit keys, and returns their current status as a logical value.

READFORMAT([<*bFormat*>]) → *bCurrentFormat*

Returns, and optionally sets, the code block that implements a format (.fmt) file.

READINSERT([<*lToggle*>]) → *lCurrentMode*

Determines whether READ and MEMOEDIT() will be invoked in insert or overstrike mode, and returns the current mode as a logical value.

READKEY() → *nReadkeyCode

Determines what keystroke or mouse event terminated the last READ.

READKILL([<*lKillRead*>]) → *lCurrentSetting*

Returns, and optionally sets, whether the current READ should be exited. Unless directly manipulated, READKILL() returns true (.T.) after a CLEAR GETS is issued for the current READ; otherwise, it returns false (.F.).

READMODAL(<*aGetList*>, [<*nGet*>],
[<*oMenu*>], [<*nMsgRow*>, <*nMsgLeft*>,
<*nMsgRight*>, <*cMsgColor*>]) → *lUpdated*

Activates editing mode for the specified <*GetList*> array.

READUPDATED([<*lChanged*>])
→ *lCurrentSetting*

Determines whether any GET variables changed during a READ and optionally changes the READUPDATED() flag.

READVAR() → *cVarName*

Returns the name of the current @...GET or MENU TO variable as a character string.

RECCOUNT() | LASTREC() → *nRecords

Returns the number of records in the current work area.

RECNO() → *Identity*

Returns the number of the current record in the current work area as a numeric value.

RECSIZE() → *nBytes*

Returns the record length of the current database file as a numeric value.

REPLICATE(<*cString*>, <*nCount*>) → *cRepeatedString*

Returns <*cString*> concatenated with itself <*nCount*> times.

RESTSCREEN([<*nTop*>], [<*nLeft*>], [<*nBottom*>], [<*nRight*>], <*cScreen*>) → *NIL*

Displays a screen previously saved by SAVESCREEN() using the specified coordinates.

RIGHT(<*cString*>, <*nCount*>) → *cSubString*

Returns <*nCount*> characters from the right of <*cString*>.

RLOCK() → *lSuccess*

In a network environment, this function places a record lock on the current shared database file. The function returns true (.T.) if the lock succeeds.

ROUND(<*nNumber*>, <*nDecimals*>) → *nRounded*

Returns <*nNumber*> rounded to <*nDecimals*> decimal places as a numeric value. If <*nDecimals*> is a negative value, rounding occurs to the left of the decimal.

ROW() → *nRow*

Returns the row position of the screen cursor as a numeric value.

[R]TRIM(<*cString*>) → *cTrimString*

Returns a copy of <*cString*> with the trailing spaces removed.

SAVESCREEN([<*nTop*>], [<*nLeft*>], [<*nBottom*>], [<*nRight*>]) → *cScreen*

Saves the screen region defined by the specified coordinates to a character variable.

SCROLL([<*nTop*>], [<*nLeft*>], [<*nBottom*>], [<*nRight*>], [<*nVert*>], [<*nHoriz*>]) → *NIL*

Scrolls the screen region defined by the specified coordinates up or down <*nVert*> rows, right or left <*nHoriz*>. If <*nVert*> is zero or not specified, and <*nHoriz*> is also zero or not specified, the screen region is blanked out. If coordinate arguments are not specified, the dimensions of the visible display are used.

SECONDS() → *nSeconds*

Returns the number of seconds elapsed since 12:00 midnight according to the system clock.

SELECT([<*cAlias*>]) → *nWorkArea*

Returns the work area number of the alias specified by <*cAlias*>.

SET(<*nSpecifier*>, [<*expNewSetting*>], [<*lOpenMode*>]) → *CurrentSetting*

Inspects or changes a global setting, and returns the current setting.

SETBLINK([<*lToggle*>]) → *lCurrentSetting*

Determines whether the asterisk (*) character in SETCOLOR() strings causes blinking or background intensity as a display attribute, and returns the current setting of SETBLINK(). If SETBLINK() is true (.T.), the default, the asterisk (*) in color strings, causes the foreground characters of the display value to blink; a value of false (.F.) cause the background of the display value to be intensified.

SETCANCEL([<*lToggle*>]) → *lCurrentSetting*

Toggles program termination with Alt+C and Ctrl+Break on (.T.) or off (.F.), and returns the current setting as a logical value.

SETCOLOR([<*cColorString*>]) → *cColorString*

Returns the current color setting as a character string and optionally, sets a new color specified by <*cColorString*>.

SETCURSOR([<*nCursorShape*>])
→ *nCurrentSetting*

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.

SETKEY(<*nInkeyCode*>, [<*bAction*>])
→ *bCurrentAction*

Assigns an action block to a key to execute during a wait state.

SETMODE(<*nRows*>, <*nCols*>) → *lSuccess*

Changes the display mode from its current size to <*nRows*> by <*nCols*>. MAXROW() and MAXCOL() will subsequently return the new maximum display coordinates.

SETPOS(<*nRow*>, <*nCol*>) →*nRow*

Moves the cursor to a new position specified by <*nRow*> and <*nCol*>.

SETPRC(<*nRow*>, <*nCol*>) → *NIL*

Sets the internal PROW() and PCOL() values to <*nRow*> and <*nCol*>.

SOUNDEX(<*cString*>) → *cSoundexString*

Converts <*cString*> to its soundex character string form.

SPACE(<*nCount*>) → *cSpaces*

Returns a character string of <*nCount*> spaces.

SQRT(<*nNumber*>) → *nRoot*

Returns the square root of a positive number.

STR(<*nNumber*>, [<*nLength*>],
[<*nDecimals*>]) → *cNumber*

Converts <*nNumber*> to a character string of length <*nLength*> containing <*nDecimals*> decimal places.

STRTRAN(<*cString*>, <*cSearch*>, [<*cReplace*>],
[<*nStart*>], [<*nCount*>]) → *cNewString*

Searches <*cString*> for occurrences of <*cSearch*> replacing them with <*cReplace*> or a null ("") string if the argument is omitted. The function replaces all occurrences from the beginning of the string, unless limited by <*nStart*> and <*nCount*>, and returns an updated copy of the character string.

STUFF(<*cString*>, <*nStart*>, <*nDelete*>,
<*cInsert*>) → *cNewString*

Replaces <*nDelete*> characters in <*cString*> with <*cInsert*> beginning at position <*nStart*>.

SUBSTR(<*cString*>, <*nStart*>, [<*nCount*>]) → *cSubstring*

Returns <*nCount*> characters from <*cString*> starting at <*nStart*>. If <*nStart*> is positive, scanning begins from the left of <*cString*>; if negative, scanning begins from the right of <*cString*>.

TIME() → *cTimeString*

Returns the system time in the form *hh:mm:ss* as a character string.

TONE(<*nFrequency*>, <*nDuration*>) → *NIL*

Sounds a speaker tone for a frequency of <*nFrequency*> cycles per second, with a duration of <*nDuration*> specified in increments of 1/18 of a second.

TRANSFORM(<*exp*>, <*cSayPicture*>) → *cFormatString*

Converts <*exp*> to a character string formatted using the PICTURE specified by <*cSayPicture*>.

TRIM(<cString>) → *cTrimString*

Removes trailing spaces from a character string. This is typically the case with database fields which are stored in fixed-width format.

TYPE(<*cExp*>) → *cType*

Returns the data type of an expression as a character string. The expression must be specified in the form of a character string.

UPDATED() → *lChange*

Returns true (.T.) if the last READ changed data in any of the associated Get objects.

UPPER(<*cString*>) → *cUpperString*

Converts <*cString*> to uppercase.

USED() → *lDbfOpen*

Returns true (.T.) if a database file is in USE in the current work area.

VAL(<*cNumber*>) → *nNumber*

Converts <*cNumber*> to a numeric value.

VALTYPE(<*exp*>) → *cType*

Returns a single character representing the data type returned by <*exp*>.

VERSION() → *cVersion*

Returns the CA-Clipper version number of the CA-Clipper library, EXTEND.LIB, as a character value.

WORD(<*nNumber*>) → *NIL

Converts <*nNumber*>, a CA-Clipper numeric, from double to integer values. This function is used solely for passing numeric parameters with CALL.

YEAR(<*dDate*>) → *nYear*

Returns the year from <*dDate*> as a four-digit numeric value.

Chapter 6
Classes

Below is a summary of the CA-Clipper classes covered in the "Language Reference" chapter of the *Reference Guide, Volumes 1 and 2.*

CheckBox Class

Check boxes present a choice to the user which can be either on or off. When a check box is clicked, its state is toggled between *checked* (on) and *unchecked* (off). The CheckBox class has been designed to be easily integrated into the standard CA-Clipper GET/READ system in addition to providing the necessary functionality to be utilized on its own.

