FlagShip

Object Oriented Database Development System

Cross-Compatible to Unix, Linux and MS-Windows

MULTISOFT

Release 8.1

Section FUN

The whole FlagShip 8 manual consist of following sections:

Section	Content
GEN	General information: License agreement & warranty, installation and de-installation, registration and support
LNG	FlagShip language: Specification, database, files, language elements, multiuser, multitasking, FlagShip extensions and differences
FSC	Compiler & Tools: Compiling, linking, libraries, make, run-time requirements, debugging, tools and utilities
CMD	Commands and statements: Alphabetical reference of FlagShip commands, declarators and statements
FUN	Standard functions: Alphabetical reference of FlagShip functions
OBJ	Objects and classes: Standard classes for Get, Tbrowse, Error, Application, GUI, as well as other standard classes
RDD	Replaceable Database Drivers
EXT	C-API: FlagShip connection to the C language, Extend C System, Inline C programs, Open C API, Modifying the intermediate C code
FS2	Alphabetical reference of FS2 Toolbox functions
QRF	Quick reference: Overview of commands, functions and environment
PRE	Preprocessor, includes, directives
SYS	System info, porting: System differences to DOS, porting hints, data transfer, terminals and mapping, distributable files
REL	Release notes: Operating system dependent information, predefined terminals
APP	Appendix: Inkey values, control keys, ASCII-ISO table, error codes, dBase and FoxPro notes, forms
IDX	Index of all sections
fsman	The on-line manual “f**sman**” contains all above sections, search function, and additionally last changes and extensions

multisoft Datentechnik, Germany

Object Oriented Database Development System,
Cross-Compatible to Unix, Linux and MS-Windows

Section FUN

Manual release: 8.1

For the current program release see your Activation Card,
or check on-line by issuing *FlagShip -version*

Note: the on-line manual is updated more frequently.

Copyright

Made in Germany. Printed in Germany.

Trademarks

Headquarter Address

multisoft Datentechnik
Schönaustr. 7
84036 Landshut
Germany

Phone: *(*+49) 0871-3300237

E-mail: support@flagship.de
support@multisoft.de
sales@multisoft.de

Web: http://www.fship.com

FUN: FlagShip Functions

FlagShip Functions FUN

Notation Used

The syntax of the FlagShip functions is the same as in other xBase languages, such as dBASE or Clipper. The FlagShip standard functions are included in FlagShip static or dynamic libraries and will be automatically linked in as needed. The libraries are located

- for Unix/Linux in <FlagShip_dir>/lib/libFlagShip_8*.a, libFlagShip_8*.so
- for Unix/Linux in /usr/lib/libFlagShip_8*.a, libFlagShip_8*.so
- for MS-Windows in <FlagShip_dir>/lib/FlagShip_8*.lib, FlagShip_8*.dll

see RELease Notes for location of <FlagShip_dir>

The following notation is used throughout this manual:

Syntax:

The required syntax, keywords and arguments of the command. The **...—>** mark denotes, that the syntax part is continued on the next line.

Syntax:

return_val = FUNCT **( )**

or:

return_val := FUNCT **(**arg1 [, arg2 [, arg3]]**)**

or:

FUNCT **(**arg1 [, arg2 [, arg3]]**)**

return_val

The resulting return value, e.g. "retC" for returning a character or a string. If the value is not needed, the assignment may be omitted.

retC, retN, retD, retL

Return value of type character, numeric, date or logical (see LNG.2.6..2.8).

retA, retS

Return value of type array or screen.

arg1...

The function arguments, sequentially numbered as arg1, arg2, ... The syntax used for "arg" is generally "exp?" where "?" is the type of this argument, see below.

arg1[,arg2]

The optional arguments (here arg2) are enclosed in [ ] brackets. It is also possible to combine more than one optional argument, such as ([arg1 [,arg2 [,arg3]]]). In such cases, entering all items is optional. You may give (arg1) or (arg1, arg2) or (arg1, arg2, arg3) or nothing at all ( ). To skip an argument, at least the comma "," (or NIL and comma) must be entered, e.g. (arg1, , arg3) or (NIL, NIL, arg3). Do not enter the [ ] brackets. See also LNG.2.3.2 and (CMD) PARAMETERS.

<item>

The text within angle brackets informs you which type of information you should specify; not the item itself. Do not enter the brackets.

item [item ...]

The item may be entered more than once. Do not type the [ ] brackets.

[item]

The entry is optional, you may either specify it or not. Do not type the [ ] brackets.

[item1 [,item2]]

The entry of both item1 and item2 is optional, you may give item1 or item1,item2 or nothing at all. Do not type the [ ] brackets themselves.

exp

Constant, variable or expression of any type.

expC, expN, expD, expL

Constant, variable or expression of type character, numeric, date or logical (see LNG.2.8).

expA, expB ,varS

Constant or variable representing an array or a code block; variable of type screen.

Arguments/Options:

Explanation of the required or optional arguments.

Multiuser:

Where special or additional requirements or actions in the multi- user and/or multi-tasking (or network) environment are necessary, they will be listed in this paragraph.

Example:

Example of one or more function usage possibilities, put into the program context.

Compatibility:

The function syntax and return value have the same syntax as in other xBase dialects, such as Clipper. If differences exist, they are noted here.

Include:

If a special #include file is available or affected (except the default std.fh), it will be listed.

Related:

Equivalent, related or similar commands and functions.

COMMANDS, KEYWORDS and standard FUNCTIONS will be specified in this manual in uppercase, but their case is disregarded during compilation.

FlagShip also supports any number of user defined functions (UDF) programmed in the FlagShip (xBase) or C language, see LNG.2.3.2. The standard functions that follow are listed in alphabetical order and may be used as a language reference. For a summary of the standard functions, see sections QRF and LNG.

AADD ()

Syntax:

`ret = AADD (expA1 [, exp2])`

Purpose:

Adds a new element to the end of an array.

Arguments:

<**expA1**> is the array to add a new element to.

<**exp2**> is the value of any type assigned to the new element. If <exp2> is another array, the new <expA1> element will contain a reference to it. If <exp2> is not specified, NIL is assumed.

Returns:

<**ret**> is the evaluated value of <exp2>

Description:

AADD() is an array function that increases the actual length (of the first dimension) by one. The newly created array element is assigned the value specified by <exp2>.

AADD() is used to dynamically increase an array when required. A good example of this is the GETLIST array used by the GET system to hold GET objects. The READ or CLEAR GETS commands set the length to zero; any @..GET command adds a new element, increasing the array length.

AADD() is similar to ASIZE(), which can increase or reduce an array to a specified size; AADD(), however, can assign a value to the new element, while ASIZE() cannot. The AINS() function is different from AADD(), since it moves elements within an array, but it does not change the arrays length.

Example:

```
LOCAL arr1 := {}, arr2[3,2]             // empty arrays
AADD (arr1, 5)                          // now: arr1[1]
AADD (arr2, {"text1", NIL})             // now: arr2[4,2]
? arr1[1], arr2[3,1], arr2[4,1]         // 5 NIL text1
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

AINS(), ASIZE()

ABS ()

Syntax:

`retN = ABS (expN)`

Purpose:

Returns the absolute value of a numeric expression.

Arguments:

<**expN**> is the numeric expression to evaluate.

Returns:

A positive numeric value, representing the absolute value of the argument.

Description:

If, as a result of some calculation, you need a positive number to express the quantity, you can use ABS () to eliminate the negative sign.

Example:

```
a1 = 21
a2 = 35
? "The difference is ", ABS(a1-a2)      // 14
? "The difference is ", ABS(a2-a1)      // 14
```

Classification:

programming

Related:

MAX(), MIN()

ACHOICE ()

Syntax:

```
retN = ACHOICE (expN1, expN2, [expN3], [expN4],
           axpA5, [expA6], [expC7], [expN8],
           [expN9], [@exp10], [expL11], [expC12],
           [expC13], [expCN14], [expL15], [expL16],
           [expC17] )
```

Purpose:

Displays an array of character strings in the form of a pop-up menu in a defined window.

Arguments:

<**expN1**> is the topmost row window coordinate in the range from 0 to MAXROW()

<**expN2**> is the leftmost column window coordinate in the range from 0 to MAXCOL()

The GUI coordinates are internally calculated in pixel from current SET FONT (or oApplic:Font)

<**expA5**> is the array of character strings to display. ACHOICE() displays the array elements up to the first non-character element. If all array elements are character strings, the whole array is displayed as menu items. You may specify hot-key for every item by prefacing the selectable character by "&" or "\&" or "\<". Otherwise the first character of the menu item is it hotkey.

Options:

<**expN3**> is the bottom row coordinate in the range from <expN1> +1 to MAXROW(). If not given or is <= 0, the bottom row is calculated automatically from the size of <expA5>. This auto-calculated row is set so, that more than 15 elements or <expN3> greater than MaxRow() produces scrollable Achoice().

<**expN4**> is the right column coordinate in the range from <expN2> +1 to MAXCOL(). If not given or is <= 0, the right window column is calculated automatically from the text width in <expA5> considering the header or footer text width, if such available.

<**expA6**> is an array of logical values, parallel to the array <expA5>. Its elements specify the availability of <expA5> elements to be chosen. If the parallel element is .F., the menu item with the same index is not available, and is painted in the "unselected" color; the light-bar always skips over it. If all elements are unavailable (.F.), ACHOICE() is in display mode, where there is no light-bar, and the cursor keys scroll the display up and down.

The <expA6> can be specified as logical .T. or .F. value to enable or disable all items of <expA5>. Also, the elements of the <expA**6**> array can be specified as strings, whereby their macro evaluation must result in a logical value. If <expA**6**> is not specified, .T. is assumed.

<**expC7**> is the name of a user-defined function. This function is executed every time when an unrecognized key is pressed. The function name must be specified as a character expression without parentheses or arguments. The behavior of ACHOICE() is affected by the presence of this argument.

<**expN8**> is the index position in the <expA5> array, at which the light-bar is initially placed. If not specified, the default is the first available item.

<**expN9**> is the initial relative position of the light-bar in the window, staring with 0. If not specified, the default is the initial row of the initial choice element.

<@**exp10**> is a local or dynamic variable (passed by reference) holding the GUI widget visible after processing Achoice(). The variable needs to be set to any value != NIL, an object will be assigned to. See also Visibility in the Description section below.

<**expL11**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

<**expC12**> is optional header text, displayed above the Achoice() box. The text is centered within the expN2..expN4 region. In Terminal i/o mode, the header text is considered only when also <expC14> is specified, the text is displayed at the box top line, <expN1> -1. In GUI mode, the header text is also displayed at line <expN1> -1.

<**expC13**> is optional footer text, displayed below the Achoice() box. The text is centered within the expN2..expN4 region, In Terminal i/o mode, the footer text is considered only when also <expC14> is specified, the text is displayed at the box bottom line <expN3>+1. In GUI mode, the footer text is also displayed below line <expN3>.

<**expCN14**> is optional string of 8 or more characters containing the box frame drawn in Terminal i/o around the Achoice(). You may use the B_* constants defined in box.fh (e.g. B_PLAIN, B_SINGLE etc.). If <expC14> is given, the frame is drawn at <expN1> -1, <expN2> -1, <expN3> +1, <expN4> +1 using the 3rd color pair. It also includes the header <expC12> and footer <expC13> in the frame, if such are specified. If <expC14> is not char and <expC12> or <expC13> is of type "C", B_SINGLE is used as default. In GUI mode you may specify BOX_PLAIN, BOX_SUNKEN, or BOX_RAISED, default is BOX_SUNKEN when <expC12> or <expC13> is of type "C" and is not empty.

<**expL15**> is optional logical value. If .T., the horizontal and/or vertical scrollbar (if any) is drawn outside of the Achoice() box, i.e. below <expN3> and/or right of <expN4>. Default behavior is to place scrollbars within the Achoice() box. Apply in GUI mode only, ignored otherwise.

<**expL16**> is optional logical value, considered in GUI mode. If .T., the Achoice() widget will be cleared and re-displayed by simple @..SAY to avoid clearing it when another wirget overwrites it. If <expL16> is not set or is NIL, the global setting specified by

```
   _aGlobSetting[GSET_L_ACHOICECLEAR] := .F.   // default
```

will apply. See also Tuning below for additional options and API.

<**expC17**> is optional character string value, considered in GUI mode. If given, it specifies color values for displaying the Achoice(). Five color values (see SET COLOR) are required:

1	Unselected items, without input focus	black/white
2	Selected item, without input focus	white/blue
3	Unselected items with input focus	black/white
4	Selected item with input focus	white/blue
5	Disabled items	#B0B0B0/white

Returns:

A numerical value, representing the index of the chosen element in the <expA5> array. If the selection process was aborted, ACHOICE() returns zero.

Description:

ACHOICE() is, in addition to the @..PROMPT/MENU TO command, another possibility of creating various kinds of pop-up menus; see also LNG.5.2.4. When ACHOICE() is executed, it displays the array items of <expA5> as a menu inside a window defined by the coordinates. If the number of items is greater than the length of the window, the window contents are scrolled each time you try to run the cursor past its edge.

If a key is pressed, the default action for that key is executed, if available. Then, if a UDF is specified, it is executed. ACHOICE() supports the following keys and actions, but in dependence on the UDF specified:

Key		Action	with	UDF
Cursor up	ctrl-E	Up one item		yes
Cursor down	ctrl-X	Down one item		yes
PgUp	ctrl-R	Previous window		yes
PgDn	ctrl-C	Next window		yes
Home	ctrl-A	First item in menu		no
End	ctrl-F	Last item in menu		no
ctrl-PgUp	ctrl--	First item in menu		yes
ctrl-PgDn	ctrl-^	Last item in menu		yes
ctrl-Home	ctrl-]	First item in window		yes
ctrl-End	ctrl-F	Last item in window		yes
Cursor <-	ctrl-S	Abort selection	(*)	no
Cursor ->	ctrl-D	Abort selection	(*)	no
Esc		Abort selection		no
Enter	ctrl-M	Select current item		no
Hot-key		Go to corresp. item	(**yes)	no
First letter		Go to corresp. item	(**yes)	no
Space		Select or search	(***)	no
Left-mouse double-click		Select item (GUI)		no
Left-mouse click		Select item (GUI)	(****)	no

(*) The abort by cursor right/left is controlled by global variable _aGlobSetting[GSET_L_ACHOICEABORTLR] which is per default .F. To achieve

FS4.48 and Clipper backward compatibility, assign .T. to it, or modify directly the source file achoicehand.prg accordingly.

(**) The selection via hotkey and the first character in item text is available per default (for backward compatibility) only without UDF. If you wish to make a selection via hot-key or search for the first item letter even with UDF, assign .T. to the global variable _aGlobSetting[GSET_L_ACHOICEUDFSEARCH] which default is .F. See also section "hot-keys" below for behavior of SET CONFIRM ON/OFF.

(***) The space key usually handles same as Enter, i.e. it selects the current item. You may change it by assigning .F. to the global variable _aGlobSetting[GSET_L_ACHOICESPACESEL], it default is .T. Alternatively, you also may assign oParam4:SelectBySpace := .F. in the UDF to hold this setting local to the current Achoice(). When space select is enabled, the search for leading space in text is disabled.

(****) Selecting an item by a single click on the left mouse button is enabled for GUI mode and behaves same as double click, or as the Enter key. You may disable this feature either by assigning .F. to the global variable _aGlobSetting[GSET_L_ACHOICESINGLE], whereby it default is .T., or by assigning oParam4:SelectBySingle := .F. in the UDF to hold this setting local to the current Achoice().

A key redirection using SET KEY TO has preference over the above ACHOICE() navigation keys.

You may freely modify the behavior of keyboard and mouse handling in the achoicehand.prg file, available in source in the .../system directory, see it header for compiling instruction.

The menu items are displayed in standard color, the light-bar in enhanced color, and the unavailable items in the unselected color.

Visibility: pass local/private/public variable by reference as 10th parameter, if Achoice() should remain visible after return or if the Achoice() is declared as "read-only". The widget (achoice data) may lose focus and visibility on exit from Achoice(). To let it visible, either:

a) pass less than 10 parameters to Achoice() - this is the usual case for backward compatible sources with one Achoice() at a time, the widget will be hold in a public variable _oAchoice declared in initio.prg, or
b) pass any declared variable by references as 10th parameter which will hold the widget, remaining visible for the variable visibility scope. This is required also if you need more than one Achoice() visible at a time.

You may clear the widget at any time by assigning _oAchoice := NIL for alternative (a), or by <exp10> := NIL for alternative (b) or via CLS, CLEAR SCREEN or @ r1,c1 CLEAR TO r2,c2 respectively.

Tuning:

To disable run-time-error on empty array and return 0 (i.e. behave like Clipper), set

```
_aGlobSetting[GSET_L_ACHOICEIGNEMPTY] := .T.   // default = .F.
```

If you wish to report all keys in mode 0 to UDF like Clipper, set

```
  _aGlobSetting[GSET_L_ACHOICEALLKEYS] := .T.     // default = .F.
```

You may disable the &x hot-keys (see below) by

```
  _aGlobSetting[GSET_L_ACHOICEHOTKEY ] := .F.     // default = .T.
```

which will translate all "&" to "&&", hence the possibility of hot key definition by prefacing the character with "\<" remains.

If you wish to abort Achoice() selection via Cursor right/left, set

```
  _aGlobSetting[GSET_L_ACHOICEABORTLR] := .T.     // default = .F.
```

You also may modify the Achoice() keyboard handler, see the source code in <FlagShip_dir>/system/achoicehand.prg

If you wish to ignore current SET CONFIRM settings, use

```
  _aGlobSetting[GSET_L_ACHOICECONFIRM] := .F.     // default = .T.
```

If you wish to abort Achoice() via Cursor Right/Left, set

```
  _aGlobSetting[GSET_L_ACHOICEABORTLR] := .T.     // default = .F.
```

If the Space key should not behave same as Enter key, set

```
  _aGlobSetting[GSET_L_ACHOICESPACESEL]:= .F.     // default = .T.
```

If a single mouse click on item should not behave same as Enter, set

```
 _aGlobSetting[GSET_L_ACHOICESINGLE] := .F.       // default = .T.
```

If you are using UDF and wish to search for 1st character too, set

```
  _aGlobSetting[GSET_L_ACHOICEUDFSEARCH] := .T.  // default = .F.
```

If you don't wish to automatically adjust row/col in GUI mode, set

```
  _aGlobSetting[GSET_G_L_LISTBOX_ADJ ] := .F.     // default = .T.
```

If the above adjustment is on (.T.), you may set the pixel values

```
  _aGlobSetting[GSET_G_N_LISTBOX_TOP ] := -2      // default
  _aGlobSetting[GSET_G_N_LISTBOX_BOT ] := 2       // default
  _aGlobSetting[GSET_G_N_LISTBOX_LEFT] := -7      // default
  _aGlobSetting[GSET_G_N_LISTBOX_RIGH] := 6       // default
  _aGlobSetting[GSET_G_N_COMBO_HEIGHT] := 4       // default
```

The default setting for <expL16>, if not given, is specified by

```
  _aGlobSetting[GSET_L_ACHOICECLEAR] := .F.       // default
```

where .T. enables re-displaying the Achice() widget by @..SAY. The corresp. function AchoiceClear() is available in achoicehand.prg, (with an API to your own function), see comments there for details.

In Terminal i/o mode, you may set

```
  _aGlobSetting[GSET_T_L_ACHOICE_LBOX] := .F.
```

if you don't wish to use the extended functionality via ListBox object (the default is .T. which enables the extend. functionality). The Achoice() will be slightly faster for large arrays. In this case, parameters 10..14 are ignored and the UDF receives 3 arguments only (i.e. without the ListBox object). This setting is ignored in GUI mode, where the ListBox class is always used.

Hot keys

Every first menu item character (case insensitive) is automatically a hot-key. The item is then selected when the corresponding key is pressed. You may specify other hot-key characters within the items text by prefacing the character by "&" or "\&" or "\<". This kind of hot-key is then preferred over the first character. If you wish to display ampersand "&" within the text, either use "&&" or add a space after the ampersand. To disable the &x hotkey, see tuning. If an item was found, the selection terminates when SET CONFIRM is OFF (the default), or requires additional Return key press when SET CONFIRM is ON. This behavior corresponds to selection in @..PROMPT/ MENU TO. However, you may disable this feature by assigning

```
_aGlobSetting[GSET_L_ACHOICECONFIRM] := .F.
```

which then behaves same as Clipper and FlagShip 4.4x, where the search selection needs to be confirmed by Return key. Alternatively you may freely modify the behavior in the achoicehand.prg available in source code in the .../system directory.

Colors:

In Terminal i/o, following color pairs (see SET COLOR) are used:

• unselected item w/o focus	S TANDARD (1st color pair)
• selected item w/o focus	U NSELECTED (5th color pair)
• unselected item with focus	S TANDARD (1st color pair)
• selected item with focus	E NHANCED (2nd color pair)
• border, header	B ORDER (3rd color pair)
• caption	S TANDARD (1st color pair)
• caption's accelerator key	B ACKGROUND (4th color pair)
• drop down button	S TANDARD (1st color pair)
• disabled items	U NSELECTED (5th color pair)

You may change the assignment to SetColor() pair by

```
_aGlobSetting[GSET_A_ACHOICE_LBOX_COLOR] := {1,5,1,2,3,1,4,1,5}
```

In GUI mode, you may specify color pairs by 17th argument.

UDF usage:

If a user-defined function UDF is specified by <expC7>, ACHOICE() processes only a limited set of navigation keys automatically (see table above). All other keys generate a key exception, which passes control to the UDF for handling. Control is also passed to the UDF when there are no more keys to process (idle).

When ACHOICE() calls the UDF, four parameters are passed to it:

- the status mode,
- the current highlighted element number in <expA5>,
- its relative position in the menu window, and
- the ListBox object.

The first three parameters are compatible to FS 4.4x and Clipper.

The status mode is a number which reports the state of ACHOICE() just before entering the UDF. This parameter can have the following values:

Mode	achoice.fh	Description
0	AC_IDLE	Idle, no more key pressed
1	AC_HITTOP	Cursor past the top of list
2	AC_HITBOTTOM	Cursor past the end of array
3	AC_EXCEPT	Exception key pressed
4	AC_NOITEM	Display mode, no choice

The UDF usually performs some actions based on LASTKEY() or the status mode, and returns a number to ACHOICE(), requesting appropriate action. Possible return values from the UDF are:

Return	achoice.fh	Description
0	AC_ABORT	Abort selection, return 0
1	AC_SELECT	Return the no. of selected item
2	AC_CONT	Continue ACHOICE selection
3	AC_GOTO	Go to item with the 1st letter same as LASTKEY() value

When an UDF is given, the **navigation** keys are preprocessed by ACHOICE() before the UDF is called with mode 0, 1 or 3. On all other keys, the UDF is first called with the exception mode 3, the action returned from the UDF is executed and the UDF is called again with mode 0, 1 or 2. Do not return 3 (AC_GOTO) in a mode other than 3 (AC_EXCEPT), since an endless loop will occur.

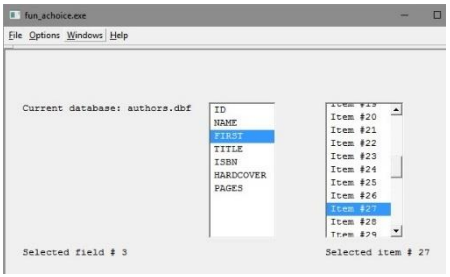
When the UDF modifies the Achoice() items, it also need to refresh the underlayed ListBox items by calling the <par4>:SetText(), see example 3 below.

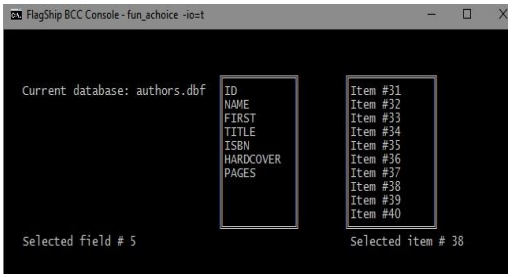
Example 1:

Displays field names as well as other choices