Class Function

CheckBox(*<nRow>*, *<nColumn>*, [,*<cCaption>*]) → *oCheckBox*

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

Instance Variables

bitmaps

Contains an array of exactly two elements that indicates the bitmap files to be displayed. The first element indicates the file name of the bitmap to be displayed when the CheckBox is selected. The second element indicates the file name of the bitmap to be displayed when the CheckBox is not selected.

buffer

Contains a logical value that indicates whether the check box is checked or unchecked. A value of true (.T.) indicates that it is checked and a value of false (.F.) indicates that it is not checked.

capCol (Assignable)

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

capRow (Assignable)

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

caption (Assignable)

Contains an optional character string that concisely describes the check box on the screen. If omitted, the default is an empty string. When present, the & character specifies that the character immediately following it in the caption is the check 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 the check box.

cargo **(Assignable)**

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

col **(Assignable)**

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

colorSpec **(Assignable)**

Contains a character string that indicates the color attributes that are used by the check box's display() method. The string must contain four color specifiers.

fBlock **(Assignable)**

Contains an optional code block that, when present, is evaluated each time the CheckBox object receives or loses input focus. The code block takes no implicit arguments. It is included in the CheckBox 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 CheckBox object has input focus. CheckBox:HasFocus contains true (.T.) if it has input focus; otherwise, it contains false (.F.).

message **(Assignable)**

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

row **(Assignable)**

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

sBlock **(Assignable)**

Contains an optional code block that, when present, is evaluated each time the CheckBox object's state changes. The code block takes no implicit arguments. It is included in the CheckBox 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 delimiter characters that are used by the check box's display() method. The string must contain four characters, the left delimiter, the checked indicator, the unchecked indicator, and the right delimiter.

typeOut

Contains the logical value false. CheckBox:typeOut never changes. It is not used by the CheckBox object and is only provided for compatibility with the other GUI control classes.

Methods

display() → *self*

display() is a method of the CheckBox class that is used for showing a check box and its caption on the screen. It uses the values of the following instance variables to correctly show the check box in its current context in addition to providing maximum flexibility in the manner a check box appears on the screen: buffer, caption, capCol, capRow, col, colorSpec, hasFocus, row, and style.

hitTest() → *nHitStatus*

hitTest() is a method of the CheckBox class that is used for determining if the mouse cursor is within the region of the screen that the check box or its caption occupies.

killFocus() → *self*

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

select() → *self*

select() is a method of the CheckBox class that is used for changing the state of a check box. Its state is typically changed when the space bar is pressed or the mouse's left button is pressed when its cursor is within the check box's region of the screen.

setFocus() → *self*

setFocus() is a method of the CheckBox class that is used for giving focus to a CheckBox object. Upon receiving this message, the CheckBox object redisplays itself and if present, evaluates the code block within its fBlock variable. This message is meaningful only when the CheckBox object does not have input focus.

Error Class

An Error object is an object that contains information pertaining to a runtime error. When a runtime error occurs, a new Error object is created and passed as an argument to the error handler block specified with the ERRORBLOCK() function. Within the error handler, the Error object can then be queried to determine the nature of the error condition.

Class Function

ErrorNew() → *oError*

Creates and returns a new Error object.

Instance Variables

args (Assignable)

Contains an array of the arguments supplied to an operator or function when an argument error occurs. For other types of errors, Error:args contains a NIL value.

canDefault (Assignable)

Contains true (.T.) if default error recovery is available for the error condition; otherwise, it contains false (.F.).

canRetry (Assignable)

Contains true (.T.) if the subsystem can retry the operation that caused the error condition. Error:canRetry never contains true (.T.) if Error:canSubstitute contains true (.T.).

canSubstitute (Assignable)

Contains true (.T.) if a new result can be substituted for the operation that produced the error condition. Error:canSubstitute never contains true (.T.) if either Error:canDefault or Error:canRetry contains true (.T.).

cargo (Assignable)

Contains a value of any data type provided as a user-definable slot.

description (Assignable)

Contains a character string describing the error condition. If Error:genCode is not zero, a description is always available.

filename (Assignable)

Contains a character value representing the name originally used to open the file associated with the error condition.

genCode (Assignable)

Contains an integer numeric value representing a CA-Clipper generic error code.

operation (Assignable)

Contains a character string describing the operation being attempted when the error occurred.

osCode (Assignable)

Contains an integer numeric value representing the operating system error code associated with the error condition.

severity (Assignable)

Contains a numeric value indicating the severity of the error condition. Four standard values—ES_WHOCHARES, ES_WARNING, ES_ERROR and ES_CATASTROPHIC—are defined in Error.ch.

subCode (Assignable)

Contains an integer numeric value representing a subsystem-specific error code.

subSystem (Assignable)

Contains a character string representing the name of the subsystem generating the error.

tries (Assignable)

Contains an integer numeric value representing the number of times the failed operation has been attempted.

Get Class

A Get object is a general purpose mechanism for editing data. It is used in CA-Clipper to implement the @...GET and READ commands. Get objects provide a sophisticated architecture for formatting and editing data, including cursor navigation and data validation. Data validation is performed via user-supplied code blocks, and display formatting can be controlled using standard picture strings.

Class Function

GetNew([<*nRow*>], [<*nCol*>], [<*bBlock*>], [<*cVarName*>], [<*cPicture*>], [<*cColorSpec*>]) → *oGet*

Returns a new Get object with the row, col, block, picture, and colorSpec instance variables set from the supplied arguments.

Instance Variables

badDate

Contains true (.T.) when the Get object is a date type and the date represented by the contents of the editing buffer is invalid. Contains false (.F.) when the date is valid or the Get is not editing a date value.

block (Assignable)

Contains the code block used to set or get the value of the GET variable. The code block takes an optional argument that should assign the value of the argument to the variable. If the argument is not specified, the code block should return the current value of the variable.

buffer (Assignable)

Contains a character value that defines the editing buffer used by the Get object. Contains a value other than NIL if the Get object has input focus.

cargo (Assignable)

Contains a value of any data and is provided as a user-definable slot.

changed

Contains true (.T.) if the buffer has changed since the Get object received input focus.

clear (Assignable)

Contains a logical value indicating that the editing buffer should be cleared before any more values are entered.

col (Assignable)

Contains a numeric value that defines the column position of the Get object display.

colorSpec (Assignable)

Contains a character string defining the display attributes for the Get object. The string must contain two color specifiers: the *unselected* color determines the color of the Get object when it does not have input focus the *selected* color determines the color of the Get object when it has input focus.

decPos

Contains a numeric value indicating the decimal point position within the editing buffer.

exitState (Assignable)

Contains a numeric value used in the CA-Clipper version of Getsys.prg to record the means by which a Get object was exited. Manifest constants for the Get:exitState values can be found in Getexit.ch.

hasFocus

Contains true (.T.) if the Get object has input focus.

message (Assignable)

Contains a character string that is displayed on the Get system's status bar line when the GET has input focus. Typically, it describes the anticipated contents of, or user response to, the GET.

minus **(Assignable)**

Contains a logical value indicating that a minus sign (-) has been added to the editing buffer.

name **(Assignable)**

Contains a character string representing the name of the GET variable.

original

Contains the value of the GET variable at the time the Get object acquired input focus.

picture **(Assignable)**

Contains a character value defining the PICTURE string used to control formatting and editing for the Get object.

pos

Contains a numeric value indicating the position of the cursor within the editing buffer. If the Get object does not have input focus, it contains NIL.

postBlock **(Assignable)**

Contains a code block used to validate a newly entered value. The Get:postBlock should contain an expression that evaluates to true (.T.) for a legal value and false (.F.) for an illegal value.

preBlock **(Assignable)**

Contains a code block that determines whether editing of the current Get object editing buffer should be permitted. The Get:preBlock should evaluate to true (.T.) if the cursor is allowed to enter the editing buffer; otherwise, it should evaluate to false (.F.).

reader

Contains a code block to implement special READ behaviors for any GET. If Get:reader contains a code block, READMODAL() evaluates that block in order to READ the GET (the Get object is passed as an argument to the block). The block may in turn call any desired function to provide custom editing of the Get object. If Get:reader does not contain a code block, READMODAL() uses a default read procedure (GetReader()) for the Get object.

rejected

Contains true (.T.) if the last character specified either by a Get:insert or a Get:overStrike message was rejected.

row **(Assignable)**

Contains a numeric value defining the row position of the Get object display.

subscript

Contains an array of numeric values representing the subscripts of a Get array element.

type

Contains a single character representing the data type of the GET variable.

typeOut

Contains true (.T.) if the most recent message attempted to move the cursor out of the editing buffer or there are no editable positions in the buffer.

Methods

assign() → *self*

Assigns the value in the editing buffer to the Get variable by evaluating Get:block with the buffer value supplied as its argument. This message is meaningful only when the Get object has input focus.

backSpace() → *self*

Deletes the character to the left of the cursor and moves the cursor one character to the left.

colorDisp([<*cColorString*>]) → *self*

Changes the color of a Get object and then redisplays it. This method is exactly equivalent to assigning Get:colorSpec and issuing Get:display.

delete() → *self*

Deletes the character under the cursor.

delEnd() → *self*

Deletes from the current character position to the end of the GET, inclusive.

delLeft() →*self*

Deletes the character to the left of the cursor.

delRight() → *self*

Deletes the character to the right of the cursor.

delWordLeft() → *self*

Deletes the word to the left of the cursor.

delWordRight() → *self*

Deletes the word to the right of the cursor.

display() → *self*

Displays the contents of the Get:buffer on the screen if the Get object has input focus; otherwise, the Get:block is evaluated and the result displays.

end() → *self*

Moves the cursor to the rightmost editable position within the editing buffer.

hitTest(<*nRow*>, <*nColumn*>) → *self*

Determines the screen position specified by <*nRow*> and <*nColumn*> is on the Get object.

home() → *self*

Moves the cursor to the leftmost editable position within the editing buffer.

insert(<*cChar*>) → *self*

Inserts <*cChar*> into the editing buffer at the current cursor position, shifting the existing contents of the buffer to the right.

killFocus() → *self*

Takes input focus away from the Get object.

left() → *self*

Moves the cursor left to the nearest editable position within the editing buffer.

overStrike(<*cChar*>) → *self*

Puts <*cChar*> into the editing buffer at the current cursor position, overwriting the existing contents of the buffer.

reset() → *self*

Resets the Get object's internal state information when the Get object has input focus.

right() → *self*

Moves the cursor right to the nearest editable position within the editing buffer.

setFocus() → *self*

Gives input focus to the Get object, creates and initializes the object's internal state information, and displays the contents of the editing buffer to the screen.

toDecPos() → *self*

Moves the cursor to the immediate right of the decimal point position in the editing buffer if it contains a numeric value.

undo() → *self*

Sets the value of the Get variable to the value of the Get:original variable, and redisplays the Get editing buffer to reflect the change.

unTransform() → *xValue*

Converts the character value in the editing buffer back to the data type of the original variable.

updateBuffer() → *self*

Sets the editing buffer to reflect the current value of the Get variable, and then redisplays the Get.

varGet() → *GetVarValue*

Returns the current value of the Get variable. For simple Get variables, this is equivalent to executing Get:block. However, if the Get variable is an array element, EVAL(aGet:block) will not return the value of the Get variable; in this case use of varGet() is required.

varPut() → *Value*

Sets the GET variable to the passed value. For simple GET variables, this is equivalent to executing Get:block with an argument. However, if the GET variable is an array element, EVAL(aGet:block, aValue) will not set the value of the GET variable; in this case use of varPut() is required.

wordLeft() → *self*

Moves the cursor one word to the left within the editing buffer.

wordRight() → *self*

Moves the cursor one word to the right within the editing buffer.

ListBox Class

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*

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

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.

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.

cargo **(Assignable)**

Contains a value of any type that is ignored by the ListBox object and 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.

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.

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. It 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.

isOpen

Contains a logical value that indicates whether the list is visible. A value of true (.T.) indicates that the list is visible; 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().

Methods

addItem() → *self*

Appends 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.

close() → *self*

Restores the screen under a drop-down list box.

delItem ()→ *self*

Removes an item from a list. ListBox:buffer is automatically adjusted when an item is deleted while the last item in the list is selected.

display() → *self*

Shows a list and its caption on the screen and 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.

findText() → *nPosition*

Determines 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.

getData() →*expValue*

Retrieves the data portion of a list box item.

getItem () → *aItem*

Retrieves a list box item.

getText() → *cText*

Retrieves the text portion of a list box item.

hitTest() → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that the list box occupies.

insItem () → *self*

Inserts a new item into 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.

killFocus() → *self*

Takes 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.

nextItem() → *self*

Changes 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.

open() → *self*

Saves the screen under a drop-down list box and displaying the list.

prevItem() → *self*

Changes 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.

scroll () → *self*

Scrolls the contents of a list box up or down.

select() → *self*

Changes 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.

setData → *self*

Changes the data that is associated with a list item.

setFocus() → *self*

Gives 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.

setItem () → *self*

Replaces an item in a list.

setText () → *self*

Changes the text that is associated with a list item.

MenuItem Class

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*

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

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. The default is an empty string.

shortcut (Assignable)

Contains an optional numeric inkey value that indicates the key that activates the menu selection. The default is 0. Shortcut keys are available only for menu items on a pop-up menu.

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.

Methods

isPopUp() → *lPopUpStatus*

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.

PopUpMenu Class

Places items on the top bar menu or another pop up menu.

Class Function

PopUp([*<nTop>*] [*<nLeft>*] [*<nBottom>*] [*<nRight>*]) → *oPopUp*

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

Instance Variables

border **(Assignable)**

Contains an optional string that is used when drawing a border around the pop-up menu. Its default value is a B_SINGLE + SEPARATOR_SINGLE. The string must contain either zero or exactly eleven characters.

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.

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.

Methods

addItem → ***self***

Appends a new item to a pop-up menu.

close → ***self***

Deactivates a pop-up menu. It determines whether or not its selected menu item contains a PopUpMenu object, restores the previous contents of the region of the screen that it occupies and lastly, it sets its selected item to 0.

delItem → ***self***

Removes an item from a pop-up menu.

display() → ***self***

Shows a pop-up menu including its items on the screen. display() uses the values of instance variables like MenuItem:checked, PopUpMenu:bottom, PopUpMenu:current, 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.

getAccel()→ ***nPosition***

Determines whether or not a key press should be interpreted as a user request to evoke the data variable of a particular pop-up menu item.

getFirst() → ***nPosition***

Determines the position of the first selectable item in a pop-up menu.

getItem()→ ***oMenuItem***

Accesses a MenuItem object after it has been added to a pop-up menu.

getLast() → ***nPosition***

Determines the position of the last selectable item in a pop-up menu.

getNext() → ***nPosition***

Determines 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.

getPrev() → *nPosition*

Determines 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.

getShortct→ *nPosition*

Determines whether a keystroke should be interpreted as a user request to select a particular pop-up menu item.

hitTest → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that the pop-up menu occupies.

insItem() → *self*

Inserts a new item to a pop-up menu.

isOpen() → *lIsOpen*

Determines 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.

open() → *self*

Activates 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.

select() → *self*

Changes 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.

setItem() → *self*

Replaces 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.

PushButton Class

Places a push button at the indicated position on the screen.

Class Function

PushButton(*<nRow>*, *<nColumn>*, [*<cCaption>*]) → *oPushButton*

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

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.

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 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.

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.

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. It 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. It 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.

typeOut

Contains the logical value false. PushButton:typeOut never changes. It is not used by the PushButton object and is only provided for compatibility with the other GUI control classes.

Methods

display() → *self*

Shows 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.

hitTest() → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that the push button occupies.

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.

select() → *self*

Activates a push button whose 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.

setFocus() → *self*

Gives 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.

RadioButto Class

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 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*

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.

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 column 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 exactly seven color specifiers.

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.

Methods

display() → *self*

Shows 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.

hitTest() → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that the radio button occupies. It 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.

isAccel() → *lHotKeyStatus*

Determines whether a key press should be interpreted as a user request to select a radio button.

killFocus() → *self*

Takes 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.

select() → *self*

Changes 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.

setFocus() → *self*

Gives focus to a RadioButto object. Upon receiving this message, the RadioButto object re displays 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.

RadioGroup Class

The RadioGroup class provides a convenient mechanism for manipulating radio buttons.

Class Function

RadioGroup(*<nTop>*, *<nLeft>*, *<nBottom>*, *<nRight>*) → *oRadioGroup*

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.

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 exactly three color specifiers.

fBlock (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.

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.

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.

Methods

addItem() → *self*

Appends a new radio button to a radio group.

delItem() → *self*

Removes a radio button from a radio group.

display() → *self*

Shows its radio buttons on the screen. It accomplishes this by calling the Display() method of each of the radio buttons in its group.

getAccel() → *nPosition*

Determines whether or not a key press should be interpreted as a user request to select a particular radio button. It accomplishes this by calling the GetAccel() method of each of the radio buttons in its group.

getItem() →*oRadioButto*

Retrieves a radio button from a radio group.

hitTest() → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that any of the RadioButto objects contained within the radio group occupies. It accomplishes this by calling the hitTest() method of each of the radio buttons in its group.

insItem() → *self*

Inserts a new radio button into a radio group.

killFocus() → *self*

Takes 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.

nextItem() → *self*

Changes 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.

prevItem() → *self*

Changes 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.

select() → *self*

Changes 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.

setColor() → *self*

Uniformly sets the color attributes of all the radio buttons in its group. It accomplishes this by setting the colorSpec instance variable of each of the radio buttons in its group to the value specified by *<cColorString>*.

setFocus() → *self*

Gives 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.

setStyle() → *self*

Uniformly sets the Style attribute of all the radio buttons in its group. It accomplishes this by setting the Style of each of the radio buttons in its group to the value specified by *<cStyle>*.

Scrollbar Class

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*

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

Instance Variables

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.

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.

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.

current (Assignable)

Contains a numeric value that indicates the number of the current item that the scroll bar refers to.

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.

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. It 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 previous arrow character, the client area character, the thumb character, and the next arrow character.

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 that the scroll bar refers to.

Methods

display() → *self*

Shows 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.

hitTest() → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that the scroll bar occupies.

update() → *self*

Changes 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.

TBColumn Class

A TBColumn object is a simple object containing the information needed to fully define one column in a TBrowse object (see the section on the TBrowse class). TBColumn objects have no methods, only exported instance variables.

Class Function

TBColumnNew(*<cHeading>*, *<bBlock>*) → *oTBColumn*

Returns a new TBColumn object with the specified heading and data retrieval block.

Instance Variables

block **(Assignable)**

Contains a code block used to retrieve data for the column. Any code block is valid and no block arguments are supplied when the block is evaluated.

cargo **(Assignable)**

Contains a value of any data type provided as a user-definable slot.

colorBlock **(Assignable)**

Contains an optional code block that determines the color of data items as they are displayed. This block is executed each time a new value is retrieved via the TBColumn:block and must return an array containing four numeric values that are indexes into the color table of the TBrowse object.

colSep **(Assignable)**

Contains an optional character string used to draw 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.

defColor **(Assignable)**

Contains an array of four numeric values used as indexes into the color table in the TBrowse object to determine the colors of the general TBrowse display, and the current browse cell, respectively. 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 used to draw a horizontal line between the data values and the footing for this data column. If it does not contain a character value, the 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 used tc draw 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.