```
LOCAL nRet, aList, ii, oSave := 1
SET FONT "Courier", 10         // set default font (GUI only)
oApplic:Resize(25,80,,.T.)     // resize window

dbcreate("authors", {{"ID","N",3,0}, {"Name","C",20,0}, ;
   {"First","C",20,0}, {"Title","C",60,0}, {"ISBN","C",20,0}, ;
   {"Hardcover","L",1,0}, {"Pages","N",4,0} })

@ 4,3 say "Current database: " + Dbf()
USE authors
DECLARE fnames[FCOUNT()]
AFIELDS(fnames)
@ 3,34 TO 14,46 DOUBLE
nRet := ACHOICE (4, 35, 13, 45, fnames, , , , , oSave)
@ 15,3 say "Selected field # " + ltrim(nRet)

// second Achoice:
aList := {}
for ii := 1 to 50
   aadd(aList,"Item #" + ltrim(ii))
next
@ 3,54 TO 14,68 DOUBLE
nRet := ACHOICE (4, 55, 13, 67, aList)
@ 15,55 say "Selected item # " + ltrim(nRet)
```

```
setpos(24,0)
wait
```





Example 2:

Usage of the UDF in ACHOICE():

```
FUNCTION mainmenu (p1, p2, p3, p4)
LOCAL text := {"Add new data",  "Edit record",  ;
               "Delete record", "Quit" }
LOCAL validentry[LEN(text)]
AFILL (validentry, .T.)
validentry[3] = "USED() .and. ! DELETED()"
RETURN ACHOICE (p1, p2, p3, p4, text, validentry, "myudf")

#include "achoice.fh"
#include "inkey.fh"

FUNCTION myudf (status, item, relrow, obj)
DO CASE
CASE status == AC_IDLE
   @ 0,0 say TRANSFORM(DATE(), "99/99/99") + " " + TIME()
CASE status == AC_HITTOP
   KEYBOARD CHR(K_CTRL_PGDN)           // wrap to last element
CASE status == AC_HITBOTTOM
   KEYBOARD CHR(K_CTRL_PGUP)           // wrap to first element
CASE status == AC_EXCEPT
   IF LASTKEY() >= ASC(" ")
      RETURN AC_GOTO                   // goto item
   ELSEIF LASTKEY() = K_RETURN
      RETURN AC_SELECT                 // terminate ACHOICE
   ELSEIF LASTKEY() = K_ESC
      RETURN AC_ABORT                  // abort
   ENDIF
ENDCASE
RETURN AC_CONT                         // continue ACHOICE
```

Example 3:

Usage of the UDF in ACHOICE() which modifies the item's text:

```
LOCAL text := {"( ) item 1", "( ) item 2", ;
               "( ) item 3", "( ) item 4" }

do while lastkey() != K_ESC
   achoice(5,5,11,25,text,,"myUdf")
enddo
```

```
quit

FUNCTION myUdf(status, itemNum, relrow, obj)
local cItemText as character
local lOnOff    as logic

if status == 3
   /* on ENTER, mark item text ( ) with (X) and vice versa
    */
   if lastkey() = K_ENTER .and. valtype(obj) == "O"
      cItemText := obj:GetText(itemNum)   // get the item text
      lOnOff    := !empty(substr(cItemText,2,1))
      cItemText := strpoke(cItemText, 2, asc(if(lOnOff, " ", "X")))
      obj:SetText(itemNum, cItemText)     // replace item text
   endif
endif
return if(lastkey() == K_ESC, 0, 2)
```

Example 4:

```
// Multiple Achoice():
// a) remain them visible in the current UDF (i.e. during the
//    visibility scope of Local variables) or until CLS, or
// b) remain them visible forever or until CLS:

local vHold1, vHold2     // a) Achoice()s remain visible in UDF
* public vHold1, vHold2  // b) Achoice()s remain visible forever

achoice(2,2,-1,-1, {"First","second","third"}, .F., ;
              nil, nil, nil, nil, @vHold1, nil)      // read-only
ret := achoice(2,32,NIL,NIL, {"First","second","third"}, .T., ;
              nil, nil, nil, nil, @vHold2, nil)      // selectable
```

Example 5:

Use frame, header and footer, auto-calculate

```
#include "box.fh"
set color to "W+/B,N/W,,,W+/R"
aTxt := {"&one", "&two", "\&three", "\>four", "fi&ve", "six"}
iSelect := Achoice(5, 10, , , aTxt, , , , , , ,  ;
                   "Header", "Footer", B_PLAIN)
```

Classification:

programming

Compatibility:

The 10th to 14th parameters are new in FS5. In Clipper, <expN3> and <expN4> are mandatory and are not calculated automatically.

Include:

achoice.fh

Related:

@...PROMPT, MENU TO, @..GET LISTBOX, OBJ.ListBox, OBJ.ComboBox

ACLONE ()

Syntax:

`retA = ACLONE (expA)`

Purpose:

Duplicates any array, also a nested or multi-dimensional one.

Arguments:

<**expA**> is the array to duplicate.

Returns:

<**retA**> contains the duplicate of <expA>.

Description:

ACLONE() creates a complete duplicate of the given array, as opposed to the assignment <retA> := <expA> which creates another entry point to the same storage only.

ACLONE() is similar to ACOPY(), but the latter one does not duplicate nested arrays.

Example:

```
LOCAL arr1 := {1,2,{3,4}}, arr2, arr3
arr2 := ACLONE (arr1)
arr3 := arr1
arr1[3,2] := 99
? arr1[3,2], arr2[3,2], arr3[3,2]       // 99  4  99
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

ACOPY(), ADEL(), AINS(), ASIZE(), OBJCLONE(), _DISPLARR()

ACOPY ()

Syntax:

```
retA = ACOPY (expA1, expA2, [expN3], [expN4],
              [expN5] )
```

Purpose:

Copies elements from one array to another.

Arguments:

<**expA1**> is the source array.

<**expA2**> is the target array. The array must already exist and be large enough to hold the copied elements. Otherwise, some elements will not be copied.

Options:

<**expN3**> is the source array element from which to start copying. If omitted, the default is 1.

<**expN4**> is the number of elements to be copied. If not specified, all elements from <expA1>, starting with <expN3>, and up to LEN(<expA1>) are copied.

<**expN5**> is the starting element position in the target array <expA2> to receive <expN4> elements from <expA1>. If omitted, the default is 1.

Returns:

<**retA**> is a reference to the target array <expA2>.

Description:

ACOPY() copies all, or the specified number of elements from the source array to the target array. It copies values of all data types including NIL and references to code blocks and objects. If an element of the source array is a sub-array, the corresponding element in the target array will contain a reference to the sub-array. Thus, ACOPY() will not create a complete duplicate of a multi-dimensional array, as opposed to the ACLONE() function.

Example:

```
LOCAL arr1 := {9, 8, 7}, arr2 := {"a", "b", "c", "d"}
ACOPY (arr1, arr2, 2, 2, 3)
AEVAL (arr2, {|par| QQOUT(par)} )       // a  b  8  7
```

Classification:

programming

Related:

ACLONE(), ADEL(), AEVAL(), AFILL(), AINS(), ASCAN(), ASORT()

ADEL ()

Syntax:

`retA = ADEL (expA1, expN2)`

Purpose:

Deletes the specified array element, leaving the array size unchanged.

Arguments:

<**expA1**> is the target array to delete an element from.

<**expN2**> is the element number to be deleted.

Returns:

<**retA**> is a reference to the target array <expA1>.

Description:

When the specified array element is deleted, all elements from the deleted one to the end of the array are shifted up one position, and the last element becomes NIL.

If the deleted element of the target array is a reference to an sub-array (such as a nested or multi-dimensional array), the reference is lost; see also example and LNG.2.6.4.

Example:

```
DECLARE testarr[10], twodim[4,3]
FOR i = 1 TO 10
   testarr[i] = i
NEXT
? testarr[5]                            &&  5
ADEL (testarr, 5)
? testarr[5], testarr[10]               &&  6  NIL

AFILL (twodim[3], 99)   //              [1,..]: NIL NIL NIL
AFILL (twodim[4], 11)   //              [2,..]: NIL NIL NIL
*                                       [3,..]: 99  99  99
*                                       [4,..]: 11  11  11

ADEL  (twodim, 2)                       // now: asymmetric
*                                       [1,..]: NIL NIL NIL
*                                       [2,..]: 99  99  99
*                                       [3,..]: 11  11  11
*                                       [4]   : NIL

twodim[4] := {44, 55, 66}               // now: symmetric again
? twodim[4,2], twodim[2,2]              // 55  99
*                                       [3,..]: 11  11  11
*                                       [4,..]: 44  55  66
```

Classification:

programming

Related:

ACOPY(), ACLONE(), AFILL(), AINS(), LEN()

ADIR ()

Syntax:

```
retN = ADIR ([expC1], [expA2], [expA3], [expA4],
            [expA5], [expA6], [expA7], [expA8],
            [expA9], [expA10] )
```

Purpose:

Fills the specified array(s) with directory information about files and return(s) the number of files matching the wildcard pattern.

Options:

<**expC1**> is the standard Unix or Windows wildcard pattern used to select files. It can include a path. Wildcard characters "?", "*" (and on Unix regular expression like "[...]") can be used. The default pattern is "*.*" for the current directory.

<**expA2**> is an existing array to be filled with **file names** matching the skeleton <expC1>. The resulting file names abide to the UNIX convention (upper/lowercase), similar to the ls -l or DIR output, but are not sorted. On MS-Windows, the file name is reported as passed from the system, but the case is irrelevant. Only the file part, without the path is stored in the array element.

<**expA3**> is an existing array to be filled with the **sizes of files** in <expA2> given as byte count. The elements are of numeric type.

<**expA4**> is an existing array to be filled with the **modification dates** of the files. Each element is of date type.

<**expA5**> is an existing array to be filled with the **modification time**s of the files. The elements are of character type in a format of "HH:MM:SS".

<**expA6**> is an existing array to be filled with the Unix or MS-Windows file attributes. The elements are of character type. On Unix, the access rights known from "ls -la", i.e. "drwxrwxrwx" for directory or "-rwxrwxrwx" for files are reported, where each "rwx" group is for user, group and others' permission. On MS-Windows, a combination of R (read only), H (hidden), S (system), D (directory) and A (archive) or "" for regular file is reported.

<**expA7**> is an existing array to be filled with the numbers of **links** to the files. Each element is numeric (or NIL in MS-Windows when the file has insufficient access rights).

<**expA8**> is an existing array to be filled with the **inode numbers** of the files. Each element is numeric (or NIL in MS-Windows when the file has insufficient access rights, otherwise the unique file index number is returned).

<**expA9**> is an existing array to be filled with the **owner names** of the files. The array elements are of character type in upper/lowercase (or NIL on MS-Windows where this information is not available). See note.

<**expA10**> is an existing array to be filled with the **group names** of the files. The array elements are of character type in upper/lowercase (or NIL on MS-Windows where this information is not available). See note.

If not all information about the files is required, pass dummy NIL arguments, or comma only for any array you wish to skip.

Note: When the identification of owner and group names is not required, omit specifying <expA9> and <expA10>, to gain on execution speed in Unix. In MS-Windows, omitting <expA7> and <expA8> will speed-up the execution.

Returns:

<**retN**> is a numeric value, representing the number of files matching the directory skeleton.

Description:

ADIR () fills the specified arrays with the directory information about the files that matched the wildcard pattern, very similar to the Unix command ls -l or DIR on MS-Windows. The arrays are filled until there are no more files, or there are no more array elements to fill. This means that ADIR() won't complain if any array are not big enough.

The best way to use ADIR() is to first find out how many files there are which match the pattern, then to declare arrays of the returned length. These arrays can then be filled with the directory information and be displayed and given for choosing files via ACHOICE().

If the <expA6> array is not specified, directories are not included in the search. If <expA6> is specified, also "." and ".." for the current and parent directories are reported in <expA2>.

Note: The similar DIRECTORY() function creates multi-dimensional array of a variable size which is mostly easier to handle.

Example:

```
count = ADIR("*.c")
IF count = 0
   RETURN
ENDIF
DECLARE files[count], owners[count]
ADIR("*.c", files, , , , , , , owners)
```

Classification: programming, file access

Compatibility:

Due to the structure of UNIX directories, you can obtain more information on files. ADIR () defaults to Clipper's ADIR () but with the difference that file attributes which are not the same in UNIX and MSDOS. <expA7> to <expA10> are available in FlagShip only. Note also the upper/lowercase in <expA2>, <expA6>, <expA9> and <expA10>.

Related:

DIRECTORY(), ACOPY(), ACHOICE(), ADEL(), AFIELDS(), AFILL(), AINS(), ASCAN(), ASORT(), LEN(), UNIX: man ls

AELEMTYPE ()

Syntax 1:

```
retL = AElemType (expA1, expC2)
```

Syntax 2:

```
retC = AElemType (expA1)
```

Purpose:

Check if all array elements are of type <expC2> or return a string with Valtype()s used in array elements (once per variable type)

Arguments:

<**expA1**> is the array to traverse.

<**expC2**> is the type (C/D/N/I/A/L/U) which all array elements should have

Returns:

<**retL**> in syntax 1 if the check succeeds.

<**retC**> in syntax 2 containing all used element types.

Classification:

programming

Compatibility:

New in FS5.

Related:

AEVAL(), VALTYPE()

AEVAL ()

Syntax:

retA = AEVAL (expA1, expB2, [expN3], [expN4])

Purpose:

Executes a code block for each element in an array.

Arguments:

<**expA1**> is the array to traverse.

<**expB2**> is a code block to execute for each element encountered. The code block parameter list may contain one or more parameters <par1 [, par2 [, ...parN]]>, e.g.

```
     aeval(myArr, { | elem | QOUT(elem) } )
or   aeval(myArr, { | elem, pos | ;
                    QOUT("myArr[" + LTRIM(pos) + "]=", elem) }
```

During the AEVAL() process, the first parameter <par1> is replaced by content of each array element within the given scope. If more than one parameter in the code block parameter list was specified, the second parameter <par2> is replaced by the current array index, i.e. 1..len(expA1) or <expN3>..<expN4> respectively. If the code block parameter list contain more than two parameters, the third and following parameters may be used for local variables in the code block body and remain unchanged by AEVAL() self.

Options:

<**expN3**> is the starting element. If not specified, the default is one (1).

<**expN4**> is the number of elements to process from <expN3>. If not specified, the default is all elements in the array = LEN(expA1).

Returns:

<**retA**> is a reference to the array <expA1>.

Description:

AEVAL() repeats the evaluation of a code block once for each element of an array, passing the element and reference index as a block parameter in the array. The return value of the block is ignored. All elements in <expA1> are processed unless the start element or the count is specified.

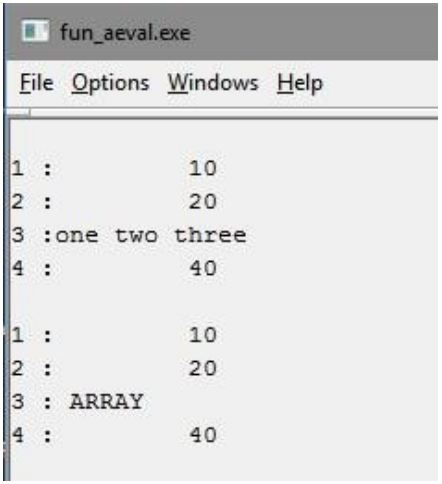
AEVAL() is similar to DBEVAL() which applies a block to each record of a database. Both can be used as a primitive for the construction of iteration commands for both simple and complex array structures.

Example:

```
// List a multi-dimensional array:
LOCAL arr1 := {10, 20, {"one ","two ","three"}, 40}, num := 0
LOCAL blk1 := {|elem| IF (VALTYPE(elem) == "A", ;
               (QOUT (ltrim(++num), ":"), ;
                AEVAL(elem, {|subelem| QQOUT(subelem)}) ;
               ), QOUT (ltrim(++num), ":", elem) )}
AEVAL (arr1, blk1)
?
```

```
// List as single-dimensional array:
num := 0
AEVAL (arr1, {|param| QOUT(ltrim(++num), ":", param)}, 1, 4)
```

Output:



Classification:

programming

Compatibility:

Available in FS and C5 only. The support of second code block parameter is available in FlagShip only.

Related:

EVAL(), DBEVAL(), LNG.2.3.3

AFIELDS ()

Syntax:

retN = AFIELDS ([expA1], [expA2], [expA3], [expA4])

Purpose:

Fill specified array(s) with information about the structure of the current database file.

Arguments:

<**expA1**> is the array to be filled with field **names** of the current database file. All elements of the array are character strings in uppercase, without trailing spaces.

<**expA2**> is the array to be filled with field **types** of the current database. All elements of the array are single characters C, N, D, L, M in uppercase.

<**expA3**> is the array to be filled with field **lengths** of the current database. The elements are of numeric type.

<**expA4**> is the array to be filled with numbers of **decimal places** of the current database fields. The elements are of numeric type. For fields that are not numeric, corresponding array elements contain zeros.

Returns:

<**retN**> is a number, representing the FCOUNT() of the current database, or the LEN() of the shortest array filled, whichever is less. If no arguments are specified, zero is returned.

Description:

AFIELDS () fills the specified arrays with the field information about the current database. If a database file is not in use, zero is returned. Instead of AFIELDS(), the more convenient DBSTRUCT() function can be used.

Example:

```
USE article
DECLARE fld_name[FCOUNT()], fld_type[FCOUNT()], ;
        fld_info[FCOUNT()]
AFIELDS(fld_name, fld_type)
FOR i = 1 TO FCOUNT()
   ? fld_info[i] := fld_name[i] + " " + fld_type[i]
NEXT
```

Classification:

programming, database

Related:

DBSTRUCT(), DBCREATE(), ADIR(), FCOUNT(), FIELDNAME()

AFILL ()

Syntax:

retA = AFILL (expA1, exp2, [expN3], [expN4])

Purpose:

Fills the specified one-dimensional array or the specified dimension of a multi-dimensional array with the value selected. To fill all elements of multi-dimensional or nested array by specified value, use AFILLALL() instead.

Arguments:

<**expA1**> is the array to be filled. With multi-dimensional arrays, enter the (dimension -1) to be filled or use AFILLALL().

<**exp2**> is an expression of any type (including an other array), the value (or reference) of which will be put in the specified array elements. The <exp2> is evaluated at the beginning of the operation, so it is possible to fill all elements with the same value.

Options:

<**expN3**> is the first element number to be filled. If not specified, the default is 1.

<**expN4**> is the number of elements to be filled, starting from the element <expN3>. If not specified, this argument defaults to all elements, starting from <expN3> to the end of the array.

Returns:

<**retA**> is a reference to the target array <expA1>.

Description:

AFILL() fills the specified array with a single value of any data type (including an array, code block, or NIL).

Example:

```
Fills 5 elements starting with the second element.

LOCAL dates[10], twodim[3,6], fourdim[2,5,3,9]
AFILL (dates, 0)                        // preset all
AFILL (dates, date(), 2, 5)             // fill array range

AFILL (twodim [2], 99)                  // twodim [2,1...6]
AFILL (fourdim[1,3,2], 0)               // fourdim[1,3,2,1...9]
```

Classification:

programming

Related:

AADD(), ACOPY(), ACLONE(), AEVAL(), AFILLALL()

AFILLALL ()

Syntax:

`retA = AFILLALL (expA1, exp2)`

Purpose:

Fills all elements of one-dimensional, multi-dimensional or nested array with the specified value.

Arguments:

<**expA1**> is the array to be filled.

<**exp2**> is an expression of any type (except an array, object or code block), the value of which will be put in all elements of the <expA1> array.

Returns:

<**retA**> is a reference to the target array <expA1>.

Description:

AFILLALL() is similar to AFILL() but supports also multi-dimensional and nested arrays. It fills elements of the specified array with a single value of any simple data type, including NIL.

Elements already containing object, code block or variable of type "special" are ignored (i.e. these elements will not be overwritten).

Example:

```
 LOCAL onedim[10], twodim[3,6], fourdim[2,5,3,9]
 AFILLALL (onedim, 0)          // set all 10 elements to zero
 AFILLALL (twodim, date())     // set all 3x6=18 elements to
curr.date
 AFILLALL (fourdim, seconds()) // all 270 elements contain same
value
```

Classification:

programming

Compatibility:

new in FS6, not available in Clipper

Related:

AADD(), ACOPY(), ACLONE(), AEVAL(), AFILL()

AINS ()

Syntax:

`retA = AINS (expA1, expN2)`

Purpose:

Inserts a NIL element into a one-dimensional array, or into the specified dimension of a multi-dimensional array, leaving the array size unchanged.

Arguments:

<**expA1**> is the target array in which the new element is to be inserted. With multi-dimensional arrays, enter the (dimension -1) to be changed.

<**expN2**> is the position where the new element is to be inserted.

Returns:

<**retA**> is a reference to the target array <expA1>.

Description:

AINS() inserts a NIL element into the array at the position <expN2>, shifting the rest of the array elements by one position, beginning with the former element <expN2>. The last element is discarded.

When AINS() is issued on a multi-dimensional symmetrical array, the symmetry will be lost, but can be restored by the program, see example.

Example:

```
LOCAL arr[10], twodim[4,3]
FOR i = 1 TO 10
   arr[i] = i
NEXT
? arr[5], arr[6], arr[10]              //  5   6  10
AINS(arr, 5)
? arr[5], arr[6], arr[10]              //  NIL 5  9

AFILL (twodim[2], 33)                  // [2,n]:  33  33  33
AINS  (twodim[2], 2)                   // [2,n]:  33 NIL  33

AINS  (twodim, 2)                      // [1,n]: NIL NIL NIL
                                       // [2]  : NIL
                                       // [3,n]:  33 NIL  33

twodim[2] := {11, 22, NIL}             // [1,n]: NIL NIL NIL
                                       // [2,n]:  11  22 NIL
                                       // [3,n]:  33 NIL  33
```

Classification:

programming

Related:

AADD(), ADEL(), ACOPY(), ACLONE(), AFILL(), ASIZE(), AEVAL()

ALERT ()

Syntax:

```
retN = ALERT (expC1, [expA2], [expC3], [expC4],
              [expN5], [expC6], [expN7])
```

Purpose:

Displays a simple modal dialog box.

Arguments:

<**expC1**> is the message shown centered (in Terminal i/o mode) and left justified in GUI mode within the alert box. You may split the text in several lines by using either ";" (semicolon) or "\n" or chr(10) character(s). If you need to display semicolon, use "\;". To insert an empty line, use "; ;" or ";;". If the argument is omitted, no text is displayed. See below for special adjustments and tuning.

Options:

<**expA2**> defines a list of possible responses to the dialog box. If not specified, a single "OK" option is presented. In GUI mode, max 3 response options are supported, the rest is ignored.

<**expC3**> defines the color specification. The default setting is `IF(ISCOLOR(),"W+/R,W+/B","W+/N,I"))`. Significant are pairs 1 and 2, see SET COLOR for further details. Ignored in GUI.

<**expC4**> defines the border characters (8 characters). The default setting is taken from _aGlobSetting[GSET_T_C_ALERTBOX], see below. Considered in Terminal i/o mode only, ignored otherwise.