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.

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.).

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: 1) the length of the heading, 2) the length of the footing, and 3) the length of the data at the first evaluation of TBColumn:block. The width of the displayed data will be the length at the first evaluation of TBColumn:block.

Methods

setstyle → *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.

TBrowse Class

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 is performed via user-supplied code blocks, allowing nearly unlimited flexibility and control. Display formatting is controlled using formatting strings.

Class Functions

TBrowseNew(*<nTop>*, *<nLeft>*, *<nBottom>*, *<nRight>*) → *oTBrowse*

Returns a new TBrowse object with the specified coordinate settings, but with no columns and no code blocks for data source positioning.

TBrowseDB(*<nTop>*, *<nLeft>*, *<nBottom>*, *<nRight>*) → *oTBrowse*

Returns a new TBrowse object with the specified coordinate settings and default skip blocks for data source positioning within database files, but no column objects. The default code blocks execute the GO TOP, GO BOTTOM, and SKIP operations.

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.

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. The default is the current SETCOLOR() value when the TBrowse object is created.

colPos (Assignable)

Contains a numeric value indicating the data column where the browse cursor is currently located.

colSep (Assignable)

Contains a character value that defines a column separator for TBColumn objects that do not contain a column separator of their own.

footSep (Assignable)

Contains a character value that defines a footing separator for TBColumns not containing a footing separator of their own.

freeze (Assignable)

Contains a numeric value that defines the number of data columns to lock on the left side of the display.

goBottomBlock (Assignable)

Contains a code block evaluated in response to the TBrowse:goBottom() message in order to position 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 evaluated in response to the TBrowse:goTop() message in order to position 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 tha executes a GO TOP command.

headSep (Assignable)

Contains a character value that defines a heading separator for TBColumns not containing a heading separator of their own.

hitBottom (Assignable)

Contains true (.T.) if an attempt was made tc move past the end of available data; 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 true (.T.) if an attempt was made tc navigate past the beginning of the available data; 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.

nColPos (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.

nMessage (Assignable)

Contains a character string that is displayed on the GET system's status bar when the TBrowse is utilized within the GET system. Typically, it describes the anticipated contents of, or user response to, the browse.

nRowPos (Assignable)

Contains a numeric value indicating the data row where the mouse cursor is currently located.

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 that defines the total number of data rows displayable not including headings, footings, or separators.

rowPos (Assignable)

Contains a numeric value indicating the data row where the browse cursor is currently located. Data rows are numbered from one, starting with the topmost data row.

skipBlock (Assignable)

Contains a code block used to reposition the data source. During stabilization, this code block is executed with a numeric argument representing the number of records to be skipped. A positive value means skip forward and a negative value means skip backward.

stable (Assignable)

Contains true (.T.) if the TBrowse object is stable; otherwise, it contains false (.F.). 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.).

Methods

addColumn (<*oColumn*>) → *self*

Adds a new TBColumn object to the TBrowse object and increases TBrowse:colCount by one.

applyKey() → *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.

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.

colWidth(<*nColumn*>) → *nWidth*

Returns the display width of column number <*nColumn*> as known to the browse. If <*nColumn*> is out of bounds or not supplied or not a number, the method returns zero.

configure() → *self*

Causes a TBrowse object to re-examine all instance variables and TBColumn objects, and then reconfigure its internal settings as required.

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*

Deletes a column from a browse. The return value is a reference to the column object being deleted.

down() → *self*

Moves the cursor down one row. 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 to true (.T.).

end() → *self*

Moves the browse cursor to the rightmost data column currently visible.

forceStable()

Performs a full stabilization of the TBrowse.

getColumn(<*nColumn*>) → *oColumn*

Returns the TBColumn object specified by <*nColumn*>.

goBottom() → *self*

Repositions the data source to logical bottom of file by evaluating the TBrowse:goBottomBlock, displays the last window full of rows, and moves the browse cursor to the bottom row.

goTop() → *self*

Repositions the data source to the logical beginning of file by evaluating the TBrowse:goTopBlock, displays the first window full of rows, and moves the browse cursor to the top row.

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() → *nHittest*

Determines if the screen position specified by *<nRow>* and *<nColumn>* is on the TBrowse object.

home() → *self*

Moves the cursor to the leftmost unfrozen column on the display.

insColumn(*<nPos>*, *<oColumn>*) → *oColumn*

Inserts a column object into the middle of a browse. The return value is a reference to the column object being inserted.

invalidate() → *self*

Causes the next stabilization of the TBrowse object to redraw the entire TBrowse display, including headings, footings, and all data rows. 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.

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*

Moves the browse cursor down one window full of rows. 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 to true (.T.).

pageUp() → *self*

Moves the cursor up one window full of rows. 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 to 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 to the left without changing the column of the browse cursor, if possible.

panRight() → *self*

Pans the display to the right without changing the column of the browse cursor, if possible.

refreshAll() → *self*

Internally marks all data rows as invalid, causing them to be refilled and redisplayed during the next stabilize loop.

refreshCurrent() → *self*

Internally marks the current data row as invalid, causing it to be refilled and redisplayed during the next stabilize loop.

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 is brought into view.

setColumn()→ *oColumnCurrent*

Replaces the TBColumn object indicated by <*nColumn*> with the TBColumn object specified by <*oColumnNew*>. The value returned is the current TBColumn object.

setKey() → *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.

stabilize() → *lStable*

Performs incremental stabilization. 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 stabilize 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.

up() → *self*

Moves the browse cursor up one row. 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 to true (.T.).

TopBarMenu Class

The top bar menu is used as a main menu in which pop-up menus reside.

Class Function

TopBar(<*nRow*>, <*nLeft*>, <*nRight*>) → *oTopBar*

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 exactly six color specifiers.

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.

Methods

addItem() → *self*

Appends a new item to a top bar menu.

delItem(nPosition>) → *self*

Removes an item from a top bar menu. When an item is deleted, the items which follow it are shifted left.

display() → *self*

Shows 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.

getAccel() → *nPosition*

Determines whether a key press should be interpreted as a user request to evoke the data variable of a particular top bar menu item.

getFirst() → *nPosition*

Determines 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.

getItem() → *self*

Accesses a MenuItem object after it has been added to a top bar menu.

getLast() → *nPosition*

Determines 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.

getNext() → *nPosition*

Determines the position of the next selectable item in a top bar menu. and 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.

getPrev() → *nPosition*

Determines the position of the previous selectable item in a top bar menu. It 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.

hitTest() → *nHitStatus*

Determines if the mouse cursor is within the region of the screen that the top bar occupies.

insItem() → *self*

Inserts a new item to a top bar menu.

select() → *self*

Changes 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.

setItem() → *self*

Replaces 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.

Chapter 7
Command Line Utilities

The following CA-Clipper command line utilities are covered in this chapter:

- CA-Clipper Compiler—CLIPPER.EXE

  A summary of the CA-Clipper Compiler options covered in the "CA-Clipper Compiler" chapter of the *Programming and Utilities Guide.*

- CA-Clipper Real Mode Linker—BLINKER.EXE

  A summary of the *real mode* linker syntax and options covered in the "CA-Clipper Real Mode Linker—BLINKER.EXE" chapter of the *Programming and Utilities Guide.*

- CA-Clipper Protected Mode Linker—EXOSPACE.EXE

  A summary of the *protected mode* linker syntax and options covered in the "CA-Clipper Protected Mode Linker—EXOSPACE.EXE" chapter of the *Programming and Utilities Guide.*

- CA-Clipper Program Maintenance Utility—RMAKE.EXE

  A summary of the RMAKE syntax and options is covered in the "Program Maintenance" chapter of the *Programming and Utilities Guide.*

CA-Clipper Compiler (CLIPPER.EXE)

Compiles one or more program (.prg) files containing procedures and user-defined functions to form an object (.OBJ) file.

Syntax

CLIPPER [*<sourceFile>* | *@<scriptFile>* [*<option list>*])

Command Line Arguments

This is the command line syntax to invoke the CA-Clipper compiler. If issued without any arguments, a help screen is displayed.

<sourceFile> is the name of the program (.prg) file to compile to an object file. If no extension is specified, (.prg) is assumed. The filename can optionally include a drive designator and a path reference.

<scriptFile> is the name of an ASCII text file containing a list of source files to compile into a single object file. The default extension for this file is (.clp).

<option list> is a list of one or more options to control the course of the compilation, all of which are described below. To get a list of options, specify the compiler command line with no arguments.

Compiler Options

All options are shown in uppercase preceded by a slash(/). However, options are not case sensitive. You can replace the slash with a dash (-) if you prefer.

Some compiler options have arguments. If an option has arguments, specify them after the option, with no space between the option and its first argument.

/A

Declares any variable included in a PRIVATE, PUBLIC, or PARAMETERS statement as MEMVAR.

/B

Includes debugging information in the object file.

/D*<identifier>*[=*<text>*]

Defines an identifier to the preprocessor with *<text>* assigned to the *<identifier>* if specified.