<**expN5**> defines the icon type, see MBOX_* in dialog.fh. The default is MBOX_WARNING. Apply for GUI mode only, ignored otherwise.

<**expC6**> is a string with the box capture/header.

<**expN7**> is the number of default button in range 1..len(<expA2>). If not specified, 1 is the default. Apply for GUI only.

Returns:

<**retN**> is a numeric value indicating which option was chosen. If the ESC key is pressed, zero is returned.

Description:

The ALERT() function creates a simple modal dialog. It is useful in error handlers and other "pause" functions. The user can respond by moving a highlight bar and pressing the RETURN or SPACE keys, or by pressing the key corresponding to the first letter of the option, similar to MENU TO. SET KEY is not active.

The Alert() function is based on the MessageBox and WarningBox class, which provides additional functionality. Alternative functions are TextBox(), InfoBox(), ErrBox() and WarnBox().

In **GUI mode**, warning icon (yellow triangle with exclamation mark) is displayed.

In **Terminal** i/o mode, the border is displayed per default in plain ASCII characters (+---+) specified as B_PLAIN in box.fh. This avoid confusion when an error message is displayed with incorrect TERM setting. You may freely change this behavior, i.e. set the display to single or double semi-graphic border characters, by either:

a) using the 4th parameter, or

b) including the line #include "fspreset.fh" at the begin of your main program - which auto invokes (c), or

c) assign somewhere, e.g. at the begin of your application
```
_aGlobSetting[GSET_T_C_ALERTBOX] := B_DOUBLE      -or-
_aGlobSetting[GSET_T_C_ALERTBOX] := B_SINGLE
```
which are pre-defined as B_PLAIN in .../system/initio.prg, where the P_PLAIN, B_SINGLE or other B_* box define's are specified in the .../include/box.fh file.

When Alert() is invoked in a by Wopen() created sub-window, and this sub-window is too small to display the message, such Alert() will be displayed in separate pop-up window.

In **GUI** mode, the InfoBox() or ErrBox() is used automatically instead.

Alignment and Tuning:

The default alignment is left justified in GUI and centered in Terminal i/o mode. You may override this defaults by assigning
```
_aGlobSetting[GSET_G_N_ALERT_ALIGN] := numValue  // default = 0
```
where <numValue> = 0: default alignment, 1: left justify, 2: center, 3: right justify.

Every single line can additionally be individualy aligned by three special characters at the line begin (i.e. after the ";" or "\n" line separator):
`"<<!"` align left, `">>!"` align right, `"><!"` or `"<>!"` to center this line, e.g.
```
alert("><!centered;<<!left;>>!right;default")
```
Of course, these special marker are filtered out from the text. See example in <FlagShip_dir>/system/memoedithand.prg

If the line is too long, it is automatically wrapped. In GUI mode, you may specify the maximal text width and height (in pixel) by assigning
```
_aGlobSetting[GSET_G_N_ALERT_WIDTH ] := nPix   // default = 0
_aGlobSetting[GSET_G_N_ALERT_HEIGHT] := nPix   // default = 0
```
or <nPix> = 0: adjust/fit box to current window size, or <nPix> = -1: adjust/fit box to desktop size. In GUI mode, the default font oApplic:FontWindow is used, except you specify your own font object by assigning
```
_aGlobSetting[GSET_G_O_ALERT_FONT] := oFont   // default = NIL
```
In Terminal i/o mode, the displayed box and wrapping is adjusted automatically according to MaxCol() and MaxRow().

Example 1:
```
alert("hello world")
```

Example2:

```
SET FONT "Arial", 10
_aGlobSetting[GSET_T_C_ALERTBOX] := B_DOUBLE
nChoice := Alert("Exit the application now ?", {"Yes ", "No"})
if nChoice == 1
   quit
endif
```

Output:

Example3:

```
LOCAL myopt := {"retry", "main menu", "quit"}
LOCAL dbname := "address", choice, answ
DO WHILE .T.
   BEGIN SEQUENCE
     choice := mymenu()       // your main menu
     USE (dbname) SHARED NEW
     WHILE !used()
        answ := ALERT("Access denied;file: "+dbname+".dbf", myopt)
        IF answ == 0 .OR. answ == 3
           QUIT
        ELSEIF answ == 2
           BREAK
        ENDIF
        USE (dbname) SHARED NEW
     ENDDO
   END // sequence
ENDDO
```

Classification:

programming

Compatibility:

Available in FS and C5 (two parameters) only.

Related:

InfoBox(), ErrBox(), TextBox(), WarnBox(), @...PROMPT, MENU TO, ACHOICE(), @..GET, READ, HELP(), OBJ.MessageBox

ALIAS ()

Syntax:

`retC = ALIAS ([expN])`

Purpose:

Obtains the alias name of the specified working area.

Arguments:

<**expN**> is the number of the specified working area.

Returns:

<**retC**> is the alias name of the specified working area in uppercase. If no argument is specified, the alias of the current working area is returned. If no database file is in use, a null string "" is returned.

Description:

ALIAS() is used to determine the alias name of a specified working area. The alias name is assigned to the working area by the USE command or the DBUSEAREA() function. ALIAS() is the inverse of the SELECT() function; whereby SELECT() returns the working area number given the alias name.

Multiuser:

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
USE article NEW
? ALIAS(), SELECT()                    && ARTICLE  1
SELECT 0
USE magazine ALIAS Mag
? ALIAS (), ALIAS (1)                  && MAG  ARTICLE
? SELECT(), SELECT("article")          && 2    1
```

Classification:

database

Related:

SELECT, SELECT(), USE, DBUSEAREA(), oRdd:Alias

ALLTRIM ()

Syntax:

`retC = ALLTRIM (expC|expN)`

Purpose:

Removes all leading and trailing spaces from a string.

Arguments:

<**expC**> is the character string to be trimmed. <**expN**> is a numeric value converted to string first before trimmed.

Returns:

<**retC**> is the resulting string, with all leading and trailing blanks removed.

Description:

ALLTRIM(expC) has the same effect as LTRIM (TRIM (expC)).

The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADR(), and PADL() functions which center, right-justify, or left- justify strings.

Example:

```
LOCAL str1 := "   my string   "
? "[" + ALLTRIM(str1) + "]"            // [my string]
? "[" + TRIM (str1) + "]"              // [   my string]
? "[" + LTRIM(str1) + "]"              // [my string   ]
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default. The numeric parameter is supported by FlagShip only

Related:

LTRIM(), TRIM(), RTRIM(), PADC(), PADR(), PADL(), FS2:RemAll()

ALTD ()

Syntax:

`NIL = ALTD ([expN])`

Purpose:

Activates the FlagShip Debugger, if the compiler switch -nd was not specified, or enables/disables the use of the Ctrl-O key to invoke it by the user.

Arguments:

<**expN**> sets the Debugger activation state:

expN	Action
None	Invokes the Debugger if it is enabled
0	Disables the Ctrl-O key
1	Enables the Ctrl-O key
2	Enables the Ctrl-O key and invokes debugger
Other	No action

Returns:

ALTD() has no return value.

Description:

The reason Ctrl-O is used instead of Alt-D in the DOS dialects is that not all terminals operating under Unix applications support the ALT key. The Ctrl-O / debug-key can be set or changed by the FS_SET("debug") function. In FlagShip for Windows, the debug key is disabled by default (performance), you need to activate it by e.g. FS_SET("debug", K_CTRL_O)

If the application is running in GUI mode, you will need to compile with the -d switch to enable the source-code debugger and hence the Altd() and Ctrl-O trapping. In terminal i/o mode, you may invoke the debugger by Altd() or Ctrl-O key press also when compiled w/o the -d switch, see further details in sections FSC.1.3 and FSC.5.

Note that, if you write your own debugger.prg and link it with your applications, it will take precedence over FlagShip's Debugger.

Classification:

programming

Related:

FS_SET(), SET ESCAPE, SETCANCEL(), sections FSC.1.3, FSC.5

ANSI2OEM ()

Syntax:

`retC = Ansi2oem (<expC1>, [<expL2>], [<expC3>])`

Syntax 2:

`retC = AnsiToOem (<expC1>, [<expL2>], [<expC3>])`

Purpose:

convert <expC1> string from ISO/ANSI character set to PC8/ASCII/OEM character set, accepts embedded \0 bytes

Arguments:

<**expC1**> is the character string to be converted.

<**expL2**> if .F. or not given, the translation should be "as close as possible" for ANSI characters not in PC8 charset. If .T., all unknown characters are replaced by <expC3>

<**expC3**> replacement for unknown/untranslatable chars, default is '?'

Returns:

<**retC**> is the resulting, translated string

Description:

The ISO/ANSI character set differs to PC8/ASCII/OEM for nearly all character values > 127. So is e.g. a-umlaut chr(132) in ASCII/PC8 set, but chr(228) in ISO/ANSI.

The used character set depends on the TERM setting (in Terminal i/o) or is per default ANSI/ISO in GUI mode. The PC8 char set is used in DOS and in Terminal i/o FS applications, the ANSI us used by X11 and Windows.

The inverse of Ansi2Oem() is Oem2Ansi(). Both functions are used in automatic database access/assign with SET ANSI ON.

Classification:

programming

Example:

see <FlagShip_dir>/examples/umlauts.prg for complete examples

Compatibility:

New in FS5

Related:

Oem2Ansi(), SET ANSI

APPIOMODE ()

Syntax:

`retC = AppIoMode ()`

Purpose:

return the Application type (run-time type)

Returns:

<**retC**> is "G" for application running in GUI mode, "T" for application using Terminal i/o, "B" for Basic mode.

Description:

The type of application is set either at **compile/link time** by using the "FlagShip ... -io=g|t|b" switch, or at run-time when a hybrid application was created by using the -io=a compiler switch, or the -io= switch was not given at all. The **run-time mode** is determined from the used environment, or set by command-line switch -io=g|t|b passed to the executable. See also initio.prg for details.

To determine whether the X11 server is available and running, use IsGuiMode()

Example 1:

```
static aMode := {{"G","GUI"},{"T","Terminal"},{"B","Basic"}}
local  cMode := aMode[ascan(aMode, {|x| x[1] == AppIoMode()}), 2]
wait "The application is running in " + cMode + " i/o mode ..."
```

Example 2:

```
if AppIoMode() == "G"
   SET GUITRANSL TEXT  ON    // draw PC8 semi-graphic char set
   SET GUITRANSL LINES ON    // draw @..TO.. using PC8 charset
   SET GUITRANSL BOX   ON    // draw @..BOX  using PC8 charset
   SET GUICOLOR ON           // use colors also for GUI mode
endif
```

Classification:

programming

Compatibility:

New in FS5

Related:

IsGuiMode(), system/initio.prg, FSC -io=? switch

APPMDIMODE ()

Syntax:

```
retL = AppMdiMode ()
```

Purpose:

determine whether the application is instantiated as MDI (Multiple Document Interface) or SDI (Single Document Interface), the default.

Returns:

<**retL**> is true (.T.) if the application is running in MDI mode or false (.F.) otherwise.

Description:

The MDI application mode can either be set automatically by the -mdi compiler switch at link stage, or manually in initio.prg at instantiating of _gAppWindow{} class. The MDI mode apply for GUI based application only.

Classification:

programming

Compatibility:

New in FS5

Related:

system/initio.prg

APPOBJECT ()

Syntax:

```
retO = AppObject ()
```

Purpose:

return the Application object or NIL

Returns:

<**retO**> is the Application object or NIL

Description:

This function is equivalent to access to the constant "oApplic" but it checks the availability, i.e. if the application was already created. See also initio.prg source for details.

Classification:

programming

Compatibility:

New in FS5

Related:

system/initio.prg

ARRAY ()

Syntax:

`retA = ARRAY (expN1 [,expN2 [,expN3 ...]])`

Purpose:

Creates an un-initialized one or multi-dimensional, symmetrical array of specified length.

Arguments:

<**expN1**> is the number of elements in the first dimension. In FlagShip the maximum number of elements in a dimension is 65535.

Options:

<**expN2..**> is the number of elements in the second, third etc. dimension. In FlagShip, up to 65535 dimensions are supported.

Returns:

<**retA**> is the created array of specified dimensions, filled by NIL elements.

Description:

ARRAY() creates one or more multidimensional arrays, filled by NIL. It is similar to the declaration statement LOCAL, STATIC or PRIVATE or the DECLARE command, e.g.: LOCAL arr1 [expN1], arr2 [expN1,expN2,expN3]

Another option is assigning a literal array to the (declared) variable, e.g.: arr3 := {1,2,3}.

ARRAY() has the advantage that the array can be created within expressions or code blocks. Also, the array copy functions ACOPY() and ACLONE() will create new arrays. See also LNG.2.6.4 - LNG.2.7.

Example:

All the array declarations are equivalent:

```
LOCAL   arr1 [3], arr2 := {NIL, NIL, NIL}, arr3
arr3 := ARRAY (3)

PRIVATE arr4 [2,2], arr5 := {{NIL, NIL},{NIL, NIL}}, arr6
arr6 := ARRAY (2,2)
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

ACLONE(), ACOPY(), AADD(), AFILL(), AINS(), ADEL(), DECLARE, LOCAL, STATIC, PRIVATE, PUBLIC

ASC ()

Syntax:

`retN = ASC (expC)`

Purpose:

Takes the leftmost character of a character expression and returns its ASCII value.

Arguments:

<**expC**> is a character or string expression for which the ASCII value is returned.

Returns:

<**retN**> is a numeric value in the range of zero to 255, representing the first character of <expC>. For null string "", zero is returned.

Description:

The ASC() function is used if numeric calculations on a character are required. The reverse operation of ASC() is CHR().

Example:

```
? ASC("Article")                          &&   65
? ASC(SUBSTR("Article",2))                &&  134
? ASC("")                                 &&    0
? ASC("z") - ASC("c")                     &&   23
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

CHR(), INKEY(), STR(), VAL()

ASCAN ()

Syntax:

`retN = ASCAN (expA1, exp2 [,expN3 [,expN4]] )`

Purpose:

Seeks a specified value in an array.

Arguments:

<**expA1**> is the target array to scan.

<**exp2**> is an expression of any type, the value of which is to be located in the array. A code block can also be specified.

Options:

<**expN3**> is the first element to be scanned. If not specified, the default starting position is 1.

<**expN4**> is the number of elements to be scanned, counting from the array element <expN3>. If not specified, all elements starting from <expN3> to the end of the array are scanned.

Returns:

<**retN**> is a numeric value, representing the number of the first matching element, or zero if a match is not found. If <exp2> is a code block, <retN> is the position of the element where the block returned TRUE.

Description:

ASCAN() operates like SEEK when searching for a simple value in an array.

On strings beginning with the leftmost character in the target element the <exp2> value is compared to the target array element, proceeding until there are no more characters left in the <exp2> string. If there is no match, ASCAN() proceeds to the next element in the array.

<exp2> is tested against array elements character by character until the end of <exp2>. If characters are matched up to this point, the search is considered successful, otherwise ASCAN() proceeds to the next element of the array. Since ASCAN() uses the equal (=) operator for comparisons, it is sensitive to the status of SET EXACT, see LNG.2.9.

If the <exp2> is a code block, ASCAN() passes the element value to the code block parameter, and then performs an EVAL() on the block. If the code block return FALSE, the operation is repeated with the next element; otherwise, the current element number is returned.

In FlagShip, the sorting order can be changed to any required human language using FS_SET("setlang").

Example 1:

Scan for a single value or use code block to perform a case insensitive search.

```
LOCAL months := {"Jan","Feb","Mar","Apr","May","Jun", ;
                 "Jul","Aug","Sep","Oct","Nov","Dec"}
LOCAL block

elem = ASCAN(months, "Ju")              &&  6
elem = ASCAN(months, "Jul")             &&  7
elem = ASCAN(months, "Jul ")            &&  0

block  := {|par| UPPER(par) == ;
                 UPPER(SUBSTR(CMONTH(actdate),1,3)) }
actdate = DATE()
? ASCAN (months, block)                 &&  8
```

Example 2:

Scan a multi-dimensional array:

```
#include "directry.fh"

dir := DIRECTORY ("*.*")
pos := ASCAN (dir, {|x| x[F_NAME] == "address.dbf" })
IF pos > 0
   ? dir[pos, F_NAME], "size=", dir[pos, F_SIZE], ;
     "date=", dir[pos, F_DATE]
ENDIF
```

Classification:

programming

Compatibility:

The usage of code blocks is new in FS4. The usage of any user- defined sorting order is available in FlagShip only.

Related:

ACHOICE(), ACOPY(), ADEL(), ADIR(), AFILL(), AFIELDS(), AINS(), ASORT(), LEN(), SEEK, FS_SET("setlang")

ASIZE ()

Syntax:

`retA = ASIZE (expA1, expN2)`

Purpose:

Increase or reduce (resize) an array.

Arguments:

<**expA1**> is the target array to resize.

Options:

<**expN1**> is the new size of the array.

Returns:

<**retA**> is a reference to the target array <expA1>.

Description:

ASIZE() changes the actual length of the one-dimensional target array by shortening or lengthening it to match the specified length of <expN1>. If the array is reduced, elements at the end of the array are lost. When increasing the target array, new NIL elements are added to the end of the array.

Increasing ASIZE() is similar to AADD(). It is different from AINS() and ADEL() functions, which do not change the array length.

Example:

```
LOCAL arr1 := {1,2}, arr2[0]
ASIZE (arr1, 4)                         // 1, 2, NIL, NIL
ASIZE (arr1, 1)                         // 1
ASIZE (arr2, 5)                         // NIL, NIL, NIL, NIL, NIL
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

AADD(), AFILL(), AINS(), ADEL()

ASORT ()

Syntax:

retA = ASORT (expA1, [expN2], [expN3], [expB4])

Purpose:

Sorts the elements of an array in any order.

Arguments:

<**expA1**> is the target array to sort.

Options:

<**expN2**> is the position of the first element to be sorted. If not specified, the default value is 1.

<**expN3**> is the number of the array elements to be sorted, starting with the element <expN2>. The default value, if not specified, is all elements, starting with element <expN2>.

<**expB4**> is an optional code block used to determine sorting order. If not specified, the default order is ascending.

Returns:

<**retA**> is a reference to the target array <expA1>.

Description:

ASORT() is similar to indexing or sorting a database. All array elements which are being sorted must be of the same data type (C, N, D, L), which does not mean that the entire array has to be of the same type.

By using a code block <expB4>, any sorting order may be imposed. The code block receives two parameters, representing two subsequent array elements n and n+1. The code block must return TRUE if the elements are in sorted order. ASORT() executes the codeblock until all the desired elements are sorted. Sorting e.g. multi-dimensional arrays is only possible by using the code block, see examples below.

In FlagShip, the sorting order can by changed to any required human language using FS_SET("setlang").

Example 1:

Sort an array in ascending and descending order:

```
LOCAL arr1 := {4, 5, 2, 19, 10}
LOCAL arr2 := {3, "Xyz", "Bc", "ab", .T.}

ASORT (arr1)                              // 2 4 5 10 19
ASORT (arr2, 2, 3)                        // 3 Bc Xyz ab .T.
ASORT (arr1,,, {|one,two| one > two})     // 19 10 5 4 2

ASORT (arr2,2,3, {|one,two| ;
       UPPER(one) <= UPPER(two)})         // 3 ab Bc Xyz .T.
```

Example 2:

Sort a multi-dimensional array in ascending order:

```
USE address NEW
arr3 := DBSTRUCT()                          // arr3[4,n]
ASORT (arr3,,, {|x,y| x[1] <  y[1] }     // field names
ASORT (arr3,,, {|x,y| x[3] <= y[3] }     // field length
```

Example 3:

Another example for sorting one dimension of a multi-dimensional array in ascending order:

```
LOCAL arr4 := {1, 2, {5, 2, 7}, {"b", "X", "a"}}, arr5
ASORT (arr4[3])                             // [3,n]: 2 5 7
arr5 := ASORT (arr4[4])                     // [4,n]: X a b
arr5[2] := "y"
AEVAL (arr5,    {|par| QQOUT(par)})      // X y b
AEVAL (arr4[4], {|par| QQOUT(par)})      // X y b
no := 0                                     // see block blk1
AEVAL (arr4, blk1)                          // in AEVAL()
*                                              1: 1
*                                              2: 2
*                                              3: 2 5 7
*                                              4: X y b
```

Example 4:

Sorting an directory:

```
#include "directry.fh"
dir := DIRECTORY ("*.*")
dir := ASORT (dir,,, {|x,y| x[F_NAME] < y[F_NAME]})
AEVAL (dir, {|x| QOUT(x[F_NAME])})  // print sorted by name

ASORT (dir,,, {|x,y| x[F_DATE] <= y[F_DATE]})  // sort date
AEVAL (dir, {|x| QOUT(x[F_DATE], x[F_NAME])})  // print it
```

Classification:

programming

Compatibility:

The usage of code blocks is new in FS and C5. The usage of any user-defined sorting order according to the human language is available in FlagShip only.

Related:

FS2:CharSort(), AFIELDS(), SORT, INDEX, FS_SET("setlang"), LNG.2.3.3

AT ()

Syntax:

`retN = AT (expC1, expC2, [expN3])`

Purpose:

Searches through a character string for the first instance of the given substring.

Arguments:

<**expC1**> is the character string to look for.

<**expC2**> is the character string to search through.

<**expN3**> is the start position, default is 1

Returns:

<**retN**> represents the position of the first letter of the first instance of the given substring. If the search is not successful, AT() returns zero.

Description:

AT() is useful when you need to know the exact position of a token in a string. If you only need information on whether the substring appears within a string, you can use the $ operator. To find the last instance of a substring within a string, use RAT().

Example 1:

```
AT("cd", "abcdabcdabcd")                // 3
AT("cd", "abcdabcdabcd", 2)             // 3
AT("cd", "abcdabcdabcd", 3)             // 3
AT("cd", "abcdabcdabcd", 4)             // 7
AT("cd", "abcdabcdabcd", 8)             // 11
AT("cd", "abcdabcdabcd", 12)            // 0
AT("ef", "abcdabcdabcd")                // 0
```

Example 2:

```
cStr   := "abcdabcdabcd"
cToken := "cd"
iCount := 0
iPos   := AT(cToken, cStr)
while iPos > 0
   iCount++
   iPos := AT(cToken, cStr, iPos +1)
enddo
? "The [" + cStr + "] string contains", ltrim(iCount), ;
  "times token [" + cToken + "]"         // here: 3-times
```

Classification:

programming

Compatibility:

Embedded zero bytes are supported. The 3rd parameter is new in FS5

Related:

$, RAT(), STRTRAN(), SUBSTR(), FS2:AtToken(), FS2:AtNum()

ATAIL ()

Syntax:

`ret = ATAIL (expA)`

Purpose:

Returns the value of the highest numbered element of an array.

Arguments:

<**expA**> is the target array.

Returns:

<**ret**> is the value (or a reference to an array, object or code block). The array remains unchanged.

Description:

ATAIL() is shorthand for ret := expA[LEN(expA)] to obtain the last element of an array.

Example:

```
arr1 := {1, 2, {"a", "b"}, 3}
arr2 := {{1, 2}, {3, 4}}
val  := ATAIL(arr1)
? VALTYPE(val), val                    // N  3
val  := ATAIL(arr2)
? VALTYPE(val), val, val[2]            // A  ARRAY  4
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

LEN(), LNG.2.6.4

ATANYCHAR ()

Syntax:

`retN = AtAnyChar (expC1, expC2)`

Purpose:

search for the first occurrence of any char of <expC1> pattern in the source <expC2> string

Arguments:

<**expC1**> is the string whose characters are searched in <expC2> <**expC2**> is the string which is searched thru

Returns:

<**retN**> is the position in <expC2> if found, or 0 otherwise.