/ES[*<severityLevel>*]

Specifies the severity level of warnings. Compatible with CA-Clipper 5.0x.

/ES0

Does not set the DOS ERRORLEVEL upon exit if warnings are encountered and if <severityLevel> is 0 or omitted. This is the default behavior of the compiler.

/ES1

Sets DOS levels upon exit if warnings are encountered during compilation but still generates an .OBJ file.

/ES2

Does not generate object file (.OBJ) and sets DOS errorlevel upon exit if warnings are encountered during compilation.

/I<pathName>

Adds the specified directory to the front of the INCLUDE path list. Multiple /I options may be specified in the same compiler session to specify several header file search directories.

/L

Excludes the program source code line numbers from the object file.

/M

Compiles only the current program (.prg) file suppressing automatic search for program (.prg) files referenced in a program file with the DO, SET FORMAT, and SET PROCEDURE commands.

/N

Suppresses the automatic definition of a procedure with the same name as the program (.prg) file.

/O<objFile>

Defines the name and/or location of the output object file.

/P

Preprocesses the program (.prg) file and copying the result to an output file with a (.ppo) extension.

/Q

Suppresses line numbers from displaying while compiling.

/R[<libFile>]

Embeds a library search request in the object file. If /R is specified without *<libFile>* name, the default requests for CLIPPER.LIB, EXTEND.LIB, TERMINAL.LIB, and DBFNTX.LIB are suppressed. Multiple /R options may be specified in the same compiler session to embed more than one library request.

/S

Checks the syntax of the current (.prg) file and no object file is generated.

/T<pathName>

Specifies a different directory for temporary files generated during compilation.

/U[<userStandardHeaderFile>]

Identifies an alternate standard header file to preprocess in place of the supplied STD.CH which is used automatically.

/W

Generates warning messages for undeclared or unaliased (ambiguous) variable references.

/Z

Suppresses shortcutting optimizations on the logical operators .AND. and .OR.. This option is provided as an aid to isolating code that depends on the behavior of older versions of CA-Clipper.

CA-Clipper Real Mode Linker (BLINKER.EXE)

CA-Clipper 5.3 contains a limited version of the full Blinker product. This real mode CA-Clipper 5.3 version of Blinker features high-speed linking, dynamic overlaying, caching of overlay, compression of the symbol table, and "burning" the CLIPPER environment variable.

Blinker Commands

Script file(s) are specified on the command line and default to an extension of .LNK. They may be nested, and more than one may be used on the command line, for example,

BLINKER @Defaults @Other LIB Mylib VERBOSE.

This command line instructs Blinker to read the script file DEFAULTS.LNK followed by the script file OTHER.LNK. It also instructs Blinker to use the library MYLIB.LIB and to display a VERBOSE listing of the .OBJ files being processed. The full names of the commands, or any unique abbreviation of the words which comprise the commands, may be used although it is recommended that at least three letters be used for clarity. For example, the command BLINKER OVERLAY OPSIZE 40 can be abbreviated to BLI OVE OPS 40.

Most Blinker commands may be specified in any order, at any point in the link script file. Typically the first FILE statement should specify the .OBJ containing the main program procedure.

 # ...

Indicates text as a script file comment. Any line in the script file beginning with the character # is treated as a comment and is ignored by Blinker. If the # ... character occurs within a line, the rest of the line is ignored.

// ...

Indicates text as a script file comment. Any line in the script file containing the characters // is treated as a comment and is ignored by Blinker. If the // ... characters occur within a line, the rest of the line is ignored.

@<scriptname>

Specify the name of a script file. Blinker supports nested script files of any size to a depth of 5 levels. The processing of nested script files is similar to a program function call.

BEGINAREA

Specifies the start of a dynamic overlay area to the linker. Each BEGINAREA should have a corresponding ENDAREA to indicate the end of the overlay area.

BLINKER CACHE EMS *<nuMax(%)>*
[, *<nuMinLeave(%)>*]

Specifies the amount of EMS (LIM version 4.0 and higher) memory to be used for the overlay cache. Two numeric parameters are used to specify the amount of EMS memory to be allocated and the amount to be left for program use. Both parameters may be specified as either a number of KB, or as a percentage of the total available EMS.

BLINKER CACHE XMS *<nuMax(%)>*
[, *<nuMinLeave(%)>*]

Specifies the amount of XMS (version 2.0 and higher) memory to be used for the overlay cache. Two numeric parameters are used to specify the amount of XMS memory to be allocated and the amount to be left for program use. Both parameters may be specified as either a number of KB, or as a percentage of the total available XMS.

BLINKER CLIPPER PAGE ON | OFF

Enables/Disables automatic CA-Clipper 5.3 code paging.

BLINKER ENVIRONMENT CLIPPER *<cName>*

Changes the name of the CA-Clipper environment variable from the default setting of "CLIPPER" to any string of up to 16 characters, and thus avoids the remote site settings for the CLIPPER environment variable.

BLINKER ENVIRONMENT NAME *<cName>*

Specifies the name of the environment variable for Blinker runtime options. By default, if the argument is not specified, the default setting is BLINKER.

BLINKER ENVIRONMENT OVERRIDE

Commands Blinker to allow the CA-Clipper environment variables set using the BLINKER EXECUTABLE CLIPPER command to be overridden at application run time by the settings of the CLIPPER environment variable. When doing this, you may want to consider renaming your CLIPPER environment variable with the command BLINKER ENVIRONMENT CLIPPER.

BLINKER EXECUTABLE CLIPPER *<cEnvironmentstring>*

Burns default values for the CA-Clipper runtime environment variables into the executable file. Those settings set using the BLINKER EXECUTABLE CLIPPER command take precedence over those set with the SET CLIPPER command.

BLINKER EXECUTABLE NODELETE

Creates .EXE regardless of unresolved externals, although the resulting .EXE is not guaranteed to operate correctly.

BLINKER LINK EMS ON | OFF

Controls whether or not EMS memory is used at link time by the virtual memory system within Blinker.

BLINKER LINK PAGEFRAME ON | OFF

Controls the use of the EMS pageframe by Blinker at link time by the virtual memory system within Blinker.

BLINKER LINK XMS ON | OFF

Controls whether or not XMS memory is used at link time by the virtual memory system within Blinker.

BLINKER MESSAGE DUPLICATES

Causes Blinker to generate the 1004 link-time warning message for all duplicate symbols within libraries encountered at link time.

BLINKER MESSAGE NOBLINK

Disables the blinking eyes which are displayed while Blinker is running.

BLINKER MESSAGE NOWARNING

Suppresses the display of Blinker link-time warning messages 1001 to 1100.

BLINKER MESSAGE WINK (LEFT)

Enables single eye winking during linking. If no parameter is entered, the right eye will wink (from Blinker's point of view, i.e., from inside the screen looking out). The parameter LEFT causes the left eye to wink.

BLINKER OVERLAY OPSIZE <*nuSize*>

Requests a size for the overlay pool at run time in KB. It can be set to any value between 12KB and 128KB. The default operating size is 40KB.

BLINKER OVERLAY PAGEFRAME ON | OFF

Allows the overlay manager to utilize the expanded memory pageframe for the overlay pool (maximum size 64Kb), on machines equipped with LIM EMS (3.2 or higher) expanded memory boards or emulators.

BLINKER OVERLAY THRESHOLD <*nuLimit*>

Sets the threshold size below which code specified to be overlaid will be moved back to the root section of the program.

BLINKER OVERLAY UMB ON | OFF

Used in conjunction with a suitable XMS driver it allows the Blinker overlay manager to execute overlays in upper memory blocks above 640KB.

BLINKER PROCEDURE DEPTH <*nuDepth*>

Specifies maximum depth of CA-Clipper procedure nesting. This depth is the number of procedure calls that are executed before a procedure return.

DEFINE <*symbol*>(, <*symbol*> ...)

Specifies symbols to be excluded at link time and points them to a dummy routine in the overlay manager.

DEFLIB

Uses default library search records to locate and link libraries which are specified within an object module.

ECHO (ON | OFF |) (<*cString*>)

ECHO ON <*cString*> turns on echoing of the script file to the screen, with the first line displayed being <*cString*> if it is specified. ECHO OFF turns off echoing to the screen of the script file and ignores the rest of the line.

ENDAREA

Specifies the end of an overlay area to the linker, as described in the BEGINAREA section.

EXTDICTIONARY

Specifies that Blinker should use the extended dictionary that is appended to library files by some librarian utility programs.

FILE (d:)(path)<*filename*>
(, (d:)(path)<*filename*> ...)

Specifies to the linker the names of one or more .OBJ files to be included in the output file at that point.

LIBRARY (d:)(path)<*libname*>
(, (d:)(path)<*libname*> ...)

Specifies one or more program libraries required by the program.

MAP (=(d:)(path)<*filename*>) (S)(,A)

Requests a segment map of the executable.

MIXCASE

Causes Blinker to treat all publics and externals as case sensitive at link time, rather than converting them to UPPERCASE.

MODULE <*module*> (,<*module*> ...)
(FROM <*libname*>)

Specifies placement of individual library modules regardless of whether the originating libraries or object modules are specified within an overlay area or not.

MURPHY

Causes the overlay manager to attempt to force any inadvertent overlay errors (programming errors) to manifest themselves during testing or debugging.

NOBELL

Suppresses the beep upon completion of link.

NODEFLIB

Ignores default library search records which are encountered within object modules.

NOEXTDICTIONARY

Disables processing of extended library dictionaries.

NOTABLEOFCONTENTS

Disables processing of library table of contents.

OUTPUT (drive:)(path)<*filename*>

Specifies the name of the output executable file to be created by the linker.

READONLY

Specifies executable file will be set to read-only status.

SEARCH (d:)(path)<*libname*>
(, (d:)(path)<*libname*> ...)

Prioritizes all symbols in the specified libraries.

SECTION INTO <*ovlname*>
(FILE (d:)(path)<*filename*>
(, (d:)(path)<*filename*>...))

Specifies files to be placed in an external overlay.

STACK <*nuSize*>

Specifies to Blinker in HEXADECIMAL the stack size in bytes required by the program .EXE file.

UPPERCASE

Specifies to the linker that all symbols should be converted to uppercase before being added to the symbol table.

VERBOSE (0 | 1 | 2)

Displays status information during linking.

WORKFILE (d:)(path)<*filename*>

Specifies drive, path, and file name of the temporary WORKFILE to use at link time if one is required.

Linker Function

nuValue := BLIVERNUM()

Returns the Blinker version number as an integer containing the version number of Blinker used to link the program.

CA-Clipper Protected Mode Linker (EXOSPACE.EXE)

Combines objects files (.OBJ) with library files (.LIB) to form a protected-mode executable file (.EXE).

Syntax

EXOSPACE [@*<lnkFile>*] [*<command list>*]

OR

EXOSPACE file[*<objFile>*]

Command Line Arguments

<lnkFile> is the name of an ASCII text file called a *script file* from which the linker takes some or all of its input. If you specify the file without an extension, .LNK is assumed. You can optionally specify a drive designator and/or a path reference for the script file.

<command list> is a list of linker commands separated by spaces. Linker commands on the command line either override or augment the same commands in a script file.

<objFile> is the file produced by the CA-Clipper compiler.

Linker Commands

This section provides a brief description of all the ExoSpace commands. You can specify commands on the ExoSpace command line or in a script file. The commands are not case-sensitive, and you can specify them in whatever order you want.

The only command that is required to link is the FILE command—all others are optional.

EXOSPACE CLIPPER 501

Specifies that you are linking a CA-Clipper 5.01 or 5.01a application.

EXOSPACE ENVIRONMENT CLIPPER *<envName>*

Specifies the environment variable to use for application runtime configuration.

EXOSPACE ENVIRONMENT OVERRIDE

Causes runtime configuration environment settings to override .EXE defaults.

EXOSPACE EXECUTABLE CLIPPER *<settings>*

Hard-codes default runtime configuration settings in the .EXE. See SET CLIPPER in the "Environment Variables" chapter of this guide for details about available settings and syntax. In addition to the settings listed there, the following additional settings are available for controlling the ExoSpace protected-mode VM system:

LOWMEM:*<nKBytes>*—Sets the amount of low DOS memory to reserve for allocation by third-party libraries for interrupt routines. The default is 0. Ignore this unless your third-party library documentation specifically recommends that you set it.

MAXMEM:*<nKBytes>*—Sets the maximum amount of physical memory to use before swapping to disk. The default is 8096 KB, which is ample for most applications. In a task switching environment, you may want to lower this value.

MINMEM:<*nKBytes*>—Sets the minimum amount of physical memory that must be available to run the program. If there is not enough memory available, an error message is generated and the application returns to DOS. The default is 1024 KB, which is the recommended minimum.

VMSIZE:<*nKBytes*>—Sets the amount of virtual memory the VM system should provide. The default is 8096 KB, which is ample for most applications.

EXOSPACE EXECUTABLE NODELETE

Creates an .EXE even if errors occur during linking.

EXOSPACE PACKAGE <*package list*>

Causes specified protected-mode support packages to be included in the .EXE. The following packages are available—use one or more in a comma-separated list:

8514—Specifies the IBM 8514 Display Adapter. Allows direct calls from protected mode, with no mode switching.

DOS25—Specifies the Absolute Disk Read/Write compatibility package. Services INT 25h and 26h.

INT10—Specifies the Video BIOS compatibility package. Services INT 10h.

IPX—Specifies the IPX/SPX compatibility package for Novell Networks. Services INT 7Ah.

IPXCT—Specifies the compatibility package for CA-Clipper Tools.

NET5C—Specifies the NETBIOS compatibility package. Replaces NETBIOS?.OBJ. Services INT 5Ch.

NOVM—Disables the ExoSpace VM system. The resulting application will run only on systems that have sufficient RAM to contain the application's entire .EXE and all its data.

EXOSPACE PROCEDURE DEPTH <*maxDepth*>

Sets the stack size of an application in terms of maximum procedure call depth.

FILE <*objFile list*>

Specifies one or more object files to be linked in a comma-separated list.

LIBRARY <*libFile list*>

Specifies one or more library files to search during linking in a comma-separated list.

MAP [=<*mapFile*>] [<*mapOption list*>]

Causes a map to be generated. By default, the map file will have the same name as the resulting .EXE file. The available map options are as follows—use one or more in a comma-separated list:

A—Public symbols sorted by address

N—Public symbols sorted by name

S—Segments with assigned addresses

MODULE <*moduleName*> FROM <*libName*>

Searches for named module in specified library only.

NODEFLIB | NODEFAULTLIBRARYSEARCH

Does not search default libraries.

OUTPUT <*exeFile*>

Specifies the name of the executable file to be generated. If not specified, the first filename encountered on the ExoSpace command line (either object or script) is used, but with an .EXE extension.

STACK <*sizeBytes*>

Sets the stack size of an application in decimal bytes. The value you specify can be any positive number up to 65,535.

Program Maintenance (RMAKE.EXE)

Automates the maintenance of multi-file program systems by keeping files up to date. It does this by comparing the date and time stamps of files related to one another and performs a series of actions if the date and time stamps do not match.

Syntax

RMAKE [<*makeFile list*>] [<*macroDef list*>] [<*option list*>]

Command Line Arguments

This is the command line syntax to invoke the CA-Clipper make utility. If issued without any arguments, a help screen is displayed.

<*makeFile list*> is a list of one or more make files to process. The default extension for this file is (.rmk).

<*macroDef list*> is a list of one or more macro definitions of the form <*macroName*>=[<*value*>].

<*option list*> is a list of one or more options to control the course of the make, all of which are described below. To get a list of options, specify the RMAKE command line with no arguments.

RMAKE Options

All options are shown in uppercase preceded by a slash (/). However, options are not case sensitive. You can replace the slash with a dash (-) if you prefer.

Some compiler options have arguments. If an option has arguments, specify them after the option, with no space between the option and its first argument.

/B

Displays debugging information.

/D<*macroName*>[:<*value*>]

Defines a macro and an optional value. If the value is not supplied, the macro is defined with a null value.

/F

Forces RMAKE to rebuild all modules whether or not they have been updated.

/I

Ignores execution errors.

/N

Performs a null make displaying the commands that would be executed without actually executing them.

/Q

Suppresses the RMAKE copyright message.

/U

Enables the # character as a comment indicator and suppresses its use as a directive indicator.

/W

Displays certain warning messages.

/XS<numSymbols>

Sets the size of the internal symbol table. If not specified, the default is 500 symbols.

/XW<numBytes>

Sets the size of the internal workspace. If not specified, the default is 2048 bytes.

Comments

Comments may be specified using C-style inline comments (e.g. /* .. */) or C-style line comments (//) or, if the /U option is used, Unix make-style comments (#). Comments cannot be nested.

Dependency Rules

A dependency rule causes certain actions to be performed if any of the dependent files have a newer date and time stamp than the target file:

<targetFile>: <dependentFile list>
    [*<action>*]
    [*<action>*]...

Inference Rules

An inference rule is used to complete a dependency rule specified with no actions. It defines the actions to take for any incomplete dependency rule that satisfies certain filename extensions:

.<dependentExtension>.<targetExtension>:
    [*<action>*]
    [*<action>*]...

Macro Definitions

Macros are defined in make files using the following syntax:

***<macroName>* = [*<value>*]**

Once defined, a macro can be referred to using the following syntax:

$(*<macroName>*)

To assign the contents of a macro to another macro, use the following syntax:

***<macroName>* := $(*<macroName>*)**

Makepath Macros

Makepath macros are special macros that define where RMAKE searches and creates classes of files as defined by extension. Makepath macros are defined using the following syntax:

makepath[.<*extension*>] = <*pathSpec*>

Predefined Macros

In addition to user-defined macros, there are several predefined macros that can be used to access dependency and target filenames.

$*

Expands to the target filename without a path or extension.

$@

Expands to the target filename including path and extension.

$**

Expands to the complete list of full dependency filenames.

$<

Expands to the full name of the first file in the dependent file list.

$?

Expands to a list of dependencies that have a more recent date and time stamp than the target file.

Directives

RMAKE provides directives to control the operation of the make process. Directives cannot be used as action lines. Macros encountered in a directive are expanded immediately. By default, either the # character or the ! character can be used to specify a directive. If the /U option is used, the # character is interpreted as a comment indicator and cannot be used to specify directives.

#! <*action*>

Executes a DOS command.

#else

Executes statements between the #else and the next #endif directive when the corresponding #if directive evaluates to false (.F.).