Description:

You may see this function is a combination of

```
AT( substr(expC1,1,1), expC2 )
AT( substr(expC1,2,1), expC2 )
:
AT( substr(expC1,n,1), expC2 )
```

and returning the lowest position found. It is of course done more efficiently.

Classification:

programming

Compatibility:

New in FS5

Related:

AT(), Rat(), FS2:AtToken()

AUTOxLOCK ()

Syntax 1:

```
retL = AUTORLOCK ([expO1], [expN2])
```

Syntax 2:

```
retL = AUTOFLOCK ([expO1], [expN2])
```

Syntax 3:

```
retL = AUTOUNLOCK ([expO1], [expN2])
```

Purpose:

Performs a "smart" RLOCK(), FLOCK() or UNLOCK to enable a write access on a SHARED database in the current working area.

Options:

<**expO1**> is an optional RDD object, if invoked from an RDD method or when using an RDD object other than of the DataServer class.

<**expN2**> is an optional try period in seconds, which temporarily overrides the current SET AUTOLOCK TO status.

Returns:

<**retL**> is a logical value, which signals success or error. This is equivalent to the RLOCK(), FLOCK() or DBUNLOCK() return value.

Description:

These functions are designed to manage automatic record and file locking and unlocking in cooperation with the SET AUTOLOCK feature. They are also called from the standard RDD driver on SHARED databases during a write attempt, when the programmer had not yet locked the record or database.

As opposed to RLOCK() and FLOCK(), these functions don't immediately report a lock failure, but try to lock the record or file repeatedly during the, by SET AUTOLOCK, specified period. If this fails, a user communication window is displayed, to allow him to choose a new trial period, perform a break to menu, ignore the lock, or abort the execution.

The behavior is highly flexible, since the source code is available in the <FlagShip_dir>/system/autolock.prg file, and may therefore be modified by the programmer.

Example:

See examples in SET AUTOLOCK command

Classification: database
Compatibility: Available in FlagShip only
Source: Available in <FlagShip_dir>/system/autolock.prg
Related: SET AUTOLOCK, USE...SHARED, RLOCK(), FLOCK(), UNLOCK, DataServer Class

BIN2I ()

Syntax:

`retN = BIN2I (expC)`

Purpose:

Takes a string in the format of a 16-bit signed integer and converts it to a numeric value.

Arguments:

<**expC**> is the 16-bit signed short integer in the form of two bytes. The least significant byte comes first (Intel convention). If there are more than two characters specified, the rest are ignored.

Returns:

<**retN**> is a numeric value in the range -32768 to +32767, representing the converted signed short integer.

Description:

BIN2I() is to be used after reading data with FREAD(), to convert a two-byte character string formatted as a signed integer to FlagShip numeric format. The inverse operation of BIN2I() is I2BIN().

Example 1:

Convert the string CHR(160)+CHR(91) which is the binary representation of the decimal value 23456 (= 5BA0hex ; 5Bhex= 91, A0hex= 160) to a numeric variable:

```
LOCAL str, num
? str := I2BIN(23456)                              // " ["
? ASC(SUBSTR(str,1,1)), ASC(SUBSTR(str,2,1))       // 160 91
? num := BIN2I (str)                               // 23456
```

Example 2:

```
LOCAL handle, numstr
handle = FOPEN("data.num")
DO WHILE .T.
   numstr = FREADSTR(handle, 2)
   IF LEN(numstr) < 2
      EXIT
   ENDIF
   ? BIN2I(numstr)
ENDDO
FCLOSE(handle)
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

BIN2L(), BIN2W(), I2BIN(), L2BIN(), FREAD(), FREADSTR()

BIN2L ()

Syntax:

`retN = BIN2L (expC)`

Purpose:

Takes a string in the format of a 32-bit signed long integer and converts it to a numeric value.

Arguments:

<**expC**> is the 32-bit signed long integer in the form of four bytes. The least significant byte comes first (Intel convention). If there are more than four characters specified, the rest are ignored.

Returns:

<**retN**> is a numeric value in the range -2,147,483,647 to +2,147,483,646, representing the converted signed long integer.

Description:

BIN2L() is to be used after reading data with FREAD(), to convert long integers to FlagShip numeric format. The inverse operation of BIN2L() is L2BIN().

Example 1:

Convert a string with the binary representation of the decimal value 1234567890 (= 499602D2hex ; 49hex= 73, 96hex= 150, 02hex= 2, D2hex= 210) to a numeric variable:

```
LOCAL str, num
? str := L2BIN(1234567890)                      // "????" (binary)
? ASC(SUBSTR(str,1,1)), ASC(SUBSTR(str,2,1)),; // 210  2
  ASC(SUBSTR(str,3,1)), ASC(SUBSTR(str,4,1))   // 150 73
? num := BIN2L (str)                            //1234567890
```

Example 2:

```
LOCAL handle, numstr
handle = FOPEN("data.num")
DO WHILE .T.
   numstr = FREADSTR(handle, 4)
   IF LEN(numstr) < 4
      EXIT
   ENDIF
   ? BIN2L(numstr)
ENDDO
FCLOSE(handle)
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

BIN2I(), BIN2W(), I2BIN(), L2BIN(), FREAD(), FREADSTR()

BIN2W ()

Syntax:

`retN = BIN2W (expC)`

Purpose:

Takes a string in the format of a 16-bit unsigned integer and converts it to a numeric value.

Arguments:

<**expC**> is the 16-bit unsigned short integer in the form of two bytes. The least significant byte comes first (Intel convention). If there are more than two characters specified, the rest are ignored.

Returns:

<**retN**> is a numeric value in the range 0 to +65,535, representing the converted unsigned short integer.

Description:

BIN2W() is similar to BIN2I(), except with regard to the range. It is generally used after reading data with FREAD() to convert unsigned short integers to FlagShip numeric format.

Example:

```
LOCAL handle, numstr
handle = FOPEN("data.num")
DO WHILE .T.
   numstr = FREADSTR(handle, 2)
   IF LEN(numstr) < 2
      EXIT
   ENDIF
   ? BIN2W(numstr)
ENDDO
FCLOSE(handle)
```

Example:

See also example in BIN2I()

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

BIN2I(), BIN2L(), I2BIN(), L2BIN(), FREAD(), FREADSTR()

BINAND ()

Syntax:

`retN = BinAnd (expNC1, expNC2, [...expNCn])`

Purpose:

Perform binary AND of two or more numeric expressions

Arguments:

<**expN1...expNn**> are numeric constants or variables which should be binary AND-ed. You may alternatively specify the number as hexadecimal string in the syntax "FF...FF" or "0xFF..FF" with max. 8 hex chars.

Returns:

<**retN**> is the product of the AND operation or NIL on error.

Description:

Binary AND set the resulting bit on (1) only if the same bit is on in all of the arguments, otherwise off (0). The functionality is equivalent to C syntax

```
iRet = iExp1 & iExp2 & iExp3 &...
```

and is often used to clear specific bits in <expN1> by bits not set or not available in the <expN2> mask. With other words, only bits set in the mask <expN2> remain set in the <expN1> result.

Example:

```
? BinAnd(0, 0)                      // 0
? BinAnd(0, 1)                      // 0
? BinAnd(1, 0)                      // 0
? BinAnd(1, 1)                      // 1
? BinAnd(1, 1, 0)                   // 0 : 1 AND 1 = 1 AND 0 = 0

? BinAnd(65535, 255)                // 255
? BinAnd(65535, 255, 5)             // 5
? BinAnd("0xFFFFFF", "0xFAFFF0F")   // 16776975 = 0xFAFF0F
? BinAnd("0xABCDEG", 14)            // NIL (error, invalid hex)

// check if specified bits are set

nVar := 2474                        // bin: 100110101010
? "Bit 2 and 6 in", ltrim(nVar), "are " + ;
  if(BinAnd(nVar,"0x22") == 0,"not ","") + "set"
? Isbit(nVar,2), Isbit(nVar,6)      // .T. .T.

// split 32-bit integer number into bytes

nNum  := 305419896                                  // 0x12345678
byte1 := BinRShift(BinAnd(nNum,"0xFF000000"),24)  //  18 = 0x12
byte2 := BinRShift(BinAnd(nNum,"0x00FF0000"),16)  //  52 = 0x34
byte3 := BinRShift(BinAnd(nNum,"0x0000FF00"),8)   //  86 = 0x56
byte4 := BinAnd(nNum,"0xFF")                        // 120 = 0x78
```

Classification:

programming

Compatibility:

New in FS5

Related:

BinOR(), BinXOR(), Hex2num(), Num2hex() FS2:IsBit(), FS2:SetBit(), FS2:ClearBit(), FS2:NumAnd(), FS2:CharAnd()

BINOR ()

Syntax:

`retN = BinOr (expNC1, expNC2, [...expNCn])`

Purpose:

Perform binary OR of two or more numeric expressions

Arguments:

<**expN1...expNn**> are numeric constants or variables which should be binary OR-ed. You may alternatively specify the number as hexadecimal string in the syntax "FF...FF" or "0xFF..FF" with max. 8 hex chars.

Returns:

<**retN**> is the product of the OR operation.

Description:

Binary OR set the resulting bit on (1) if the same bit is on (1) in any of the arguments. It is 0 only is this bit is 0 in all of the arguments. The functionality is equivalent to C syntax

```
iRet = iExp1 | iExp2 | iExp3 |...
```

It is often used to set bits in <expN1> by corresponding bits set in <expN2> mask.

Example:

```
? BinOr(0, 0)                       // 0
? BinOr(0, 1)                       // 1
? BinOr(1, 0)                       // 1
? BinOr(1, 1)                       // 1
? BinOr(1, 0, 1)                    // 1 : 1 OR 0 = 1 OR 1 = 1

? BinOr(6000, 255)                  // 6143
? BinOr(6000, 250, 4, 1)            // 6143
? BinOr("0x1770", "FA", 4, 1, 0)    // 6143 = 0x17FF
? x := BinOr("0xABCD12", "0xFF00")  // 11271954
? Num2hex(x,,.T.)                   // "0xABFF12"
? BinOr("0xABCDEG", 14)             // NIL (error, invalid hex)
```

Classification:

programming

Compatibility:

New in FS5

Related:

BinAND(), BinXOR(), Hex2num(), Num2hex() FS2:IsBit(), FS2:SetBit(), FS2:ClearBit(), FS2:NumOr(), FS2:CharOr()

BINXOR ()

Syntax:

`retN = BinXor (expNC1, expNC2, [...expNCn])`

Purpose:

Perform binary Exclusive-OR of two or more numeric expressions

Arguments:

<**expN1...expNn**> are numeric constants or variables which should be binary XOR-ed. You may alternatively specify the number as hexadecimal string in the syntax "FF...FF" or "0xFF..FF" with max. 8 hex chars.

Returns:

<**retN**> is the product of the XOR operation.

Description:

Binary Exclusive-OR set the resulting bit on (1) if the same bits in the arguments are different (1 and 0 or 0 and 1), and off (0) if the corresponding bits in arguments are equal (both 1 or both 0). The X-OR operation is reversible, when the same mask is used. The functionality is equivalent to C syntax

```
iRet = (((iExp1 ^ iExp2) ^ iExp3) ^ ...)
```

Example:

```
? BinXor(0, 0)                    // 0
? BinXor(0, 1)                    // 1
? BinXor(1, 0)                    // 1
? BinXor(1, 1)                    // 0
? BinXor(1, 0, 1)                 // 0 : 1 XOR 0 = 1 XOR 1 = 0

? res := BinXor("0x2222", "0x101") // 8995 = 0x2323
? BinXor(res, "0x101")            // 8738 = 0x2222
? BinXor(res, "0x2222")           //  257 = 0x101

? BinXor(123456, 1)               // 123457
```

Classification:

programming

Compatibility:

New in FS5

Related:

BinAND(), BinOR(), Hex2num(), Num2hex() FS2:IsBit(), FS2:SetBit(), FS2:ClearBit(), FS2:NumXor(), FS2:CharXor(), FS2:Crypt()

BINLSHIFT ()

Syntax:

`retN = BinLshift (expNC1, expN2)`

Purpose:

Perform binary left shift of the numeric expressions

Arguments:

<**expN1**> is a numeric constants or variables which should be binary shifted. You may alternatively specify the number as hexadecimal string in the syntax "FF...FF" or "0xFF..FF" with max. 8 hex chars.

<**expN2**> is a numeric constants or variables specifying the number of shifted bits. The valid range is 1 to 31.

Returns:

<**retN**> is the product of the shift operation.

Description:

Left shift by one bit is equivalent to multiplying by 2, shift by two bits multiplying by 4 etc. but usually faster. You may see the <exp2> as n-times power of 2 multiply <expN1>. The functionality is equivalent to C syntax

```
iRet = iExp1 << iExp2
```

Example:

```
? BinLShift(1, 1)               // 2           = 0x02
? BinLShift(2, 8)               // 512         = 0x200
? BinLShift(1, 31)              // -2147483648 = 0x80000000
? BinLShift("0x8001", 1)        // 65538       = 0x010002
? BinLShift("0xFF01", 8)        // 16711936    = 0xFF0100
? BinLShift("0x80000001", 1)    // 2           = 0x02
```

Classification:

programming

Compatibility:

New in FS5

Related:

BinAND(), BinOR(), BinXOR(), BinRshift(), Hex2num(), Num2hex() FS2:IsBit(), FS2:SetBit(), FS2:NumRol(), FS2:ClearBit()

BINRSHIFT ()

Syntax:

`retN = BinRshift (expNC1, expN2)`

Purpose:

Perform binary right shift of the numeric expressions

Arguments:

<**expN1**> is a numeric constants or variables which should be binary shifted. You may alternatively specify the number as hexadecimal string in the syntax "FF...FF" or "0xFF..FF" with max. 8 hex chars.

<**expN2**> is a numeric constants or variables specifying the number of shifted bits. The valid range is 1 to 31.

Returns:

<**retN**> is the product of the shift operation.

Description:

Left shift by one bit is equivalent to divide by 2, shift by two bits as divide by 4 etc. but is usually faster. You may see it as <expN1> divide <exp2>times power of 2. The functionality is equivalent to C syntax

```
iRet = iExp1 >> iExp2
```

Example:

```
? BinRShift("0x1F", 1)        // 0x0F
? BinRShift("0x1F", 4)        // 0x01
? BinRShift("0x8FABCD", 4)    // 0x8FABC
? BinRShift("0x8FABCD", 6)    // 0x23EAF

// split 32-bit integer number into bytes

nNum  := 305419896                                  // 0x12345678
byte1 := BinRShift(BinAnd(nNum,"0xFF000000"),24)  //  18 = 0x12
byte2 := BinRShift(BinAnd(nNum,"0x00FF0000"),16)  //  52 = 0x34
byte3 := BinRShift(BinAnd(nNum,"0x0000FF00"),8)   //  86 = 0x56
byte4 := BinAnd(nNum,"0xFF")                      // 120 = 0x78

? Num2hex(nNum,,.T.), "=", Num2hex(byte1,,.T.), ;
  Num2hex(byte2,,.T.), Num2hex(byte3,,.T.), Num2hex(byte4,,.T.)
```

Classification:

programming

Compatibility:

New in FS5

Related:

BinAND(), BinOR(), BinXOR(), BinLshift(), Hex2num(), Num2hex() FS2:IsBit(), FS2:SetBit(), FS2:ClearBit(), FS2:NumRol()

BETWEEN ()

Syntax:

`retL = Between (expNDC1, expNDC2, expNDC3)`

Purpose:

checks if the expression is >= minimum and <= maximum

Arguments:

<**expNDC1**> is a numeric, date or character constant or variable which should be checked to fit in boundary.

<**expNDC2**> is a numeric, date or character constant or variable but of the same type as <expNDC1> corresponding the lower boundary.

<**expNDC3**> is a numeric, date or character constant or variable but of the same type as <expNDC1> corresponding the upper boundary.

Returns:

<**retL**> is .T. if the <expNDC1> is greater or equal to <expNDC2> and lower or equal to <expNDC3>, i.e. if it is in the given boundary. If not, .F. is returned.

Description:

The function is comparable to the operation

```
retL := expNDC1 >= expNDC2 .and. expNDC1 <= expNDC3
```

Example:

```
? Between(10,  5, 15)             // .T.
? Between(10, 11, 15)             // .F.
? Between("abc", "ab", "ad")      // .T.
? Between("Abc", "ab", "ad")      // .F.
```

Classification:

programming

Compatibility:

New in FS5, compatible to FoxPro

Related:

Min(), Max(), MinMax(), <= and >= operator

BOF ()

Syntax:

`retL = BOF ()`

Purpose:

Detects if there was an attempt to move past the beginning of the current database file.

Returns:

<**retL**> is TRUE (.T.) after an attempt to move the record pointer backward past the first logical record using SKIP -n; otherwise, it returns FALSE.

If the database file is empty, or all records have been filtered out, both BOF() and EOF() return .T.

Description:

BOF () is used to test for a boundary condition when the database record pointer moves backward using the SKIP command. It can also be used after SET FILTER condition and GOTO TOP to determine if all the records have been filtered out.

Once BOF() is set to TRUE, it retains its value until there is another attempt to move the record pointer in the corresponding working area.

By default, BOF() operates on the currently selected working area. To determine the status of an unselected working area, use alias->(BOF()); see also LNG.4.4.

Only the SKIP command, and GOTO TOP, BOTTOM (on empty or filtered- out database) and GOTO n (where n < 1 or n > LASTREC()) can set BOF() to TRUE.

The inverse operation of BOF () is EOF ().

Example:

```
USE article
GO TOP
? RECNO(), BOF()                        && 1  .F.
SKIP -1
? RECNO(), BOF()                        && 1  .T.
```

Classification:

database

Related:

SKIP, GOTO, EOF(), LASTREC(), SET FILTER, oRdd:Bof

BREAK ()

Syntax:

`NIL = BREAK (exp)`

Purpose:

Branches out from a BEGIN SEQUENCE...END construct. This is equivalent to the command BREAK [WITH exp].

Arguments:

<**exp**> is the value passed to the RECOVER clause, if any. Note that <exp> is not optional. NIL may be specified if there is no break value.

Returns:

The function always returns NIL.

Description:

The BREAK() function is identical in functionality to the BREAK statement. It must be executed during the BEGIN SEQUENCE...END construct; otherwise, a run-time error occurs. See more in (CMD) BEGIN SEQUENCE and LNG.2.5.3. With the ISBEGSEQ() function, you may check, if a BREAK is possible.

Example:

```
olderr := ERRORBLOCK ( {|par| BREAK (par) } )
:
BEGIN SEQUENCE
:
RECOVER USING acterr
:
END
ERRORBLOCK (olderr)
```

Classification:

programming

Related:

BEGIN SEQUENCE...END, ISBEGSEQ()

BROWSE ()

Syntax:

```
retL = BROWSE ([expN1], [expN2], [expN3], [expN4],
              [expA5], [expL6])
```

Purpose:

Screen oriented display and modification of the currently selected database.

Options:

<**expN1**>...<**expN4**> are the window coordinates: top row, left column, bottom row, right column. If not specified, the default window coordinates are 1, 0, MAXROW(), MAXCOL(). The GUI coordinates are internally calculated in pixel from current SET FONT (or oApplic: Font)

<**expA5**> is an array of character expressions specifying the color setting of BROWSE() elements:

expA5[1]	color of the browse display and bar
expA5[2]	color of the BROWSE() border
expA5[3]	color for field editing via GET/READ
expA5[4]	color of the status display in header
expA5[5]	color of a MEMO field display and edit
expA5[6]	color of the MEMO editing border

If the array is not given, or the array element is empty or NIL, the current SETCOLOR() is used.

<**expL6**> specifies, if the "Abort Browse?" prompt should be displayed (TRUE, the default) or should not.

Returns:

<**retL**> is set to FALSE if no database is in USE, TRUE otherwise.

Description:

BROWSE() is a user-interface function that invokes a general purpose table-oriented browser and editor for records in the current working area.

Since BROWSE() uses the standard function DBEDIT() with an UDF, the navigation keys of DBEDIT() are also valid for the BROWSE() function.

BROWSE() supports following modes:

- **Browsing:** This is the default mode of BROWSE(). Pressing any DBEDIT() navigation key (e.g. cursor left/right/up/down or PgUp, PgDn) moves the high-light bar to a new column or row.
- **Field edit:** Pressing RETURN on any field enters field edit using @..GET/READ. Pressing RETURN terminates the edit mode, saving the changes. ESC terminates without saving changes.

- **Append:** going to the last record (e.g. with Ctrl-PgDn) and then pressing cursor down enters append mode with the message indicating <new> on the status line. A new blank record is then inserted. Pressing cursor up terminates the append mode, saving the new record if data has been entered. If no data has been entered, the new record is not saved.

BROWSE() displays the current status in the upper right corner of the browse window:

Message	Meaning
<new>	Append mode
<bof>	Top-of-file
<delete>	Current record is deleted
Record	Record number display

Example 1:

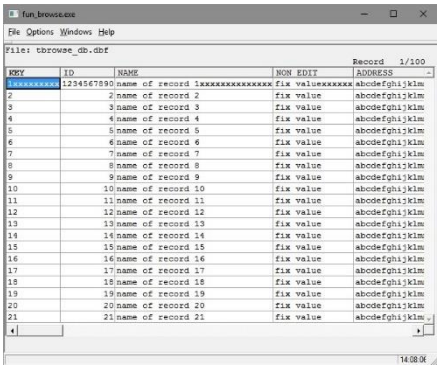
```
USE address INDEX adr1, adr2 SHARED NEW
SET COLOR TO "W+/N,N/W"
@ 1,0 say "use cursor keys to browse, ESC to quit"
BROWSE (2)             // start at line 2
```

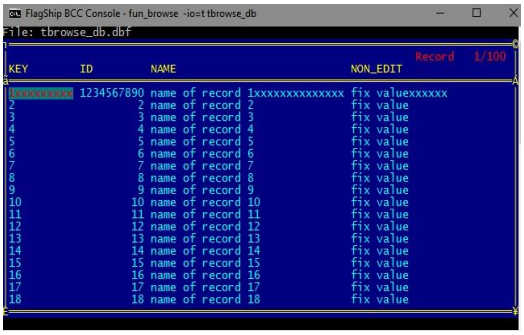
Example 2:

The simplest, complete program for database maintenance. See also examples in chapter LNG.10 and the file <FlagShip_dir>/examples/mydbu.prg

```
SET FONT "Courier", 10         // set default font (GUI only)
oApplic:Resize(25,80,,.T.)     // resize window

PARAMETER dbname
if empty(dbname)
  quit
endif
USE (dbname) SHARED
if !used()
  Alert("File " + dbname + " not available")
  quit
endif
@ 0,0 say "File: " + dbf()
BROWSE (1, 0, maxrow()-1, maxcol(),                          ;
        if(iscolor(), {"GB+/B,R+/BG", "GR+/B",               ;
                       "W+/N,N/W",    "R+/B",                ;
                       "W+/N",        "G/N"  }, NIL), .T.)
```

Output:





Classification:

database

Compatibility:

The 5th and 6th parameters are available in FlagShip only.

Related:

DBEDIT(), TbrowseDb (), TbrowseArr(), OBJ.Tbrowse

CDOW ()

Syntax:

`retC = CDOW ( [expD] )`

Purpose:

Converts a date value to a character day of the week.

Arguments:

<expD> is the date value. If not specified, the current system date() is used.

Returns:

<**retC**> is a character string, containing the name of the day of the week (e.g. "Monday" or "Montag").

Description:

CDOW() can be used to enhance reports, printouts and screen displays.