#end[if]

Terminates the block of statements defined by a #if directive.

#ifdef <*macroName*>

Executes statements between the #ifdef directive and the corresponding #else or #endif directives if the specified <*macroName*> exists.

#ifeq <*word1*> <*word2*>

Executes statements between the #ifeq directive and the corresponding #else or #endif directives if <*word1*> and <*word2*> are identical. A word consists of one or more characters up to the next white space character. To include a white space character within a word, enclose the word in double quote (" ") marks.

#iffile <*fileSpec*>

Executes statements between the #iffile directive and the corresponding #else or #endif directives if any files match the specified <*fileSpec*>.

#ifndef <*macroName*>

Executes statements between the #ifndef directive and the corresponding #else or #endif directives if the specified <*macroName*> does *not* exist.

#include "<*fileName*>"

Inserts and processes the contents of <*fileName*> before continuing with the current make file.

#stderr "<*text*>"

Writes the <*text*> to the standard error file o device.

#stdout "<*text*>"

Writes the <*text*> to the standard output file or device.

#undef <*macroName*>

Removes any previous definition of the specified <*macroName*>.

Chapter 8
Menu Utilities

The following CA-Clipper menu utilities are covered in this chapter:

- Program Editor—PE.EXE
- Database Utility—DBU.EXE
- Report and Label Utility—RL.EXE
- The Guide To CA-Clipper—NG.EXE

Program Reference

PE [*<filename>*]

PE is the CA-Clipper Program Editor. With it you can create and edit simple program files. PE.EXE is located in \CLIP53\BIN and the source files are in \CLIP53\SOURCE\PE.

<filename> is the name of the text file to edit or create. The default extension is (.prg). If this is not specified, a new file is created.

DBU [/*<color option>*] [*<filename>*] /E

DBU is the Database Utility. With it you can create and modify database structures as well as perform other database operations. DBU.EXE is located in \CLIP53\BIN and the source files are in \CLIP53\SOURCE\DBU.

<color option> is /C for color and /M for monochrome.

<filename> is the name of a view (.vew) file previously created in DBU or a database (.dbf) file . This argument causes DBU to open and browse the specified file; otherwise, the main screen is active where you can open files and access the DBU menu bar.

/E opens the file EXCLUSIVE. This switch is optional and not case sensitive.

RL

RL is the Report and Label Utility. With it, you can create and modify report (.frm) and label (.lbl) files to use with the REPORT FORM and LABEL FORM commands. RL.EXE is located in \CLIP53\BIN and the source files are in \CLIP53\SOURCE\RL.

NG [*<command line>*]

NG is the Norton Instant Access Engine used to display online documentation databases. Supplied with the Instant Access Engine is *The Guide To CA—Clipper*, the online documentation for CA-Clipper. Both files are located in the \CLIP53\NG directory.

<command line> is any valid DOS command, including arguments. If this is specified, NG is loaded in pass through mode; otherwise, it is loaded in memory-resident mode.

Chapter 9
The CA-Clipper Debugger

The command line syntax used to invoke the CA-Clipper debugger is covered in this section as are the menu commands that you can use while debugging your program. This information is a summary of the "CA-Clipper Debugger" chapter found in the *Programming and Utilities Guide*.

The CA-Clipper Debugger (CLD.LIB)

Allows you to debug your source code while your executable file is running.

Syntax

CLD [[/43 | /50 | /S] [@*<scriptFile>*]
**   *<exeFile>* [*<argument list>*]]**

Command Line Arguments

This is the command line syntax to invoke the debugger. If issued without any arguments, a brief help screen is displayed.

/43 specifies 43-line mode and is available on EGA monitors only.

/50 specifies 50-line mode and is available on EGA and VGA monitors only.

/S is available on EGA and VGA monitors only. This option splits the screen between your application and the debugger, allowing you to view the application and the debugger simultaneously. On a VGA monitor /S uses 50-line mode, and on an EGA monitor it uses 43-line mode.

In split screen mode, the top 25 lines of the screen are used by your application and the remaining lines are used for the debugger display.

<scriptFile> is the name of a script file with a default extension of (.cld). CLD searches for the specified <*scriptFile*> in the current directory and then searches the DOS PATH. A script file is simply an ASCII text file containing one or more debugger commands, with each command appearing on a separate line. When the debugger is invoked with a script file, each command in the file is executed automatically after the <*exeFile*> file is loaded.

In addition to any script file called for on the CLD command line, the debugger automatically searches for a script file with the name *Init.cld*. If a file by this name is located in the current directory or anywhere in the DOS PATH, the debugger executes it as a script file. If both Init.cld and a command line script file are present, Init.cld is executed first followed by the command line script file.

<exeFile> is the name of the executable file (.EXE) you want to debug. CLD searches for the *<exeFile>* in the current directory only—the DOS PATH is not searched. If this file has not been compiled using the /B compiler option to embed debugging information, debugging is not possible. The file must include line numbers and debugging information.

<argument list> is the argument list for *<exeFile>*. There must be a space between *<argument list>* and *<exeFile>*.

Menu Commands

Listed below is a summary of debugger commands from the "CA-Clipper Debugger" chapter of the *Programming and Utilities Guide*.

? | ?? <exp>

Displays the value of an expression.

BP ((At) <lineNum> ((In) <idProgramFile>)))
BP *<idFunction>* | *<idProcedure>*

Sets or removes a breakpoint at a specified program and line number or at a particular function/procedure call.

Callstack [on | Off]

Toggles the display of the Callstack window.

Delete All [BP | TP | WP]
Delete BP | TP | WP *<number>*

Deletes breakpoints, watchpoints, and tracepoints, either individually or as a whole.

File DOS

Loads a temporary copy of COMMAND.COM, allowing you to enter DOS commands without leaving the current application.

File Exit

Terminates the debugger, closes all files, and returns to DOS.

File Open *<idFileName>*

Opens the specified file for viewing in the Code window.

File Resume

Returns from viewing a file to the program being debugged.

Help [Keys | Windows | Menus | Commands]

Activates the Help window.

List BP | TP | WP

Lists watchpoints, tracepoints, and breakpoints.

Locate Case

Toggles search case sensitivity setting.

Locate Find *<searchString>*

Searches the current file for the specified character string, obeying the Locate Case setting.

Locate Goto *<lineNum>*

Moves the cursor to a specified iine in the Code window.

Locate Next

Locates the next occurrence of the character string specified by the last Locate Find command.

Locate Previous

Locates the previous occurrence of the character string specified by the last Locate Find command, obeying the Locate Case setting.

Monitor All

Toggles the display of variables of all storage classes in the Monitor window.

Monitor Local

Toggles the display of local variables in the Monitor window.

Monitor Private

Toggles the display of private variables in the Monitor window.

Monitor Public

Toggles the display of public variables in the Monitor window.

Monitor Sort

Controls the order in which variables are displayed in the Monitor window.

Monitor Static

Toggles the display of static variables in the Monitor window.

Num [On | off]

Toggles the display of line numbers at the beginning of each line in the Code window.

Options Codeblock

Controls the tracing of code blocks during single step mode.

Options Color

Opens the Set Colors window.

Options Exchange

Controls the display of program output while in animate mode.

Options Line

Toggles the display of line numbers at the beginning of each line in the Code window.

Options Menu

Toggles the debugger menu bar display.

Options Mono

Switches the debugger display mode between color and monochrome.

Options Path <*idPathList*>

Defines the search path for source files.

Options Preprocessed

Toggles the display of preprocessed code in the Code window.

Options Restore <*idScriptFile*>

Reads commands from a script file.

Options Save <idScriptFile>

Saves the current debugger settings to a script file.

Options Swap

Controls the display of the application screen when input is required.

Options Tab <*tabSize*>

Sets the tab size for the Code window.

Point Breakpoint

Sets or removes a breakpoint at the current cursor position.

Point Delete <*number*>

Deletes a tracepoint or watchpoint setting.

Point Tracepoint <*exp*>

Specifies an expression as a tracepoint.

Point Watchpoint <*exp*>

Specifies an expression as a watchpoint.

Run Animate

Executes the application in animate mode.

Run Go

Executes the application in run mode.

Run Next

Executes the application until line zero of the next activation is encountered. This is equivalent to creating a tracepoint of "PROCLINE() == 0."

Run Restart

Reloads the current application in preparation for re-execution.

Run Speed <*delay*>

Sets the step delay for animate mode.

Run Step

Executes the current program in single step mode.

Run To

Executes the current program up to the current cursor position.

Run Trace

Executes the current program in trace mode.

View <*idFileName*>

Opens the specified file for viewing in the Code window.

View App

Displays program output.

View Callstack

Toggles the display of the Callstack window.

View Sets

Displays the View Sets window.

View Workareas

Displays the View Workareas window.

Window Iconize

Toggles the active debugger window between icon and window display modes.

Window Move

Moves the active debugger window.

Window Next

Selects the next debugger window.

Window Prev

Selects the previous debugger window.

Window Size

Changes the size of the active debugger window.

Window Tile

Restores the debugger windows to their default size and location.

Window Zoom

Toggles the active debugger window between window and full-screen display modes.

Chapter 10
Environment Variables

Listed below are all of the environment variables used by the various utilities in the CA-Clipper package. For each variable, the complete DOS SET syntax is given which can be specified at the DOS prompt or in a batch file such as AUTOEXEC.BAT.