In FlagShip, the names for foreign human languages can be defined in an ASCII file (e.g. FSsortab.ger) and activated when required with FS_SET("loadlang") and FS_SET ("setlang"). By default, English names are in lowercase; the first letter is in uppercase.

If invalid date is specified, a null string is returned.

The alternative Cdow2() function available in the FS2 Toolbox allows you to define month names independent on the current language setting via FS_SET("setlang"|"loadlang").

Example:

```
? DATE()                                  && 08/23/93
? CDOW(DATE())                            && Monday
? CDOW(DATE() + 3)                        && Thursday
? CDOW(CTOD("08/20/83"))                  && Friday

SET DATE GERMAN
if FS_SET ("load", 1, "FSsortab.ger")
   FS_SET ("setLang", 1)                  && german language
endif
? CDOW(CTOD("20.08.93"))                  && Freitag
```

Classification:

programming

Compatibility:

The support for other human languages is available in FlagShip only. Clipper requires the <expD> value, it does not use current date() when <expD1> parameter is empty.

Related:

CMONTH(), CTOD(), DATE(), DAY(), DOW(), DTOC(), DTOS(), FS_SET(), MONTH(), YEAR(), FS2:Cdow2(), FS2:CtoDow(), FS2:MDY(), FS2:DMY()

CHECKBOX ()

Syntax:

oRet := CheckBox ([expN1], [expN2], [expC3], [expL4])

Purpose:

Creates CheckBox, same as instantiating the object via CheckBox{…} class.

Options:

<**expN1**> is a vertical coordinate (row) of the upper left edge. Either in pixels or col/row coordinates, dependent on the current state of SET PIXEL on|OFF or the <expL6> parameter. Modifiable by oRet:Row, default is 0.

<**expN2**> is a horizontal coordinate (column) of the upper left edge. Either in pixels or col/row coordinates, dependent on the current state of SET PIXEL on|OFF or the <expL6> parameter. Modifiable by oRet:Col, default is 0.

<**expC3**> caption text, optional. If not redefined by :CapCol and/or :CapRow, the text is displayed in the <expN1> row and <expN2> + 4 column.

<**expL4**> if true(.T.), the row and column data are in pixel; if false (.F.), data are in row/col coordinates, otherwise the current SET PIXEL is used.

Returns:

<**oRet**> is an instantiated CheckBox object variable, which instances can be additionally assigned/executed. At least oRet:Show() needs to be executed to display the box.

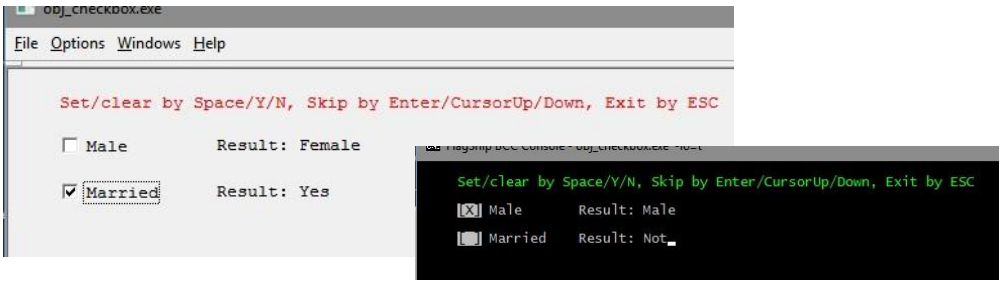
Description:

Creates CheckBox icon to be processed by mouse click (or keyboard) and displays it by subsequent invocation of oRet:Show(). The default handler is named CheckBox-Handler() and is available in source. See other properties in section OBJ.CheckBox

Example 1: In addition to the @...GET CHECKBOX, the CheckBox can also be used stand-alone independent of READ:

```
SET FONT "Courier", 10
@ 1,5 SAY "Set/clear by Space/Y/N, Skip by Enter/CursorUp/Down,"+ ;
          " Exit by ESC" COLOR "G+" GUICOLOR "R+"

oBox1 := CheckBox(3,5,"Male")            // create box#1
oBox1:Handler := {|obj| CheckBoxHandler(obj)}  // checkboxhand.prg
oBox1:Display()                          // yet display only

oBox2 := CheckBox(5,5)                   // create box#2
oBox2:Caption := "Married"
oBox2:Handler := {|obj| myHandler(obj)}
oBox2:Display()                          // yet display only

while lastkey() != K_ESC                 // handle both boxes
  oBox1:Show()                        // process via default handler
  @ oBox1:Row, oBox1:Col+15 SAY "Result: " + ;
```

```
                                if(oBox1:Value, "Male  ", "Female")
  if lastkey() == K_ESC
     exit
  endif
  oBox2:Show()                          // process via own handler
  if lastkey() == K_UP
     loop
  else
   oBox2:Show()
  endif
enddo

setpos(8,0)
wait "Done ..."

FUNCTION MyHandler(obj) // simplified CheckBoxHandler(obj)
LOCAL key := 0
obj:SetFocus()
while .T.
  key := inkey(0)
  do case
  case key == K_SPACE
    obj:Select(!obj:Buffer)     // toggle on/off & stay
  case chr(key) $ "+yYtT"
    obj:Select(.T.)             // set on & stay
  case chr(key) $ "-nNfF"
    obj:Select(.F.)             // set off & stay
  otherwise
    if key > 0                  // exit loop otherwise
      exit
    endif
  endcase
  @ obj:Row, obj:Col+15 SAY "Result: " + if(obj:Value,"Yes","Not")
enddo
obj:KillFocus()
return
```



Example 2:

See additional examples in section OBJ.CheckBox and CMD.@...GET CHECKBOX

Classification: programming
Compatibility: available in VFS8 and newer
Related: OBJ.CheckBox, @...GET CHECKBOX

CHR ()

Syntax:

`retC = CHR (expN, [expN2], [expN3, ... ])`

Purpose:

Converts one or more ASCII code(s) to a character.

Arguments:

<**expN**> is a number in the range of one (zero) to 255, representing the ASCII code. Negative values are supported for the FN key equivalents; see the INKEY() table.

If more than one argument are specified, the following arguments are coded by the same way and added to the resulting string.

Returns:

<**retC**> is a single character, corresponding to the given ASCII value. If <expN> is negative, a two-byte code of the corresponding FN key is returned.

If more than one argument is specified, the resulting string is a concatenation of each argument.

Description:

CHR() can be used to display graphic characters on the screen, to send control codes to the printer, to fill the keyboard type-ahead buffer simulating user input, etc. The inverse operation of CHR() is ASC().

Example 1:

Prints the ASCII table to the screen. Note: check the correct terminal setting for TERM=FSxxxx ; export TERM

```
FOR i = 0 TO 15
   FOR j = 0 TO 15
      code = i*16 + j
      @ i, 5*j   SAY CHR(code)
      @ i, 5*j+2 SAY code PICTURE "999"
   NEXT
NEXT
```

Example 2:

The resulting variables "x" and "y" are equivalent = “ABC”

```
x := chr(65) + chr(66) + chr(67)
y := chr(65, 66, 67)
```

Example 3:

Simulate F2 + Return key depress:

```
#include "inkey.fh"
KEYBOARD CHR(K_F2) + CHR(K_RETURN)
DO WHILE (key := INKEY()) != 0
   IF key = -1     ; ? "F2 key"             // K_F2
   ELSEIF key = 13 ; ? "Return"             // K_RETURN
   ELSE            ; ? "Key:", key
```

```
   ENDIF
ENDDO
```

Classification:

programming

Compatibility:

The support of FN keys is available in FlagShip only. To support the CHR(0), FS_SET("zerobyte",.T.) must be set, see LNG.2.6.5. More than one arguments is supported in FS5 only.

Include:

Predefined function key constants are available in "inkey.fh"

Related:

KEYBOARD, ASC(), INKEY(), FS_SET()

CHR2SCREEN ()

Syntax:

`retS = CHR2SCREEN (expC)`

Purpose:

Converts a character string to a screen variable. This function is applicable for Terminal i/o mode only.

Arguments:

<**expC**> is a string, generated with SCREEN2CHR()

Returns:

<**retS**> is a screen variable of the type "S".

Description:

CHR2SCREEN() can be used to restore the screen constants from a database or .dbt field, text file etc. previously stored by using the SCREEN2CHR() function.

The inverse operation of CHR2SCREEN() is SCREEN2CHR().

If you need to transfer a DOS stored screen contents to FlagShip, use the SCRDOS2UNIX() function.

GUI mode: The Chr2Screen() **cannot** be used in GUI mode, where the screen variable <varS> is compressed or uncompressed bitmap object an hence cannot be extracted or converted by this function. You may save GUI (and terminal) screen to .dbv file directly, or to .dbt via MemoCode(), e.g.

```
varS := SaveScreen()
REPLACE MyMemo WITH MemoCode(varS)
...
RestScreen( MemoEncode(MyMemo) )
```

Example:

Save and restore the screen contents in a .dbt field (terminal i/o):

```
LOCAL my_screen
FIELD memofield
my_screen = SAVESCREEN (10,5, 20,40)
? VALTYPE(my_screen)                          // "S"
:
USE initdata ALIAS init
REPLACE memofield WITH CHR2SCREEN (my_screen)
:
RESTSCREEN (10,5, 20,40, SCREEN2CHR (memofield))
```

Classification:

programming

Compatibility:

In FlagShip, the storage resulting from SAVESCREEN() or SAVE SCREEN TO is of type screen ("S"). To manipulate it, see the _retscw() function in the EXT section. The type

of the "S" variable is incompatible with the DOS/Clipper screen storage in variables of type "C", but may be transferred vice-versa by using SCRUNIX2DOS() and SCRDOS2UNIX() functions.

By default, FlagShip supports direct saving and restoring of screen variables in/from .mem files; see also FS_SET ("memcompat").

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode, see SaveScreen() for details.

Related:

SCREEN2CHR(), SCRDOS2UNIX(), SCRUNIX2DOS(), SAVESCREEN(), RESTSCREEN(), SAVE TO, RESTORE FROM, FS_SET ("memcompat")

CMONTH ()

Syntax:

`retC = CMONTH ( [expD] )`

Purpose:

Converts a date value to a character month.

Arguments:

<expD> is the date value. If not specified, the current system date() is used.

Returns:

<**retC**> is a character string, containing the name of the month (e.g. "March" or "M„rz").

Description:

CMONTH() can be used to enhance reports, printouts and screen displays.

In FlagShip, the names for foreign human languages can be defined in an ASCII file (e.g. FSsortab.ger) and activated when required by FS_SET("loadlang") and FS_SET ("setlang"). By default, the English names are in lowercase; the first letter is in uppercase. If invalid date is specified, a null string is returned.

The alternative Cmonth2() function available in the FS2 Toolbox allows you to define month names independent on the current language setting via FS_SET("setlang" | "loadlang").

Example:

```
? DATE()                                        && 08/23/93
? CMONTH(DATE())                                && August
? CMONTH(CTOD("03/20/94"))                      && March
SET CENTURY ON
today := DATE()
? "Today is " + CDOW(today) + ", " ;
  CMONTH(today) + " " + LTRIM(STR(DAY(today))) + ;
  ", " + LTRIM(STR(YEAR(today)))

SET DATE GERMAN
? DATE()                                        && 23.08.1993
if FS_SET ("load", 1, "FSsortab.ger")
   FS_SET ("setLang", 1)                        && german language
endif
? CMONTH(DATE())                                && August
? CMONTH(CTOD("20.03.1994"))                    && Maerz
```

Classification: programming

Compatibility: The support for other human languages is available in FlagShip only. Clipper requires the <expD> value, it does not use current date() when <expD1> parameter is empty

Related: CDOW(), CTOD(), DATE(), DAY(), DOW(), DTOC(), DTOS(), FS_SET(), MONTH(), YEAR(), FS2:Cmonth2(), FS2:CtoMonth(), FS2:MDY(), FS2:DMY()

COL ()

Syntax:

`retN = COL ([expL1], [expN2])`

Purpose:

Reports the current column of the cursor on the screen.

Options:

<**expL1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col, otherwise the current SET PIXEL status is used.

<**expN2**> is optional numeric value to change the Col() value.

Returns:

<**retN**> is a numeric value in the range 0 to MAXCOL(), representing the current cursor column. When the global variable `_aGlobSetting[GSET_L_ROUNDROWCOL]` is set .T. (default is .F., see initio.prg), the returned <retN> value is rounded to integer. When SET COORD UNIT is set to PIXEL, MM, CM or INCH, and <expL1> is not given, the returned value corresponds to these units (GUI only). If not set, the GUI column coordinate corresponds to current SET FONT (or oApplic:Font)

Description:

The value of COL() changes whenever the cursor position changes on the screen. Both console and full-screen commands can change the cursor position. The COL() value is reset to zero whenever a CLEAR, CLEAR SCREEN, or CLS command or SETPOS(0,0) function is executed.

The COL() and ROW() values are updated only if SET DEVICE TO SCREEN (the default) is set. On SET DEVICE TO PRINTER, the PROW() and PCOL() values are updated instead.

COL() can be used to position the cursor relative to previous output. Usually, the COL() is used in combination with the ROW() function.

If you are using sub-windows via standard MDIopen() or the Wopen() from FS2 Toolbox, Row() and Col() reports the cursor position in the current sub-window.

In GUI mode, any output is pixel oriented. The COL(.T.) return value therefore corresponds to the real display position, i.e. where the next output displays. For your convenience and to achieve cross compatibility to textual based applications, FlagShip supports also coordinates in common row/col values. When <expL1> is not given and SET PIXEL is OFF, and SET COORD UNIT is not set, the returned COL() value corresponds to manual Pixel2col(Col(.T.)) calculation.

When you change the font size or select different font name using SET FONT command or the FONT class, the current COL() position remain unchanged, but the next line and COL() coordinate may differ from the current setting. See further details in LNG.5.3, LNG.5.3.1, LNG.5.3.2, Col2Pixel() and SET FONT.

In Terminal and Basic i/o mode, the column width is fix 1, independent on the used font, hence COL() and COL(.T.) returns the same value in row/col coordinates.

Example 1:

```
@  10,   5    SAY "Flag"
@ ROW(),COL() SAY "Ship"                // Result is: FlagShip
y := ROW()
x := COL()
? y, x                                  // 10  13
```

Example 2:

```
SET FONT "helvetica", 12
@ 0,0 say "     Hello" ; ?? COL()       // text columns differs
@ 1,5 say "Hello" ; ?? COL()            // with proportional font
SET FONT "courier", 12
@ 2,0 say "     Hello" ; ?? COL()       // text columns aligns
@ 3,5 say "Hello" ; ?? COL()            // with fixed font
```

Classification:

programming, for screen oriented output

The return value is usable only with enabled screen oriented output facility (the default, see sect. SYS.2.7).

Compatibility:

The optional parameter is new in FS5

Related:

ROW(), PCOL(), PROW(), @..SAY, MAXCOL(), MAXROW()

COL2PIXEL ()

Syntax:

```
retN = Col2pixel ([expN1], [expO2])
```

Purpose:

calculates pixels of given column corresponding to the given or current i/o Font.

Arguments:

<**expN1**> is a numeric constant or variable specifying the column size which should be recalculated to pixels. If not given or < 0, the pixel size of one column is returned.

<**expO2**> is a Font object to be used for the pixel calculation. If not specified, the default font set by SET FONT (or oApplic:Font) is used.

Returns:

<**retN**> is an integer value corresponding to the width of given columns in pixels.

Description:

In GUI mode, any output is pixel oriented. For your convenience and to achieve cross compatibility to textual based applications, FlagShip supports also coordinates in common row/column values. It then internally re-calculates the given rows by using Row2pixel() and columns by using Col2pixel() function. The line and character spacing is affected by the currently used default font.

This function allows you to manually calculate the X pixel position from given row coordinate. It is applicable for COL(), SETPOS() and other positioning commands and functions. The return value is equivalent to return of oApplic:Col2Pixel(expN1) method.

In Terminal and Basic i/o mode, the col width is fix 1 independent on the used font and hence Col2pixel(expN1) returns <expN1>.

See further details about font and coordinates in LNG.5.3, LNG.5.4, Col() and SET FONT. For minimal porting effort, best to use fixed fonts (e.g. SET FONT "Courier", 12).

Classification:

programming

Compatibility:

New in FS5

Related:

Row2pixel(), Pixel2col(), Pixel2row(), oApplic:Col2Pixel()

COLVISIBLE ()

Syntax:

`retN = ColVisible ([expL1])`

Purpose:

Reports the currently visible columns on the screen.

Options:

<**expL1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col, otherwise the current SET PIXEL status is used.

Returns:

<**retN**> is a numeric value, representing the currently visible columns in the range between zero and MAXCOL(). If <expL1> is not set, the GUI coordinate is internally calculated from current SET FONT (or oApplic:Font)

Description:

COLVISIBLE() returns the currently visible columns in the range of zero to MAXCOL(). In GUI mode, it considers the current/resized window size. In Terminal or Basic mode, the value is equivalent to MAXCOL().

ColVisible() is equivalent to m->oAppWindow:CurrSize(APP_X_USER)

Example:

```
? "MaxCol()=",ltrim(MaxCol(),"ColVisible()=",ltrim(ColVisible())
if AppIoMode() == "G"
   wait "Resize the screen width by mouse, then press any key..."
endif
? "MaxCol()=",ltrim(MaxCol(),"ColVisible()=",ltrim(ColVisible())
wait
```

Classification:

programming, for screen oriented output

Compatibility:

ColVisible() is new in FS6

Related:

COL(), MAXCOL(), RowVisible(), oAppWindow:CurrSize()

COLOR2RGB ()

Syntax:

`retA = Color2rgb (expCAO1, [expA2])`

Purpose:

transform color string or color object to array of RGB triplets

Arguments:

<**expCAO1**> is a color string, Color object or an array of colors. The array may be 1-dimensional with 3 rgb tokens, or 2-dimensional with tokens for foreground and background, or 3-dimensional for standard, enhanced etc. color pairs.

<**expA2**> is a an array specifying default colors, either 1, 2 or 3-dimensional.

Returns:

<**retA**> is 3-dimensional array [6,2,3] with translated color of <expCAO1> to tokens.

Classification:

programming, almost internally used

Compatibility:

New in FS5

Related:

COLORSELECT ()

Syntax:

`retCA = COLORSELECT (expN)`

Purpose:

Activate attribute from the current color setting (terminal i/o only).

Arguments:

<**expN**> is a number corresponding to the ordinal position in the current list of color attributes, as set by SETCOLOR().

expN	"color.fh" constant	Equivalent to command
0	CLR_STANDARD	SETSTANDARD
1	CLR_ENHANCED	SETENHANCED
2	CLR_BORDER	n/a
3	CLR_BACKGROUND	n/a
4	CLR_UNSELECTED	SETUNSELECTED
5	CLR_EXTRA	n/a
6	CLR_DISABLED	n/a
7	CLR_INACTIVEWIND	n/a

Returns:

<**retCA**> is the selected color pair. In terminal i/o mode <retAC> is a string "Foregr/Backgr", in GUI mode an array {aForegr,aBackgr} of RGB colors.

Description:

COLORSELEC() activates the specified color pair from the current list of color attributes, established by SET COLOR TO or SETCOLOR(). COLORSELECT() is similar to the above listed SETxxx commands.

The current color string remains unchanged.

Example:

```
#include "color.fh"
SET COLOR TO "+W/B, R/G, W/N, N/W, R+/B, R/N, G/N, BG+/N"
? SETCOLOR()                            // W+/B,R/G...
?
COLORSELECT (CLR_ENHANCED)              // or: SETENHANCED
? "the output is red on green"
COLORSELECT (CLR_UNSELECTED)            // or: SETUNSELECTED
? "the output is bright red on blue"
COLORSELECT (CLR_STANDARD)              // or: SETSTANDARD
? "the default output is bright white on blue"
?
ColorSelect(0) ; ? "CLR_STANDARD     =", ltrim(CLR_STANDARD    )
ColorSelect(1) ; ? "CLR_ENHANCED     =", ltrim(CLR_ENHANCED    )
ColorSelect(2) ; ? "CLR_BORDER       =", ltrim(CLR_BORDER      )
ColorSelect(3) ; ? "CLR_BACKGROUND   =", ltrim(CLR_BACKGROUND  )
ColorSelect(4) ; ? "CLR_UNSELECTED   =", ltrim(CLR_UNSELECTED  )
ColorSelect(5) ; ? "CLR_EXTRA        =", ltrim(CLR_EXTRA       )
```

```
ColorSelect(6) ; ? "CLR_DISABLED     =", ltrim(CLR_DISABLED    )
ColorSelect(7) ; ? "CLR_INACTIVEWIND =", ltrim(CLR_INACTIVEWIND)

SETSTANDARD
wait
```

Output:

Classification:

programming

Compatibility:

Available in FS and C5 only.

Include:

The color constants are available in "color.fh"

Related:

SET COLOR, SETCOLOR(), SETSTANDARD, SETENHANCED, SETUNSELECTED

CONSOLEOPEN ()

Syntax:

`retL = CONSOLEOPEN ( )`

Purpose:

Open console window, if required. Applicable only in GUI mode for MS-Windows, ignored in Unix/Linux. In Terminal i/o mode, only the addition console is displayed, except a redirection to a file is used (e.g. `myapplic.exe 2>myapplic.txt`).

Returns:

The function returns .T. when the console window was newly opened and .F. if console is already open or could not be created.

Description:

This ConsoleOpen() is designed for MS-Windows environment. It is supported for compatibility purposes in Unix/Linux, but ignored.

When the GUI application is compiled without **-io=g** or with **-io=a** switch, it automatically opens temporary console window. If you wish to open this console window manually, use this ConsoleOpen() function. Such console window is used in GUI mode to display output directed to stdout or stderr (e.g. via ?#, ??## commands or OutStd(), OutErr() functions) or from C UDFs using printf(), fprintf(stdout,...) or fprintf(stderr,...).

You may safely call ConsoleOpen() as often as you want, since only the first invocation opens the temporary window, all subsequent calls are ignored and returns .F.

MS-Windows use random position on the desktop when creating new console window. You may set it position and behavior either at start-up using the FSCONSOLE environment variable (see section FSC), or at run-time using the ConsoleSize() function.

To be able to view the last output, a newly created console window remains open 10 seconds after the application terminates. You may re-define this timeout period by assigning

```
_aGlobSetting[GSET_N_WAITCLOSEWIND] := 10   // the default
```

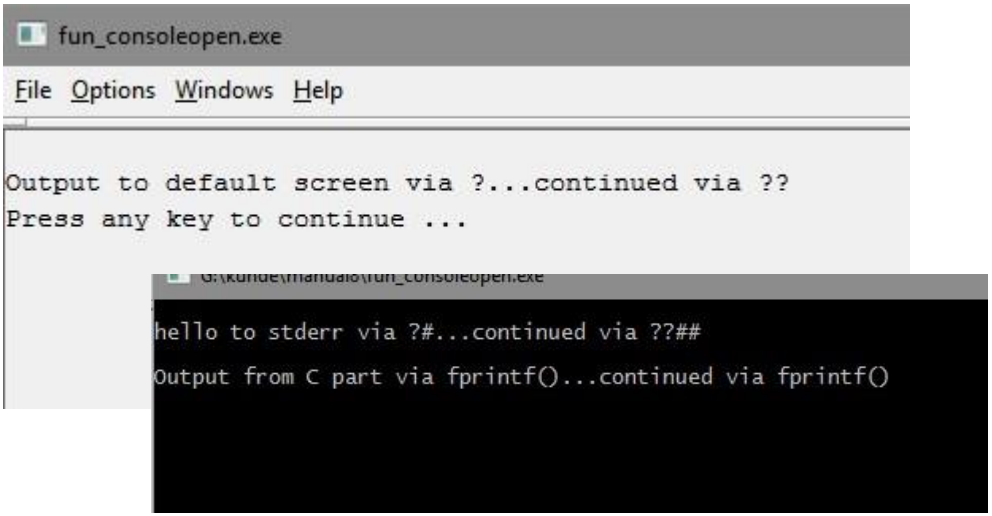
or disable the delay at all by setting this value to 0. To avoid delays in CGI scripts, the delay is disabled (i.e. assumed 0) when the application was compiled by using the `-io=b` switch.

If you wish to ensure the console window is open in a C part for an output to stdout or stderr, you may invoke the equivalent int OpenConsole() function directly from C. Otherwise, when no console is open, MS-Windows will simply swallow such output.

Example:

```
SET FONT "Courier", 10       // set default font (GUI only)
oApplic:Resize(25,80,,.T.)   // resize window

ConsoleOpen()                // usually not required
```

```
ConsoleSize( {-1, -1} )      // center the console on desktop
?# "hello to stderr via ?#"
??## "...continued via ??##"

#Cinline
{
   OpenConsole();       /* required if console is not open yet */
   fprintf(stderr,"\nOutput from C part via fprintf()");
   fprintf(stderr,"...continued via fprintf()\n");
}
#endCinline
? "Output to default screen via ?"
?? "...continued via ??"
wait
```



Classification:

programming

Compatibility:

Available in VFS6 and newer.

Related:

ConsoleSize(), FSCONSOLE envir. variable

CONSOLESIZE ()

Syntax:

`retA = CONSOLESIZE ( [expA] )`

Purpose:

Retrieve or and/or set the size of console window.

Options:

<expA> is optional array containing one or more numeric elements containing the new value to be set. Invalid or non-numeric element values, i.e. NIL or numbers lower than -1 for expA[1..2] or lower then 1 for expA[3..n] are ignored, same as expA[7..n] elements. In Unix/Linux, all expA[1..n] input values are ignored.

expA[n]	Console coordinates	Accepted input
1	x-left coordinate in pixel	0..n, -1 = center horiz
2	y-top coordinate in pixel	0..n, -1 = center vert
3	x-right coordinate in pixel	1..n
4	y-bottom coordinate in pixel	1..n
5	max columns	retA[5]..n columns
6	max rows	retA[6]..n rows

Returns:

<retA> is an array of numeric elements describing the currently used console window and desktop size. The console is either the cmd.exe console if Terminal based application is running from it, or it is the temporary console window created via ConsoleOpen().

retA[n]	Console, Desktop	Returned value,equiv.to
1	x-left cons coord in pixel	0..n MS-Windows only
2	y-top cons coord in pixel	0..n MS-Windows only
3	x-right cons coord in pixel	1..n MS-Windows only
4	y-bottom cons coord in pixel	1..n MS-Windows only
5	max console columns	1..n MS-Windows only
6	max console rows	1..n MS-Windows only
7	terminfo columns	0..n envir.var COLUMNS
8	terminfo rows	0..n envir.var LINES
9	desktop left x coord pixel	0..n usually 0
10	desktop right x coord pixel	0..n oApplic:DesktopWidth
11	desktop top y coord pixel	0..n usually 0
12	desktop bottom y coord pixel	0..n oApplic:DesktopHeight

Not available or not determinable values (means e.g. desktop values for non GUI environment) are set -2. In Unix/Linux, only values in retA[7..12] are filled, when available.

The <retA> value represents the resulting values set/modified by the input <expA> if a parameter was passed.

Description:

With ConsoleSize(), you may determine, resize or move the console window used in Terminal based application (-io=t or -io=b), or on console window temporarily created via ConsoleOpen() when an output to stdout or stderr (via ?#, ??## commands or OutStd(), OutErr() functions) was invoked in GUI mode (-io=g). With the environment variable FSCONSOLE (see section FSC), you may set axpA[1,2,5,6] values already at start-up time.

ConsoleSize() is designed mainly for MS-Windows but supported for compatibility purposes in Unix/Linux. The set CMD terminal size (height and/or width) can only be in range (be max) of currently open CMD window size (Properties -> Layout -> Screen Buffer Size), which is however configurable. In Linux, use xterm settings by newfswin script to set terminal size and coordinates.

Example 1:

```
aCons := ConsoleSize()            // determine console/desktop size
_DisplArrStd(aCons,"ConsoleSize") // display array elements
```

Example 2:

```
ok := ConsoleOpen()              // open console window if possible
aCons := ConsoleSize()           // get coordinates
? "orig. x/y console coordinates =", ;
  ltrim(aCons[1]), "/", ltrim(aCons[2]), "pixel"

aCons := ConsoleSize( {-1, -1} )   // center the console on desktop
? "new x/y console coordinates =", ;
  ltrim(aCons[1]), "/", ltrim(aCons[2]), "pixel"
```

Example 3:

```
ConsoleSize( {-1, 10, , , 100} )   // center horiz, set 100 columns
```

Classification:

programming

Compatibility:

Available in FS6 only.

Related:

ConsoleOpen(), FSCONSOLE envir. variable

CP437_UTF8 ()

Syntax:

`retC = CP437_UTF8 ( expC1 )`

Purpose:

Converts string from CP-437 (US-ASCII, PC-8) to Unicode UTF-8.

Arguments:

<**expC1**> is a character string in ASCII PC-8 Codepage CP437 encoding

Returns:

<**retC**> is the converted string into Unicode UTF-8.

Description:

When the i/o font is set to Unicode, e.g. oFont:CharSet(FONT_UNICODE) or oApplic:Font:CharSet(FONT_UNICODE), you may display all characters chr(1..255) same as in charset CP-437 by converting the PC-8 characters to Unicode by this function. The returned <retC> may be up to three times longer than <expC1>.

Note: if you need to display CP-437 special symbols chr(1..31) via @...SAY, you don't need FONT_UNICODE nor CP437_UTF8(), but just set SET GUITRANSL LOWCP437 ON or Set(_SET_GUILOWCP437, .T.)

You may create own translations for another character sets than PC-8/CP-437, see <FlagShip_dir>/system/trans2utf8.prg

Unicode is supported in GUI mode only, see further details in section LNG.5.4.5.

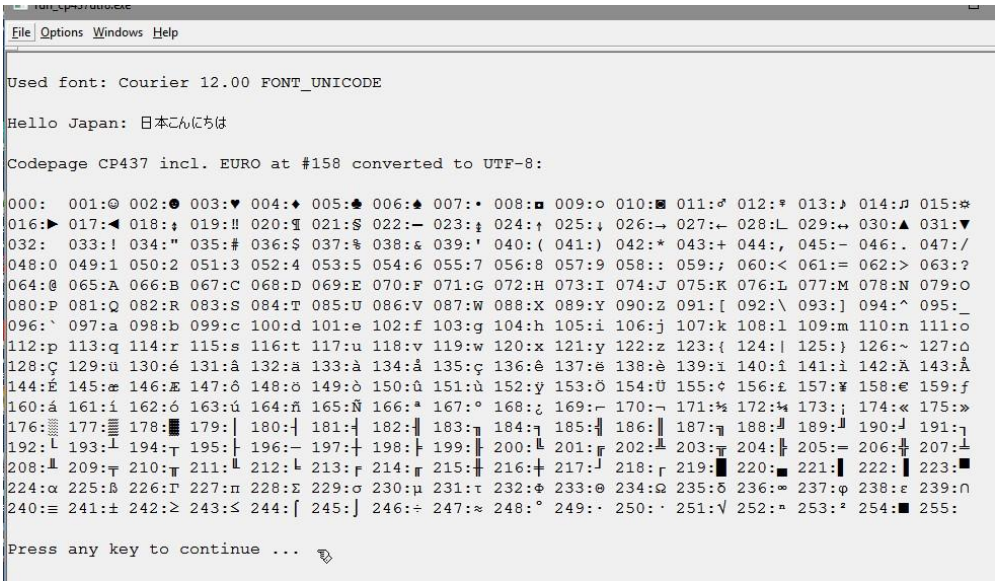
To display Unicode in Linux, you may need to set corresponding Unicode font, e.g. SET FONT "mincho" for Japanese.

Example:

```
// display greeting in Japanese
// and the whole CodePage 437 charset in Unicode
local ii, cc, rr, cHello
#include "font.fh"                    // for FONT_UNICODE
#ifdef FS_LINUX
  SET FONT "mincho", 16               // Unicode font
#else
  SET FONT "Courier", 12              // or other font
#endif
m->oApplic:Resize(30, 100,,.T.)       // resize the screen

// cHello := "Hello Japan: ???????"  // you may enter UTF-8 chars
cHello := "Hello Japan: " + ;
      chr(230,151,165, 230,156,172, 227,129,147, 227,130,147) + ;
      chr(227,129,171, 227,129,161, 227,129,175)

SET GUICHARSET FONT_UNICODE          // output in Unicode
? "Used font:", oApplic:Font:Name, ltrim(oApplic:Font:Size),;
  oApplic:Font:CharSetName()
?
```

```
? cHello                                  // displays Japanese glyphs
?
? "Codepage CP437 incl. EURO at #158 converted to UTF-8:"
rr := row() +2
cc := 0
@ rr,cc say strzero(0,3) + ": "
cc += 6
for ii := 1 to 255                  // display coverted PC-8 chars
   if (ii % 16) == 0 .and. ii > 1
      cc := 0
      rr++
   endif
   @ rr,cc say strzero(ii,3) + ":" + cp437_utf8(chr(ii))
   cc += 6
next
?
wait
```

Output:



Classification:

programming, Unicode input/output in GUI mode

Compatibility:

New in VFS7

Related:

UTF16_UTF8(), LNG.5.4.5, @..SAY, ?, ??, SET GUICHARSET, OBJ.Font

CRC32 ()

Syntax:

`retN = Crc32 (expCA1, [expN2])`

Purpose:

calculates CRC32 value conf.to ANSI X3.66, ADCCP, CCITT X.25

Arguments:

<**expCA1**> is a character string or an array of strings

<**expN2**> is length of the string to be calculated. If not given, the whole string or the array of strings is calculated.

Returns:

<**retN**> is the calculated CRC value of <expCA1>.

Description:

The CRC value can be used for validity checking and for security purposes and for test of communication errors. It uses 32-bit CRC value conformant to ANSI X3.66, ADCCP, CCITT X.25

Example 1:

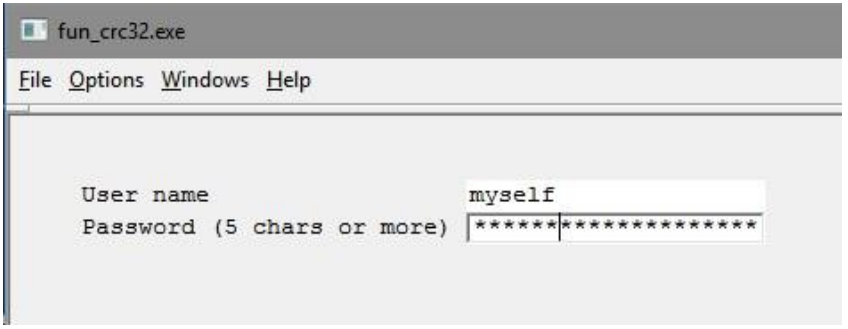
```
cText := "Hello World!"
? "CRC32 from '" + cText + "' is", crc32(cText)   // 472456355
wait
```

Example 2:

Check and/or store user password, where only the password hash is stored, hence it is protected from crack. You additionally may crypt the password via CharXor() for even more security against Brute-Force-Attack.

```
local cUser, cPassw := space(20), nCRC := 0
local cGeneralKey := "My Secret Key"    // modify!
SET FONT "Courier", 10       // set default font (GUI only)
oApplic:Resize(25,80,,.T.)   // resize window

if !file("mypassw.dbf")
  dbcreate("mypassw", {{"User","C",20,0},{"PASSW","N",16,0}})
endif
cUser := padr(substr(NetName(), 1, at("@", Netname()) -1), 20)
while lastkey() != K_ESC
  @ 2,5 say "User name                 " get cUser
  @ 3,5 say "Password (5 chars or more)" get cPassw PICTURE "@P"
  READ
  if len(trim(cPassw)) < 5 .or. lastkey() == K_ESC
     loop
  endif

  nCRC := crc32(cPassw)  // or: crc32(CharXor(cPassw,cGeneralKey))
  USE mypassw
  LOCATE FOR USER == padr(cUser,20)
  if found()
```

```
      if PASSW != nCRC
         alert("Incorrect password")
         loop
      endif
   else
      APPEND BLANK
      REPLACE USER with cUser, PASSW with nCRC
   endif
   exit
enddo
if lastkey() != K_ESC   // test only
   ? "User=",trim(User),"Passw=",trim(cPassw),"Hash=",ltrim(PASSW)
endif
wait
```



Classification:

programming

Compatibility:

New in FS5

Related:

FS2:Crc16(), FS2:CharXor()

CTOD ()

Syntax:

`retD = CTOD (expC)`

Purpose:

Converts a date string to a data value.

Arguments:

<**expC**> is a character string consisting of numbers representing month, day and year separated by any character other than a digit. The order of the parts of the date depends on the current SET DATE value; the default is SET DATE AMERICAN ("mm/dd/yy"). If only two digits are specified for the year, the SET EPOCH value (the default is 1900) is considered.

An empty date can be specified as SPACE(8), null-string "", or " / / ".

Returns:

<**retD**> is the resulting date value. An invalid <expC> leads to an empty date of {00/01/0001} or " / / " being returned.

Description:

CTOD() is used when initializing or comparing a date variable, or to set a date variable or field to a specific value. FlagShip also supports a date constant given in curly brackets, so {08/20/93} is equivalent to CTOD("08/20/93").

Example:

```
LOCAL today := DATE(), newyear := CTOD("01/01/94")
IF today = CTOD("12/31/1999")
   ? "use SET EPOCH 2000 tomorrow !"
ELSEIF today = newyear
   ? "Happy new year!"
ENDIF
SET CENTURY ON
? {02-15-94}                              //  2/15/1994
SET DATE GERMAN
? CTOD("31 10 93")                        // 31.10.1993
? CTOD(SPACE(8)) + 2                      // 02.01.0001
```

Classification:

programming

Compatibility:

The date constant given in curly brackets is not available in C5. FlagShip supports valid date values from 01/01/0001 to 12/31/9999. FS supports embedded zero bytes by default.

Related:

SET DATE, CDOW(), CMONTH(), DATE(), DAY(), DOW(), DTOC(), DTOS(), MONTH(), YEAR(), STOD()

CURDIR ()

Syntax:

retC = CURDIR ([expC1], [expL2])

Purpose:

Returns the current Unix or Windows directory.

Options:

In DOS or MS-Windows, <**expC1**> specifies optional letter of the disk drive (A ... Z) optionally with colon (A: ... Z:), case insensitive. If <expC1> is not specified or is NIL or empty, the directory path of current drive is returned.

In Unix/Linux, FlagShip supports the drive letter by substituting the environment variable ?_FSDRIVE (where ? is the drive letter, like D_FSDRIVE) with a given Unix directory, see LNG.9.5 and FSC.3.3. <expC1> is here case sensitive and must contain colon. If the argument is not specified, or the corresponding environment variable is not exported, the current directory is used.

<**expL2**> is a compatibility switch to Clipper. If set .T., the returned path is without drive letter and without leading backslash, long path names are converted to short names and the result is in upper case (in FlagShip for MS-Windows). In Unix/Linux, the returned path is without the leading slash when <expL2> is .T.

Returns:

<**retC**> is a character string, representing the current Unix directory in upper and lower case. In MS-Windows, it contain fully qualified path including drive, except when <expL2> is .T.

If an error occurred (e.g. access denied), a null string is returned.

Example:

```
#ifdef FS_WIN32
  /*
   * FlagShip for MS-Windows
   */
  ? CURDIR()              && C:\Documents and Settings\John Miller
  ? CURDIR(,.T.)          && DOCUME~2\JOHN~ER
  ? CURDIR("E:")          && E:\server\Common data\source
  ? CURDIR("E:", .T.)     && SERVER\COMMON~1\SOURCE

  SET DIRECTORY TO ("E:\server\Common data")
  ? CURDIR()              && E:\server\Common data
  ? CURDIR("",.T.)        && SERVER\COMMON~1

#else
  /*
   * FlagShip for Linux/Unix
   * assuming C_FSDRIVE and D_FSDIVE is set, see LNG.9.5 & FSC.3.3
   */
  ? CURDIR()                    && Result: /usr/users/martin
```

```
  ? GETENV("C_FSDRIVE")          && "/usr/data/hugo"
  ? GETENV("D_FSDRIVE")          && "/home"
  ? GETENV("E_FSDRIVE")          && ""
  ? CURDIR("D:")                 && Result: /home
  ? CURDIR("C:")                 && Result: /usr/data/hugo
  ? CURDIR("E:")                 && Result: /usr/users/martin

  SET DIRECTORY TO /tmp

  // or with FS2 Toolbox:
  DirChange("/tmp")              && change directory

  ? CURDIR()                     && Result: /tmp

  SET DIRECTORY TO "D:\data"
  // ? DirChange("D:\data")
  ? CURDIR()                     && Result: /home/data
  ? CURDIR(, .T.)                && Result: home/data
#endif
```

Classification:

system, file access

Compatibility:

On Unix the path and file names are case sensitive, the directory separator is a slash character. See also LNG.3.2 and LNG.9.3. The second parameter achieves compatibility to Clipper in MS-Windows.

Related:

SET DIRECTORY, SET DEFAULT, SET PATH, GETENV(), DIRECTORY(), FSC:environment variables, FS2:DIRCHANGE(), FS2:DIRMAKE()

DATE ()

Syntax:

`retD = DATE ()`

Purpose:

Returns the system date in form of a date value.

Returns:

<**retD**> is a date value, representing the current system date.

Description:

DATE() is used whenever we need to know the system date, and perform operations with it.

The date output depends on the current state of SET DATE, SET CENTURY and SET EPOCH commands. The date value stored is not affected by these settings. See also LNG.2.6.4.

Example:

```
? DATE()                                  && 08/20/15
? DATE() + 5                              && 08/25/15
? DATE() + 30                             && 09/19/15
? YEAR(DATE())                            && 2015
SET DATE GERMAN
? DATE()                                  && 20.08.15
? DATE() - 22                             && 29.07.15
```

Classification:

programming

Compatibility:

The proper date and time value depends on the correct setting of the TZ environment variable.

Related:

DATEVALID(), SET DATE, SET CENTURY, SET EPOCH, DAY(), MONTH(), YEAR(), CDOW(), CMONTH(), DOW(), DTOC(), DTOS(), TIME(), FS2:AddMonth(), FS2:DoY(), FS2:DoM(), FS2:IsLeap(), FS2:DMY(), FS2:MDY(), FS2:Week(), FS2:Quarter(), FS2:WoM(), FS2:SetDate()

DATEVALID ()

Syntax:

`retL = DateValid (expD)`

Purpose:

Test the given date if valid.

Returns:

<**retL**> is .T. if <expD> is a valid date, and is .F. when <expD> is not of date type or contains invalid date value.

Description:

DateValid() is used to test date value for validity.

Example:

```
? DateValid( 25 )                    // .F.
? DateValid( date() )                // .T.
? DateValid( ctod("12/31/02") )      // .T.
? DateValid( ctod("13/31/02") )      // .F.
? DateValid( stod("") )              // .F.
```

Classification:

programming

Compatibility:

Available in FlagShip only

Related:

DATE(), CTOD(), STOD(), EMPTY(), VALTYPE()

DAY ()

Syntax:

`retN = DAY (expD)`

Purpose:

Extracts the day of the month from a date value and returns it as a number.

Arguments:

<**expD**> is the date value.

Returns:

<**retN**> is a numeric value, representing the day of the month. This number ranges from 1 to 31. If the month is February, leap years are considered. If the date value is empty or invalid, zero is returned.

Description:

DAY() can be used in cases where the day of month is needed for calculations. Other date extracting functions are MONTH() and YEAR().

Example:

```
? DATE()                                && 08/24/93
? DAY(DATE())                           && 24
? DAY(DATE()-11)                        && 13
? DAY(CTOD(""))                         && 0
```

Classification:

programming

Related:

DATE(), DOW(), MONTH(), YEAR(), CDOW(), CMONTH(), DTOC(), DTOS() FS2:CtoDow(), FS2:DoY(), FS2:DMY(), FS2:MDY(), FS2:LastDayOm(), FS2:Week(), FS2:Quarter()

DBAPPEND ()

Syntax:

`retL = DBAPPEND ()`

Purpose:

Adds a new empty record to the end of the currently selected database.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBAPPEND() function is equivalent to the APPEND BLANK command. It adds a new record to the database associated with the current working area. After APPENDing, the new blank record becomes the current record. The new field values are initialized to empty values for each data type: character fields are filled with spaces; numeric fields with zero; logical fields are assigned FALSE; date fields are assigned CTOD("") and memo fields are left empty.

Multiuser:

In shared mode, DBAPPEND() automatically locks the new record. The lock remains active until UNLOCK or another lock (or DBAPPEND(), APPEND BLANK) is executed by the corresponding user. This automatic lock does not release a FLOCK().

If another application or user has locked the database with FLOCK(), the record is not appended and NETERR() function returns TRUE.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
USE address  SHARED NEW
WHILE NETERR(); USE address SHARED NEW; END
USE employee SHARED NEW
WHILE NETERR(); USE employee SHARED NEW; END
? address->(LASTREC())                      // 23456
? LASTREC()                                 // 100
address->(DBAPPEND())
WHILE NETERR()                              // check success
   address->(DBAPPEND())                    // or try again
ENDDO
? address->name                             // "          "
```

Classification: database

Compatibility: Clipper always returns NIL

Related:

APPEND BLANK, NETERR(), APPEND FROM, RLOCK(), FLOCK(), UNLOCK, oRdd:Append()

DBCLEARFILTER ()

Syntax:

`retL = DBCLEARFILTER ( )`

Purpose:

Clears a filter condition.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBCLEARFIL() function is equivalent to the SET FILTER TO command with no arguments. It clears the logical filter condition, if any, for the current working area. To clear a filter in any working area, use alias->(DBCLEARFILTER()).

For more information, refer to the SET FILTER command, the DBSETFILTER() function and LNG.4.6.

Example:

```
Set/reset filter in a non-current working area:

USE address  NEW
USE employee NEW
address->(DBSETFILTER( ;
          {|| TRIM(UPPER(address->name)) == "SMITH"} ))
address->(DBGOTOP())
DO WHILE ! address->(EOF())
   ? address->name, address->address
   address->(DBSKIP())
ENDDO
address->(DBCLEARFILTER())
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

SET FILTER, DBSETFILTER(), DBFILTER(), oRdd:ClearFilter(), oRdd:Filter

DBCLEARINDEX ()

Syntax:

`retL = DBCLEARINDEX ()`

Purpose:

Closes all open indices in the current working area, just as the SET INDEX TO or CLOSE INDEXES commands.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

The function is equivalent to the CLOSE INDEXES or SET INDEX TO command, without arguments. Any pending index changes are written to the file before it is closed and flushed.

To apply the function on a different working area from the current one, alias->(DBCLEARIND()) may be used.