Variable Reference

SET CLIPPER=
[//BADCACHE]
[//CGACURS]
[//DYNF:<*nHandles*>]
[//E:<*nExpandedKbytes*>]
[//F:<*nHandles*>]
[//INFO]
[//NOIDLE]
[//SWAPK:<*nBytes*>]
[//SWAPPATH:'<*path*>']
[//TEMPPATH:'<*path*>']
[//X:<*nKbytes*>]

Specifies configuration information to an application program when the program is invoked. Each of the configuration options is described briefly below:

BADCACHE

Saves/restores EMM page frame on each EMM access.

CGACURS

Prevents use of EGA/VGA extended cursor capability.

DYNF

Specifies number of file handles for dynamic overlay system use.

E

Configures amount of expanded memory.

F

Sets maximum number of file handles.

INFO

Displays memory configuration information at startup.

NOIDLE

Prevents detection of idle time during execution of compiled applications.

SWAPK

Specifies maximum size of disk swap file used for VM system.

SWAPPATH

Specifies location of VM swap file.

TEMPPATH

Controls placement of temporary sort and index files.

X

Excludes available memory.

You should adhere to the following rules when specifying these environment settings regardless of whether they are specified on the application command line or in the CLIPPER environment variable:

- Preface each setting with a double-slash
- Place a single blank space between settings
- Place a colon between setting and argument with no intervening space

SET CLIPPERCMD=<*option list*>

Specifies a list of compiler options used each time the compiler is executed. Options defined in CLIPPERCMD are overridden by the same options specified on the compiler command line.

SET INCLUDE=<*pathSpec list*>

Specifies one or more paths, separated by semicolons (;), the compiler uses to search for header (#include) files if the files cannot be found in the current directory. If the /I compiler option is used to specify an additional search directory, the indicated directory is searched after the current directory and before the path specified in the INCLUDE variable. The default location for CA-Clipper header files is the \CLIP53\INCLUDE directory.

SET LIB=<*pathSpec list*>

Specifies one or more paths, separated by semicolons (;), that the linker(s) uses to search for library (.LIB) not found in the current directory at link time. At runtime, the LIB environment variable is used to search for files not in the directory where the executable file (.EXE) is located or the path specified by the environment variable. The default location of the CA-Clipper libraries is the \CLIP53\LIB directory.

SET OBJ=<*pathSpec list*>

Specifies one or more paths, separated by semicolons (;), the linker(s) uses to search for object (.OBJ) files if the files cannot be found in the current directory. Object (.OBJ) files are created by the CA-Clipper compiler which creates them either in the current directory or the location specified with the /O command line option.

SET RMAKE=<*option list*>

Specifies a list of RMAKE command line options processed each time you invoke RMAKE.EXE. Options defined in the RMAKE environment variable are overridden by the same options specified on the RMAKE command line.

SET TMP=<*pathName*>

Specifies a single directory where both CLIPPER.EXE and BLINKER.EXE write temporary work files. If this variable is not specified, temporary files are created in the current directory. The /T compiler option can be used to override the TMP setting for the current compiler session.

Chapter 11
Operators

Listed below is a summary of the CA-Clipper operators covered in the "Language Reference" chapter of the *Reference Guide, Volume 1.*

Operators

<cString1> $ <cString2>

Performs a case-sensitive substring search and returns true (.T.) if <cString1> is found within <*cString2*>.

<nNumber1> % <nNumber2>

Divides <*nNumber1*> by <*nNumber2*> and returns the remainder as a number.

&<cMacroVar>[.]
&(<cMacroExp>)

Allows runtime compilation of expressions and text substitution within strings. Whenever the macro operator (&) is encountered, the operand is submitted to a special runtime compiler referred to as the macro compiler that can compile expressions, but not statements or commands.

()

Used in expressions either to group certain operations in order to force a particular evaluation order or to indicate a function call.

<nNumber1> * <nNumber2>

Multiplies <*nNumber1*> and <*nNumber2*> and returns the result as a number.

<nNumber1> ** <nNumber2>
<nNumber1> ^ <nNumber2>

Raises <*nNumber1*> to the power of <*nNumber2*> and returns the result as a number.

<nNumber1> + <nNumber2> (addition)
<dDate> + <nNumber> (addition)
<cString1> + <cString2> (concatenation)

For numeric operands, adds the values and returns the result as a number. This operator may also be used as a unary positive sign with a single numeric operand. For the date/numeric operand combination, adds <*nNumber*> as a number of days to <*dDate*> and returns the result as a date. For two character operands, concatenates <*cString2*> onto the end of <*cString1*>.

++<idVar> **(prefix increment)**
<idVar>++ **(postfix increment)**

Increments the value of <*idVar*> by one.

<nNumber1> - <nNumber2> **(subtraction)**
<dDate1> - <dDate2> **(subtraction)**
<dDate> - <nNumber> **(subtraction)**
<cString1> - <cString2> **(concatenation)**

For numeric operands, subtracts the values and returns the result as a number. This operator may also be used as a unary negative sign with a single numeric operand. For the date/numeric operand combination, subtracts <*nNumber*> as a number of days from <*dDate*> and returns the result as a date. For two character operands, concatenates <*cString2*> onto the end of <*cString1*>. Trailing blanks are trimmed from <*cString1*> and concatenated to the end of the return string.

--<idVar> **(prefix decrement)**
<idVar>-- **(postfix decrement)**

Decrements the value of <*idVar*> by one.

<nNumber1> / <nNumber2>

Divides <*nNumber1*> by <*nNumber2*> and returns the result as a number.

<exp1> <> <exp2>
<exp1> != <exp2>
<exp1> # <exp2>

Compares two values of the same data type and returns true (.T.) if <*exp1*> is not equal to <*exp2*>.

<object>:<message>[(<argument list>)]

Sends a message to an object.

<idVar> := <exp>

Assigns values of any data type to variables of any storage class.

<lCondition1> .AND. <lCondition2>

Performs a logical AND operation on the two operands and returns the result as a logical value.

! <lCondition>
.NOT. <lCondition>

Performs a logical NOT operation on the operand and returns the result as a logical value.

<lCondition1> .OR. <lCondition2>

Performs a logical OR operation on the two operands and returns the result as a logical value.

<exp1> < <exp2>

Compares two values of the same data type and returns true (.T.) if <*exp1*> is less than <*exp2*>.

<exp1> <= <exp2>

Compares two values of the same data type and returns true (.T.) if <*exp1*> is less than or equal to <*exp2*>.

<exp1> = <exp2>

Compares two values of the same data type and returns true (.T.) if <*exp1*> is equal to <*exp2*> according to CA-Clipper rules of comparison.

<exp1> == <exp2>

Compares two values of the same data type for exact equality or equivalence depending on the data type. It returns true (.T.) if <*exp1*> is found to be exactly equal to <*exp2*>.

<exp1> > <exp2>

Compares two values of the same data type and returns true (.T.) if <*exp1*> is greater than <*exp2*>.

<exp1> >= <exp2>

Compares two values of the same data type and returns true (.T.) if <*exp1*> is greater than or equal to <*exp2*>.

<idVar> = <exp>

Assigns a value to a variable.

<idVar> += <cString>	**(concatenation)**
<idVar> += <nNumber>	**(addition)**
<idVar> -= <cString>	**(concatenation)**
<idVar> -= <nNumber>	**(subtraction)**
<idVar> -= <dDate>	**(subtraction)**
<idVar> *= <nNumber>	**(multiplication)**
<idVar> /= <nNumber>	**(division)**
<idVar> %= <nNumber>	**(modulus)**
<idVar> **= <nNumber>	**(exponentiation)**

Performs indicated operation using <*idVar*> and <*nNumber*> as operands, and then assigns the resulting value to <*idVar*>.

<idAlias>-><idField>
<idAlias>->(<exp>)
(<nWorkArea>)-><idField>
(<nWorkArea>)->(<exp>)
FIELD-><idVar>
MEMVAR-><idVar>

When used with <*idAlias*> or <*nWorkArea*> as the first operand, accesses field information or evaluates an expression in the indicated work area. Also used to make an explicit reference to a field or variable using either the FIELD or the MEMVAR alias.

@<idVar>

Passes variables by reference to functions or procedures invoked with function-calling syntax.

<aArray>[<nSubscript>, ...]
<aArray>[<nSubscript1>][<nSubscript2>] ...

[ ] specifies a single array element. May be in C or Pascal syntax.

{ <exp list> }	**(literal array)**
{ |<param list>|	
<exp list> }	**(code block definition)**

Delimits references to literal arrays or code blocks.

Chapter 12
Comment Indicators

Listed below is a summary of the CA-Clipper language comment indicators.

Comment Indicators

/*<comment text>*/

Used for multiple-line comments, or comment blocks. All text following the /* symbol is ignored by the compiler, including carriage return/linefeeds, until the */ symbol is encountered to indicate the end of the comment block.

// <comment text>
**** <comment text>***

Used to place a single-line comment on a line by itself. All text following the // or * symbol until the next carriage return/linefeed is treated as a comment and ignored by the compiler.

<statement> // <inline comment text>
<statement> && <inline comment text>

Used to place an inline comment at the end of another language statement. All text following the // or && symbol until the next carriage return/linefeed is treated as a comment and ignored by the compiler.