Example:

```
USE address NEW
SET INDEX TO adr1, adr2
USE employee NEW
address->(DBCLEARIND())
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

SET INDEX, CLOSE INDEX, DBSETINDEX(), oRdd:ClearIndex()

DBCLEARRELATION ()

Syntax:

`retL = DBCLEARRELATION ()`

Purpose:

Clears all open relations in the current working area, just as the SET RELATION TO command.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

The function is a equivalent to the SET RELATION TO command, without arguments. It clears any active relations for the current working area. For more information on relations, refer to the SET RELATION command and LNG.4.7.

To apply the function in a different working area from the current one, alias->(DBCLEARREL()) may be used.

Example:

```
USE employee NEW
SET INDEX TO emplid
USE children
SELECT employee
SET RELATION TO emplid INTO employee
LIST employee->name, first, birthdate
DBCLEARRELation()
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

SET RELATION, DBSETRELATion(), oRdd:ClearRelation()

DBCLOSEALL ()

Syntax:

`retL = DBCLOSEALL ()`

Purpose:

Closes database and index files in all working areas.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

The function is equivalent to the CLOSE DATABASES command or to invoking the USE command or DBCLOSEAREA() function in every working area used.

DBCLOSEALL() releases active filters, database and index locks and relations, flushes pending changes to the database and index files, and closes the files in all working areas.

Example:

```
USE address NEW
SET INDEX TO name, city
USE employee NEW
SET INDEX TO emplid
// any database processing
DBCLOSEALL()
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

CLOSE DATABASES, CLOSE, USE, DBCLOSEAREA(), COMMIT, DBCOMMITALL(), oRdd:Close()

DBCLOSEAREA ()

Syntax:

retL = DBCLOSEAREA ([expCN1])

Purpose:

Closes database and index files in the current working area.

Options:

<**expCN1**> is optional numeric working area, or character Alias name specifying the database to be closed. If <expCN1> is not given, the database in the current working area is closed.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

The function is equivalent to the CLOSE command or to invoking USE without arguments, or to CLOSE <alias> command.

DBCLOSEARE() releases active filters, database and index locks and relations in the current working area, flushes pending changes to the database and index files, and closes the files associated with those working areas.

To apply the function in a different working area from the current one, DBCLOSEAREA("cAlias") or DBCLOSEAREA(nSelect) may be used.

Tuning:

When the database is closed, FlagShip flushes the database and index files to hard disk. You may optimize this by setting

```
_aGlobSetting[GSET_N_CLOSEOPTIMIZE] := 1  //default
```

where 0 = flush always except opened read only, 1 = only if changed, and 3 = don't flush.

Example:

```
USE employee NEW
SET INDEX TO emplid
USE children
SELECT employee
SET RELATION TO emplid INTO employee
LIST employee->name, first, birthdate
DBCLOSEAREA()                          // close children
SELECT employee
LIST name, birthdate
```

Classification:

database

Compatibility: Available in FS4, VFS and C5 only. Clipper always returns NIL

Related:

CLOSE, USE, DBCLOSEALL(), COMMIT, DBCOMMIT(), oRdd:Close()

DBCOMMIT ()

Syntax:

`retL = DBCOMMIT ( [expL1] )`

Purpose:

Writes database and index changes of the current working area to the UNIX (or Windows) buffers and flushes them to the hard disk.

Options:

<**expL1**> is an optional logical value that specifies whether an EXCLUSIVE open database should be COMMITed too. If <expL1> is not given or is .F., only SHARED open databases are committed. Passing .T. will commit both SHARED and EXCLUSIVE open databases. You may set the global switch

```
_aGlobSetting[GSET_L_DBCOMMIT_EXCL] := .T.    // default is .F.
```

which then behaves same as passing .T. for <expL1>

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBCOMMIT() is similar to COMMIT or DBCOMMITALL(), but it flushes and updates the internal buffers only of the current working area, not all of them. It either writes the system buffers immediately or asynchronously (in the background) to the hard disk, see below.

To apply the function in a different working area from the current one, alias->(DBCOMMIT()) may be used.

Multiuser:

DBCOMMIT() or DBCOMMITALL() should be executed before you free the by Flock() or Rlock() locked records or database. If SET AUTOLOCK is ON (the default), DbCommitAll() is executed automatically in AutoUnlock(), see <FlagShip_dir>/system/autolock.prg.

FlagShip stores the current .dbf record in internal working area buffers. The current work area get flushed to hard disk via DbCommit() also by:

- SKIP 0 or DbSkip(0) SHARED
- SELECT or DbSelectArea(n) SHARED
- GOTO Recno() or DbGoto(Recno()) SHARED

The DbCommitAll() function instead flushes all open work areas to hard disk also by:

- CLOSE / USE SHARED (EXCL)
- QUIT, user abort (Ctrl-K) SHARED (EXCL)
- COMMIT or DbCommit() or DbCommitAll() SHARED
- COMMIT ALL or DbCommit(.T.) or DbCommitAll(.T.) SHARED (EXCL)

When you use relations, DbCommitAll() = COMMIT ALL should be used instead of DbCommit(), since this is bound to current work area. See also SET COMMIT for further tuning.

Tuning:

see SET COMMIT

Example:

```
USE address NEW SHARED
SET INDEX TO name, city
WHILE !RLOCK(); ENDDO
REPLACE name WITH "Miller", city WITH "Munich"
DBCOMMIT()                              // flush changes
UNLOCK
```

Classification:

database, system, file access

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL and does not support the optional parameter nor tuning. Previous FlagShip versions (FS4, VFS5 and lower releases of VFS6) have used sync mode 1 by default. This default was changed to mode 4 to increase security.

Related:

COMMIT, SET COMMIT, SKIP, UNLOCK, DBCOMMITALL(), USE, oRdd:Commit()

DBCOMMITALL ()

Syntax:

`retL = DBCOMMITALL ( [expL1] )`

Purpose:

Writes database and index changes of all working areas used to the UNIX (or Windows) buffers and flushes them immediately to the hard disk.

Options:

<expL1> is an optional logical value that specifies whether an EXCLUSIVE open database should be COMMITed too. If <expL1> is not given or is .F., only SHARED open databases are committed. Passing .T. will commit both SHARED and EXCLUSIVE open databases. You may set the global switch

```
_aGlobSetting[GSET_L_DBCOMMIT_EXCL] := .T.    // default is .F.
```

which then behaves same as passing .T. for <expL1>

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBCOMMITALL() is equivalent to the COMMIT command. As opposed to the DBCOMMIT() function, it writes the work area buffers (.dbf, .dbt, .fpt, .dbv, .idx) for **all** open databases to hard disk.

The DbCommitAll() function flushes all open work areas to hard disk also by:

- CLOSE / USE SHARED/EXCL
- QUIT, user abort (Ctrl-K) SHARED/EXCL
- COMMIT or DbCommit() or DbCommitAll() SHARED
- COMMIT ALL or DbCommit(.T.) or DbCommitAll(.T.) SHARED/EXCL

The DbCommit() function instead flushes current work area to hard disk via also by:

- SKIP 0 or DbSkip(0) SHARED
- SELECT or DbSelectArea(n) SHARED
- GOTO Recno() or DbGoto(Recno()) SHARED

See SET COMMIT for tuning these defaults.

Multiuser:

DBCOMMITALL() or COMMIT makes all database updates immediately visible to and from other processes. The execution may be slower than the similar invocations DBCOMMIT() or SKIP 0 which flushes only current database. For more information and for differences, refer to the COMMIT command and section LNG.4.8.

DBCOMMIT() or DBCOMMITALL() should be executed before you free the by Flock() or Rlock() locked records or database. If SET AUTOLOCK is ON (the default), COMMIT is executed automatically in AutoUnlock().

Tuning:

see SET COMMIT

Example:

```
USE address NEW SHARED
SET INDEX TO name, city
USE invoice NEW SHARED
:
@ 1,0 GET address->name
@ 2,0 GET invoice->number
WHILE !RLOCK(); ENDDO                // lock invoice
WHILE !(address->(RLOCK())); END     // lock address
READ                                 // perform update
DBCOMMITALL()                        // flush, make changes visible
UNLOCK ALL                           // release locks
```

Classification:

database, system

Compatibility:

Available also in FS4, VFS and C5. Clipper always returns NIL and does not support the optional parameter.

Related:

COMMIT, DBCOMMIT(), SET COMMIT, SKIP, UNLOCK, USE, oRdd:Commit()

DBCREATE ()

Syntax:

```
retL = DBCREATE (expC1, expA2, [expC3],
                 [expN4|expC4])
```

Purpose:

Creates a database file from a structure array.

Arguments:

<**expC1**> is the name of the new database file to be created. If the extension is not specified, .dbf is used. If specified without path, the database is created in the SET DEFAULT or the current directory.

<**expA2**> is a two-dimensional array that contains the structure of the database to be created. Each sub-array contain the information for one .dbf field:

Element	Type	#define	Description	Usage
expA2 [n,1]	expC	DBS_NAME	Field name	1..10 chars
expA2 [n,2]	expC	DBS_TYPE	Field type	C,N,F,D,L,M
expA2 [n,3]	expN	DBS_LEN	Field length	1..65535
expA2 [n,4]	expN	DBS_DEC	Deci places	0..18

Valid entries for the field specification:

Field type		[n,2]	Field length [n,3], value	Deci [n,4]
Character		C	1..65535 = chars	0
Number		N	1..19 = digits incl.sign & deci	0..18
Float Number		F	1..19 = digits, same as N	0..18
2-byte bin int	*	2	2 = -32768 to 32767	0
4-byte bin int	*	4	4 = -2147483648 to 2147483647	0
8-byte bin float	*	8	8 = 4.94e-324 to 1.79e+308	0
Date		D	8 = julian date	0
Logical		L	1 = 0,1,T,F,Y,N	0
Memo		M	10 = 0 to 64KB block ea 512	0 (.dbt)
Variable Char	*	V	10 = 0 to 2 GB variable size	0 (.dbv)
Variable Char	*	VC	10 = 0 to 2 GB variable size	0 (.dbv)
Var Char compr	*	VCZ	10 = 0 to 2 GB variable size	0 (.dbv)
Binary/Blob	*	VB	10 = 0 to 2 GB variable size	0 (.dbv)
Bin/Blob compr	*	VBZ	10 = 0 to 2 GB variable size	0 (.dbv)

If the field name is > 10 characters, it will be truncated - with developer's warning if such is set by FS_SET("devel", .T.). If the field length or deci size does not fit for field type D,L,M it will be automatically fixed (with developer's warning). On other errors, RTE (run-time error) displays and the database creation is aborted. As opposite to Clipper, FlagShip checks also for the same, double defined field names.

(*) Field type V is equivalent to VC. Field types V, VC, VCZ, VB, VBZ, 2, 4, 8 are FlagShip specific. VCZ and VBZ are auto-compressed types VC and VB using LZH

algorithm. You may store binary data (e.g. images) also in V, VC, VCZ fields, when temporarily disabling automatic translation to ISO/ASCII via Set(_SET_VMEMOBIN,.T.), see also Tuning in CMD.REPLACE

Options:

<**expC3**> specifies the replaceable database driver RDD for processing in the current working area.

<**expN4**>|<**expC4**> are the access rights associated to the given file in a semi-octal notation or as a string (e.g. 664 = "rw-rw-r--"). The semi-octal notation includes three digits (for the owner, group, world), each in the range 0..7 are required. 0 digit specifies no permission, 4 = read only, 2 = write only, 6 = read/ write. If not specified, the default "umask" is used.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBCREATE() is similar to the CREATE FROM command which creates a new database file from a structure extended file instead of an array variable.

Before using DBCREATE(), the array <expA2> must be specified and filled with the field definitions. All four elements of the sub-array have to be filled with a valid entry; see above table. The field name given in the [n, 1] element can be given in upper or lower case. It will be translated to uppercase. Up to 10 characters are significant; refer also to LNG.4.2. The field type in the [n,2] element must contain at least the first character of the data type, e.g. both "C" or "Char" are valid entries.

If FS_SET("upperfile" or "lowerfile") is set .T., the database (and Memo file if any) is created in uppercase or lowercase, regardless the case of <expC1>. This is mainly relevant for Linux, where the file case is significant.

Unicode:

In GUI mode, FlagShip supports also Unicode (UTF-8 and UTF-16). Since each glyph is stored in UTF-8 encoding which results in one to four bytes each - usually as chr(128..255), you may need to specify the "C" field containing glyphs correspondingly (e.g. to 30 or more characters to accept 10 Japanese or Chinese glyphs). To display the Unicode field, the current or default font must be set to Unicode by SET GUICHARSET FONT_UNICODE or oApplic:Font:CharSet(FONT_UNICODE)

Memo field files:

Data for Memo fields (M) and data for Variable Char/Blob (V*) are stored in separate file, using the database name and following extension (default in lowercase):

.DBT with Memo (M) field (default by SET MEMOFILE TO DBT), compatible to dBaseIII+, Clipper and any standard xBase. Stores 0 to 2 GB chars each (but most other drivers supports only 64 KB, consider this for cross portability). It may not contain chr(0) and chr(26) - if required, use the MEMOENCODE() function. The chr(26) terminates the memo field. The .DBT structure uses subsequent blocks of 512 bytes. Blocks for the same record are re- used

when the new data occupy less or equal blocks, otherwise new block sequence is created. The database (.dbf) header will contain 0x83 = chr(131) in the first byte, or 0x93 = chr(147) when also variable data ("V*") are used.

.FPT with Memo (M) field and setting SET MEMOFILE TO FPT, compatible to Foxbase, FoxPro, and Clipper's DBFCDX RDD. Stores 0 to 64KB characters each, may contain any bytes of chr(0..255). The FPT structure uses blocks of 8 to n bytes, default block size is 64 bytes, modifiable by assigning `_aGlobSetting[GSET_N_DBMEMOFPTBLOCK]:= n // default n=64` before invoking DBCREATE(), n is in range 1 to 65535, but some other drivers accepts 32 to 1024 only. Blocks for the same record are re-used when the new data occupy less or equal blocks, otherwise new block sequence is created. The database (.dbf) header will contain 0xF5 = chr(245) in the first byte.

.DBV with Variable Character or Blob (V*), optional together with .DBT Available in FlagShip only. Stores up to 2 GigaBytes characters (or binary data) in compressed or native format, in continuous space, may contain any bytes of chr(0..255). See tuning details in CMD.REPLACE. Current data are re-used when the new data size is less or equal to previous size, otherwise new data sequence is stored. The database (.dbf) header will contain 0x13=chr(19) in the first byte, or 0x93 = chr(147) when also .DBT field is used.

Note: the kind of Memo file is stored in database header and cannot be changed after creating the database. The Memo file extension is fix, the file is created in the same directory as the database.

Example 1:

```
LOCAL aStru := {{"Name",     "C", 30, 0}, ;  // up to 30 chars
                {"Birth",    "D",  8, 0}, ;  // date
                {"Zip_Code", "N",  6, 0}, ;  // 999999
                {"City",     "C", 25, 0}, ;  // up to 25 chars
                {"Earn",     "N",  9, 2} }   // 999999.99

DBCREATE ("Address", aStru)      // creates "Address.dbf"
// DBCREATE ("/tmp/other.dbf", aStru, "DBFIDX", 666) // -rw-rw-rw-
```

Example 2:

Create database with two memo fields:

```
LOCAL aStru := {{"Name",     "C", 30, 0}, ;  // up to 30 chars
                {"Birth",    "D",  8, 0}, ;  // date
                {"Zip_Code", "N",  6, 0}, ;  // numeric 999999
                {"City",     "C", 25, 0}, ;  // up to 25 chars
                {"Earn",     "N",  9, 2}, ;  // numeric 999999.99
                {"Memo1",    "VC",10, 0}, ;  // 0..2GB chars
                {"Memo2",    "M", 10, 0} }   // 0..64KB chars

DBCREATE("address", aStru)      // "address.dbf" and "address.dbt"

SET MEMOFILE TO FPT             // use .FPT memo field structure
FS_SET("upperfile", .T.)        // convert file names to uppercase
```

```
 DBCREATE("address2.xyz", aStru) // "ADDRESS2.XYZ" and
"ADDRESS2.FPT"
```

Example 3:

Modify a database structure:

```
LOCAL aStru, oldlen

USE mydbf
aStru  := DBSTRUCT()                    // current structure
oldlen := LASTREC()
USE
RENAME mydbf.dbf TO mydbf.old           // save old data
IF DOSERROR() > 0 .OR. !FILE("mydbf.old")
   ? "Error on rename" ;  QUIT
ENDIF
IF aStru[5,1] == "XYZ"
   ADEL (aStru,5)                        // delete 5th field
   ASIZE(aStru, LEN(stru) -1)
ENDIF

AADD(aStru, {"NewField", "Num", 8, 2} ) // add a new field
DBCREATE ("mydbf", aStru)               // create new database
USE mydbf
APPEND FROM mydbf.old                   // get the old data
IF LASTREC() != oldlen                  // OK ?
   ? "Error, modification failed" ; QUIT
ENDIF
```

Classification:

database

Compatibility:

Available in FS4, VFS, C5 and VO, the fourth parameter is available in FS4 and VFS only. C5 returns NIL. The optional .DBV and .FPT Memo file is available in FlagShip only.

Include:

The constants for array elements are available in the "dbstruct.fh" file.

Related:

COPY STRUCT EXTEND, CREATE FROM, SET MEMOFILE, DBSTRUCT(), FS_SET(), oRdd:CREATEDB(), Unicode LNG.5.4.5

DBCREATEINDEX ()

Syntax:

```
retL = DBCREATEINDEX (expC1, expC2, expB3, [expL4],
          [expL5] )
```

Purpose:

Creates an index file, equivalent to the INDEX ON command.

Arguments:

<**expC1**> is the name of the index file to be created. The default file name extension is .idx, if none is specified.

<**expC2**> is an expression that returns, evaluated as a macro, the key value to place in the index. The given <expC2> must be a character string, the macro evaluated value can be character, numeric, date, or logical.

The <expC2> is stored in the index file header and is evaluated later on any database/index movement. The maximum length of the given <expC2> is 420 bytes, of the macro evaluated expression (index key) up to 238 bytes.

LOCAL, STATIC and typed variables cannot be used in <expC2> expressions. Also, the compile-time declaration using MEMVAR or FIELD are not valid within an index key expression <expC2>; but the explicit aliasing may be used instead, if required.

To create **descending** order indices, use the DESCEND() function or the DESCENDING clause in INDEX ON command. The DESCEND() function accepts any data type as an argument and returns the value in complemented form. Then, when performing a SEEK into the index, use DESCEND() as a part of the SEEK argument. By using the DESCENDING clause, SEEK takes normal arguments (without the DESCEND() function).

<**expB3**> is a code block that returns the index key of the current record. If the block returns a FIELD variable only (like {|| name} etc.), the indexing is optimized to use this field directly. Therefore, to be sure of executing the code block, e.g. to report the ready status, use a UDF, standard function or any other operation, such as {|| UPPER(name)}, {|| myudf()}, {|| name + ""} or {|| zip+0} etc.

Instead, specifying i.e. {|| myudf(), name} will enter the myudf() function only once, since the return value of the code block is the FIELD "name".

On the first invocation of the code block, prior the indexing itself, both EOF() and BOF() are set to TRUE. When indexing is finished, EOF() and BOF() are set to TRUE only if the database is empty or filtered out.

Options:

<**expL4**> is an optional logical value that specifies whether a UNIQUE index is to be created. If <expL4> is omitted or NIL, the current SET UNIQUE status is used.

<**expL5**> is an optional logical value that specifies avoiding of the check for the FLOCK() or EXCLUSIVE open, being in SHARED mode..

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

The DBCREATEIN() function is equivalent to the INDEX ON..TO command (without condition). To create a conditional index, use the INDEX ON command or ORDCONDSET() and ORDCREATE() functions.

When DBCREATEIN() is invoked, all open index files in the current working area are closed and the new index file is created. After the command is completed, the index file created remains open, becoming the controlling index, and the record pointer is positioned to the first record in the index. The created index file can be used by the executable which creates it only, until another DBCREATEIN() is issued, or SET INDEX, USE or CLOSE releases it to shared mode.

The DBCREATEIN() function stores following data in the header of the .idx file:

- the <expC2> string as given (max. 430 bytes),
- the UNIQUE (or actual SET UNIQUE) status,
- ascending index order,
- the name of the active database, and
- the current update counter of the .dbf to synchronize integrity checking (see LNG.4.5)

- Note: Additional header data, as the FOR condition or the descending order may be set by INDEX ON or ORDCREATE (), if this alternative index function is invoked instead of DBCREATEIND().

On REINDEX, this data will be considered. For more information, refer to INDEX ON and REINDEX.

When an alias->(DBCREATEIN()) is used, the aliased working area is automatically selected as the current one before the execution of DBCREATEIN(); the previously selected working area is automatically restored afterwards.

Multiuser:

During DBCREATEIN() execution, the required database must be opened exclusively using USE.. ..EXCLUSIVE or SET EXCLUSIVE ON. If the index file is not used by another user, the FLOCK() can alternatively be used in SHARED mode to ensure data integrity, or AUTOFLOCK will be used, if not disabled. See also LNG.4.5 and LNG.4.8. You may avoid the FLOCK() check or the AUTOFLOCK() invocation by the optional <expL**5**> argument.

Example:

Graphically display the percentage and the status of the indexing process, allow aborting indexing using the ESC key:

```
USE address EXCLUSIVE
SET CURSOR OFF
DBCREATEINDEX ("adrname", "UPPER(name + first)", ;
               {|| showindex (UPPER(name+first),"address")})
SET CURSOR ON
```

```
USE address SHARED
SET INDEX TO adrname

FUNCTION showindex (key, text)              // general purpose
*****************************
STATIC savescreen, percent := 0, lastrec, every
STATIC y := MIN(23, MAXROW()-1), x := 14
IF EOF()
   RETURN key
END
IF RECNO() <= 1                             // draw index box
   savescreen := SAVESCREEN(y-1,x-2, y+1,x+51)
   @ y-1, x-2  CLEAR TO y+1,x+51
   @ y-1, x-2  TO y+1,x+51
   @ y-1, x    SAY "Creating index for " + text
   @ y-1, x+40 SAY "  0% ready"
   lastrec := LASTREC()                     // optimize access
   every   := MAX(ROUND(lastrec / 50, 0), 1)
   percent := 0
ELSEIF RECNO() = lastrec                    // finished
   RESTSCREEN (y-1,x-2, y+1,x+51, savescreen)
ELSEIF INKEY() == 27                        // user termination
   @ y,x say " **** aborted by user **** "
   QUIT
ELSEIF (RECNO() % every) == 0 .and. percent < 100
   percent += 2
   @ y, x-1 + (percent/2) SAY CHR(219)
   @ y-1, x + 40 SAY STR(percent,3)
ENDIF
RETURN key
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL and does not support the 5th parameter.

The index <file> has the default extension .idx in FlagShip, .NTX in Clipper and .NDX in dBASE. The internal structures of the index files and the locking mechanism are not compatible in these different dialects. Index files .idx from FoxPro and from FS4 are not compatible to VFS5, VFS6 and VFS7. Programs ported from DOS or FS4 have to create new indices using INDEX ON. To check the index file existence by using FILE(), either INDEXEXT() or FS_SET("trans") can be used for translation of the index extension for portable applications.

Related:

SET INDEX, REINDEX, USE, SET EXCLUSIVE, FLOCK(), ORDCREATE(), oRdd:CreateIndex(), oRdd:CreateOrder(), oRdd:SetOrderCondition()

DBDELETE ()

Syntax:

`retL = DBDELETE ( )`

Purpose:

Marks a record for deletion, equivalent to the DELETE command.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBDELETE() performs the same function as the standard DELETE command without arguments. It marks the current record as deleted. Afterwards, deleted records remain in the database until they are removed with PACK or reinstated with RECALL or DBRECALL(). These are queried with DELETED() and filtered out with SET DELETED ON.

Even if SET DELETED is ON, the current deleted record remains visible until the record pointer is moved.

Multiuser:

In a multiuser / multitasking environment, DBDELETE() requires that the current record be locked with RLOCK(). Otherwise, AUTO- RLOCK() is used automatically, if SET AUTOLOCK is active. See also LNG.4.8 and the DELETE command.

Example:

```
DBUSEAREA (.T., , "invoice", "inv", .T. )
DBSETINDEX("invdate")
WHILE ! EOF() .and. invdate < DATE() -30 .and. paid
   WHILE !RLOCK(); ENDDO                // lock record
   DBDELETE()                           // DELETE
ENDDO
DBUNLOCK()                              // UNLOCK last lock
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

DELETE, RECALL, DBRECALL(), RLOCK(), PACK, ZAP, SET DELETED, DELETED(), oRdd:Delete()

DBEDIT ()

Syntax:

```
NIL = DBEDIT ([expN1], [expN2], [expN3], [expN4],
           [expA5], [expBC6], [expA7], [expA8],
           [expA9], [expA10], [expA11], [expA12],
           [expBC13], [expBC14], [expN15],
           [expL16], [expAL17], [expL18], [expL19],
           [expL20] )
```

Syntax 2:

```
NIL = DBEDIT ([expN1], [expN2], [expN3], [expN4],
           [expA5], [expAL17], [expL18], [expL19],
           [expL20] )
```

Syntax 3:

```
NIL = DBEDIT6 ([expN1], [expN2], [expN3], [expN4],
           [expA5], [expBC6], [expA7], [expA8],
           [expA9], [expA10], [expA11], [expA12],
           [expBC13], [expBC14], [expN15] )
```

Purpose:

Displays records from one or more database files in the form of a menu defined with screen coordinates. The behavior of DbEdit6() function corresponds to FlagShip 6, where the editing, appending or deleting of records needs to be handled by user's UDF.

Arguments:

<**expN1**...**expN4**> are the window coordinates (top, left, bottom and right, in the same order). The default values are 0, 0, MAXROW(), MAXCOL(). If <expL16> is .T. or SET PIXEL is ON, the coordinates are in pixel instead of row/col (GUI mode only). If not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

<**expA5**> is an array of character expressions containing **field names** or expressions to be displayed in the respective columns. If this argument is not specified, DBEDIT() displays all fields of the current database.

<**expBC6**> is the name of a **user-defined function**, or a user-defined **code block**. Such a function or code block is used to customize the behavior of DBEDIT(). After performing the default action, if any, DBEDIT() executes the UDF or codeblock, if present. If the <expBC6> is a logical value or array, it is treated as <expAL17> = shifted parameters 17..19 (Syntax 2).

<**expA7**> is an array of character strings containing the **picture** formats for <expA5>. If the array element is NIL, default picture is used. If <expC> is specified instead, all columns are formatted with the same picture format.

<**expA8**> is an array of character strings containing the **column headings**. An <expC>, instead of an array, gives all columns the same heading. To display a multi-line heading, embedded semicolons can be used as meta-characters for NEWLINE (not in GUI). If <expA8> is not specified or the array element is NIL, the field name from <expA5> or from the database is displayed.

<**expA9**> contains the character strings used as a header separator to draw **horizontal lines** between the header and the database columns. An <expC> instead of an array gives all columns the same heading line. If not specified, CHR(196,194,196) = single line left, down and right ("T") is used. To achieve backward compatibility to Clipper and FS4, you may set CHR(205,209,205) = double line with single line down instead - but this may be displayed incorrectly on systems with codepage CP850. Applicable for Terminal i/o, ignored in GUI.

<**expA10**> contains the character strings used to separate columns drawing the **vertical lines**. An <expC> instead of an array gives all columns the same separator character. If not specified, single line down embedded in space CHR(32,179,32) is used. Applicable for Terminal i/o, ignored in GUI.

<**expA11**> contains the character strings used to separate the field display area from the footings area. An <expC> instead of an array uses the same **footing separator** for all columns. If not specified, there is no footing separator. Applicable for Terminal i/o, ignored in GUI.

<**expA12**> is an array of character strings containing the column footings. An <expC> instead of array uses the same **footing text** for all columns. If not specified, there are no column footings. Applicable for Terminal i/o, ignored in GUI.

<**expBC13**> is a code block or string specifying the FOR (display) condition. The block or the expression, if given, must evaluate to a logical. If specified, DBEDIT() displays only records satisfying the condition, whereas other records are invisibly skipped. Note: on a large database, a conditional index may perform significantly faster.

<**expBC14**> is a code block or string specifying the WHILE (display scope) condition. The block or the expression, if given, must evaluate to a logical. If specified, DBEDIT() displays only a block of (sorted) records meeting the condition. If the condition returns FALSE when invoking DBEDIT(), no records are displayed. The same occurs for each further movement, if another process changes the WHILE key value of the current record in the meantime.

<**expN15**> is the NEXT scope, specifying the range of records (starting with the current one), to be displayed. To simulate the REST clause, simply give a large number (like 4 000 000 000). This parameter also restricts the range of processed records for the FOR and/or WHILE condition. On the other hand, the WHILE scope and EOF() also automatically restrict the NEXT size.

<**expL16**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

<**expAL17**> is either logical value or an array with logical values. If the value is .T., all fields are **editable**, .F. or NIL specifies that all fields are read-only. If <expAL17> is an array, it elements allows or disables editing of the corresponding DbEdit() column. Editing is considered only if the database is not open in read-only mode. If <expAL17> is given, and is 6th parameter, remaining parameters are shifted and treated as parameter 18..19 (Syntax 2). Note that Editing of specific column fields is automatically disabled for composed field columns (like "field1 + field2"), or for functions like "str(field3,5)"

<**expL18**> is a logical value. If .T., DbEdit() handles **appending** of new records in the selected database by Ctrl-PgDn + Cursor-Down key combination. Other than .T. value, or read-only open database disables the automatic append.

<**expL19**> is a logical value. If .T., DbEdit() handles **deleting** and un-deleting of records in the selected database by Delete key, with optional confirmation (see Tuning below). Other than .T. value, or read-only open database disables this feature.

<**expL20**> is a logical value. If .T. or not specified, enables edit current cell by mouse double click, in addition to Enter key. The .F. value disables it, and allows editing by Enter only. Considered only in GUI mode with enabled editing by <expAL17>.

Returns:

The function always returns NIL.

Description:

DBEDIT () is a menu function, which displays parts of database files inside a defined window, allowing the user to browse around the database (starting at the current record), to choose records for performing various tasks, with the greatest possible information in front of him. The behavior of DBEDIT() can be easily customized by means of writing user-defined functions. Fields from the current database file, and other database files (usually binded by relations), and any expressions, can be displayed in the window.

DBEDIT() supports the following keys and actions, but depends on the UDF specified:

Key		Action	w.UDF
Cursor up	ctrl-E	Up one row	yes
Cursor down	ctrl-X	Down one row	yes
Cursor <-	ctrl-S	Column left	yes
Cursor ->	ctrl-D	Column right	yes
TAB	ctrl-H	Column left	no
shift-TAB	shift-ctrl-H	Column right	no
PgUp	ctrl-R	Previous window	yes
PgDn	ctrl-C	Next window	yes
Home	ctrl-A	Leftmost curr. column	yes
End	ctrl-F	Rightmost curr.column	yes
ctrl-Home	ctrl-]	First item in window	yes
ctrl-End	ctrl-F	Last item in window	yes
ctrl-PgUp	ctrl--	First screen row	yes
ctrl-PgDn	ctrl-^	Last screen row	yes

Esc		Terminate DBEDIT()	no
Enter	ctrl-M	Terminate DBEDIT() *	no
Delete	ctrl-G	Delete/Undelete record **	no

(*) When DbEdit() is auto-editable, i.e. <expAL17> is .T. (or at least one element of the array is set .T.), the Enter key will trigger editing modus, and DbEdit() exits by ESC key only.

(**) Delete key is supported in VFS7 and considered only when <expL19> is .T.

A key redirection using SET KEY TO has preference to the above DBEDIT() navigation keys.

Performance:

It should be pointed out that it is not recommended to display large, filtered (SET FILTER) databases by means of DBEDIT(), because it may be rather slow. It may be much faster to create and use an filtered INDEX. Using the FOR parameter together with the WHILE or NEXT condition may also perform faster than SET FILTER outside of DbEdit().

UDF usage:

If the <expC6> or <expB6> argument is used to specify the name of a user-defined function or code block, DBEDIT() executes the function (or block) every time a key is pressed when the navigation key action is performed. The function name given in <expC6> has to be specified without parenthesis and arguments.

Each time that DBEDIT() calls the UDF or code block, three parameters are passed to it:

argN1: The status mode, indicating the current state of DBEDIT() depending on the last key executed,

argN2: The index number of the current column, pointing to the position of the current column definition in the <expA5> array. If not specified, <argN2> indicates the current field position in FIELDS().

argO3: The allocated Tbrowse object self. You may change the object instances at any time (but preferably at the program beginning, when <argN1> returns DE_START).

Possible values of status modes <argN1> are:

mode	dbedit.fh	Meaning
0	DE_IDLE	Idle, no keystrokes are pending
1	DE_HITTOP	Attempt to pass top of file
2	DE_HITBOTTOM	Attempt to pass bottom of file
3	DE_EMPTY	Database is empty
4	DE_EXCEPT	Key exception (Esc or Enter)
5	DE_START	Start of DbEdit()

The UDF is entered first unconditionally with mode 5 == DE_START which allows you to set or modify its behavior before processing, e.g. by using the Tbrowse object

property. You should return 1 == DE_CONT or 2 == DE_REFRESH to continue the DbEdit() process. Thereafter, the UDF is entered each time with mode 0...4 when a key was pressed by the user - which you may retrieve by LastKey() - or when the inkey() timeout is expired.

After the UDF or code block has finished its actions, which usually depend on status, field pointer, LASTKEY(), RECNO() etc., it should return a number to DBEDIT(). Here is a list of possible return values with their effects:

retVal	dbedit.fh	Meaning
0	DE_ABORT	Abort DBEDIT()
1	DE_CONT	Continue DBEDIT()
2	DE_REFRESH	Continue, repaint the window
3	DE_APPEND	Append a record

If the database was edited from inside a UDF, it should RETURN 2, so that the changes become visible on the screen. If RETURN 1 is issued, and the new record chosen is inside the same screen, only the old current record and the new current record are painted again, showing the new value. If the screen has moved, all the records are painted again, even if one is returned. Returning 2 on mode 4 may produce an endless loop.

Tuning:

An array of strings, stored in _aGlobSetting[GSET_A_TBROWSE_MSG] specifies user defined messages displayed in the status bar of GUI based application, together with color specification. See defaults in <FlagShip_dir>/system/initio.prg

The logical value in `_aGlobSetting[GSET_A_TBROWSE_DISPLMSG, 1]` specifies whether the above messages should be displayed (if .T., the default setting is .F.)

The logical value in `_aGlobSetting[GSET_A_TBROWSE_DISPLMSG, 2]` specifies whether a message should be displayed (if .T.) when the current record is marked as "deleted", the default setting is .F. Apply only if `_aGlobSetting[GSET_A_TBROWSE_DISPLMSG, 1]` is also .T.

The logical value in `_aGlobSetting[GSET_L_ACCEPT_WHEEL]` controls support mouse wheel for scrolling. The default is .T. which enables mouse wheel in GUI mode.

The logical value in `_aGlobSetting[GSET_L_DBEDIT_STAY_ON_DELETED]` specifies whether DbEdit() shoud stay on current position of changed record (only with SET DELETED ON or with active filter), i.e. should or should not search for the changed record for this special case. The default is .F., i.e. it should not stay, but will consider the next described `_aGlobSetting[GSET_N_DBEDIT_SEARCH_CHANGED]` setting. You may set it .T. for VFS6 backard compatibility.

The numeric value in `_aGlobSetting[GSET_N_DBEDIT_SEARCH_CHANGED]` specifies whether DbEdit() should try to find current position of changed record by skipping backwards and/or forwards. Value of 1 (the default) searches backwards and then forwards, value of 2 searches forwards and then backwards, 0 avoids the search.

Considered only if `_aGlobSetting[GSET_L_DBEDIT_STAY_ON_DELETED]` is .F. (the default).

The FUNCTION _DbeditUdf(p1,p2,p3) available in dbedit.prg is the default keyboard handler used without user's UDF specified in <expC6>, but it also can be called explicitly from your UDF.

If user's UDF was not specified, the default _DbeditUdf() handler forces oTbr:AutoRefresh() every 3 seconds to check for .dbf update by other users/ processes and to refresh the display on changes.

See the <FlagShip_dir>/system/dbedit.prg source for further details; you of course may modify the behavior of DbEdit() according to your needs by changing and re-compiling this source.

Example 1:

The minimal DBEDIT() program:

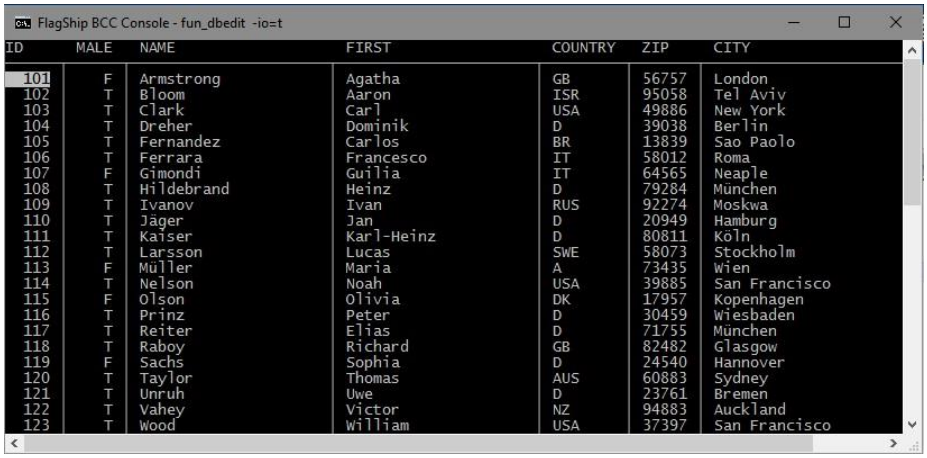
```
USE address shared
DbEdit()
```

The same minimal DBEDIT() program considering data in different character set and extended chars > 127 (here for .dbf data in ASCII/PC8/OEM charset like Clipper, with umlauts):

```
USE address shared             // open the database
if AppIoMode() == "G"
  /* GUI mode */
  SET FONT "Courier", 10       // set default font
  oApplic:Resize(25,80,,.T.)   // optional: resize window
  SET DBREAD  PC8     // optional if data in ASCII/PC8/OEM charset
  SET DBWRITE PC8     // optional if data in ASCII/PC8/OEM charset
else
  /* Terminal i/o mode */
  oApplic:Resize(25,100,,.T.)  // optional: resize window
  // SET DBREAD  ANSI          // opt. if data in ANSI/ISO charset
  // SET DBWRITE ANSI          // opt. if data in ANSI/ISO charset
endif

DbEdit(,,,,, .T.)              // browse and edit, syntax 2
```

Output:

ID	MALE	NAME	FIRST	COUNTRY	ZIP	CITY
101	F	Armstrong	Agatha	GB	56757	London
102	T	Bloom	Aaron	ISR	95058	Tel Aviv
103	T	Clark	Carl	USA	49886	New York
104	T	Dreher	Dominik	D	39038	Berlin
105	T	Fernandez	Carlos	BR	13839	Sao Paolo
106	T	Ferrara	Francesco	IT	58012	Roma
107	F	Gimondi	Guilia	IT	64565	Neaple
108	T	Hildebrand	Heinz	D	79284	München
109	T	Ivanov	Ivan	RUS	92274	Moskwa
110	T	Jäger	Jan	D	20949	Hamburg
111	T	Kaiser	Karl-Heinz	D	80811	Köln
112	T	Larsson	Lucas	SWE	58073	Stockholm
113	F	Müller	Maria	A	73435	Wien
114	T	Nelson	Noah	USA	39885	San Francisco
115	F	Olson	Olivia	DK	17957	Kopenhagen
116	T	Prinz	Peter	D	30459	Wiesbaden
117	T	Reiter	Elias	D	71755	München
118	T	Raboy	Richard	GB	82482	Glasgow
119	F	Sachs	Sophia	D	24540	Hannover
120	T	Taylor	Thomas	AUS	60883	Sydney
121	T	Unruh	Uwe	D	23761	Bremen
122	T	Vahey	Victor	NZ	94883	Auckland
123	T	Wood	William	USA	37397	San Francisco



ID	MALE	NAME	FIRST	COUNTRY	ZIP	CITY
101	F	Armstrong	Agatha	GB	56757	London
102	T	Bloom	Aaron	ISR	95058	Tel Aviv
103	T	Clark	Carl	USA	49886	New York
104	T	Dreher	Dominik	D	39038	Berlin
105	T	Fernandez	Carlos	BR	13839	Sao Paolo
106	T	Ferrara	Francesco	IT	58012	Roma
107	F	Gimondi	Guilia	IT	64565	Neaple
108	T	Hildebrand	Heinz	D	79284	München
109	T	Ivanov	Ivan	RUS	92274	Moskwa
110	T	Jäger	Jan	D	20949	Hamburg
111	T	Kaiser	Karl-Heinz	D	80811	Köln
112	T	Larsson	Lucas	SWE	58073	Stockholm
113	F	Müller	Maria	A	73435	Wien
114	T	Nelson	Noah	USA	39885	San Francisco
115	F	Olson	Olivia	DK	17957	Kopenhagen
116	T	Prinz	Peter	D	30459	Wiesbaden
117	T	Reiter	Elias	D	71755	München
118	T	Raboy	Richard	GB	82482	Glasgow
119	F	Sachs	Sophia	D	24540	Hannover
120	T	Taylor	Thomas	AUS	60883	Sydney
121	T	Unruh	Uwe	D	23761	Bremen
122	T	Vahey	Victor	NZ	94883	Auckland
123	T	Wood	William	USA	37397	San Francisco

Example 2:

This example shows the skeleton for an edit procedure using DBEDIT(), and a temporary index created "on-the-fly":

```
#include "dbedit.fh"
PRIVATE fields [2]
fields[1] = "PADR(TRIM(lastname) + ', ' + firstname, 40)"
fields[2] = "article"

_aGlobSetting[GSET_A_TBROWSE_DISPLMSG, 1] := .T.     // see Tuning
_aGlobSetting[GSET_A_TBROWSE_DISPLMSG, 2] := .T.     // see Tuning
_aGlobSetting[GSET_A_TBROWSE_MSG, 15] :=  ;
```

```
     {"", "N/W+,W+/R"}  // GUI color for field editing, see Tuning

SET DELETED OFF                    // display also deleted records
USE authors NEW
INDEX ON UPPER(lastname) + UPPER(SUBSTR(article,1,10)) ;
      FOR UPPER(SUBSTR(magazine,1,1)) $ "ABCD" TO temp
DBEDIT(10,0, 20,79, fields)                       // browse database
// DBEDIT(10,0, 20,79, fields,.T.,.T.,.T.)       // browse & edit
// DBEDIT(10,0, 20,79, fields,,,,, ,,,,, ,,.T.,.T.,.T.)  // dito
QUIT
```

Example 3:

This example is similar to the above one, but uses an available index to display only a subset of names and has own UDF handler:

```
#include "dbedit.fh"
LOCAL fields[2]
fields[1] = "PADR(TRIM(lastname) + ', ' + firstname, 40)"
fields[2] = "article"

USE authors NEW INDEX lastname
SEEK "SMITH"
if !eof()
   DBEDIT(10,0, 20,79, fields, "choose", , , , , , ,   ;
          {|| UPPER(SUBSTR(magazine,1,1)) $ "ABCD"},   ;
          "upper(left(LastName,5)) == 'SMITH'", 100 )
endif
QUIT

FUNCTION choose (mode, fld_ptr, oTbrow)
LOCAL lk := LASTKEY()
LOCAL getlist := {}, fieldvar, comma
DO CASE
CASE mode = 5                          // no action at start up
   return 1
CASE mode = 1 .or. mode = 2            // BOF, EOF
   ?? CHR(7)
CASE mode = DE_EXCEPT                  // exception
   IF lk = 27                          // Exit DBEDIT()
      RETURN 0
   ENDIF
   fieldvar := &(fields[fld_ptr])
   @ ROW(), COL() GET fieldvar
   READ CLEAR
   IF UPDATED()
      IF fld_ptr == 1
         comma := AT(",", fieldvar)
         REPLACE lastname WITH SUBSTR(fieldvar, 1, comma-1)
         REPLACE firstname WITH SUBSTR(fieldvar, comma+1)
      ELSE
         REPLACE &(fields[fld_ptr]) WITH fieldvar
      ENDIF
   ENDIF
CASE lk = -5                           // F6: print current
   SET PRINTER ON; SET CONSOLE OFF
   ? &(fields[1]), &(fields[2])
```

```
   SET PRINTER OFF; SET CONSOLE ON
CASE lk = -6                              // F7: print all
   REPORT FORM article TO PRINT NOCONSOLE
ENDCASE
RETURN 1
```

Example 4:

See additional example in <FlagShip_dir>/examples/tdbedit.prg and tbrowse_db.prg

Classification: database

Compatibility:

FlagShip's DBEDIT() is a superset of the C5 or VO's one. The FOR, WHILE conditions and the NEXT scope are available in FS only. Also, C5 and VO do not support code block <expB6>, pass only the first 2 parameters to the UDF, and do not support DE_START = mode 5. Parameters 16..19 are available in VFS7 only. The heading separator <expA9> was changed in VFS7. The AutoRefresh is available in VFS7 and newer. For backward compatibility to VFS6, you may use DbEdit6() instead.

Include: The #include constants are available in "dbedit.fh"
Source: Available in <FlagShip_dir>/system/dbedit.prg
Related: BROWSE(), TBROWSE(). OBJ.Tbrowse

DBEVAL ()

Syntax:

```
retL = DBEVAL (expB1, [expB2], [expB3], [expN4],
              [expN5], [expL6])
```

Purpose:

Evaluates a code block for each record matching a scope and condition.

Arguments:

<**expB1**> is a code block to execute for each record processed. The code block parameter list may contain one or more parameters, which remain unaffected by DBEVAL() self and which may be used for local variables in the code block body.

Options:

<**expB2**> is a **condition** specified as a code block, evaluated for each record in the scope <expN5> to <expL6>. It is equivalent to the FOR clause of the database commands. The <expB1> code block is not evaluated on records where the condition returns FALSE.

<**expB3**> is a **condition** specified as a code block, evaluated for each record from the current record until the condition returns FALSE. It is equivalent to the WHILE clause of the database commands.

<**expN4**> is the number of records to be processed starting with the current record. It is equivalent to the NEXT clause of the database commands.

<**expN5**> is the record number to be processed. It is equivalent to the RECORD clause of the database commands.

<**expL6**> is the scope that determines whether the DBEVAL() processes all the records (FALSE, the default), or the remainder starting with the current record (TRUE).

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBEVAL() is a database function that evaluates a single block for each record within the current working area that matches a specified scope and/or condition. All records within the scope or which match the condition are processed until the end of file is reached or <expB3> returns FALSE.

To execute the evaluation in a working area other than the current one, use alias->(DBEVAL(...)).

DBEVAL() is similar to AEVAL() which applies a block to each element in an array. Like AEVAL(), DBEVAL() can be used as a primitive for the construction of user-defined commands that process database files. See also the <FlagShip_dir>/include/ **std.fh** file, which translates many standard commands using DBEVAL().

Example:

Delete a scope of records, issuing a record locking:

```
LOCAL count := 0
USE address SHARED
SET INDEX TO adrname
DBEVAL ({|| count++, DBDELETE()}, ;                 // action
        {|| .not. DELETED()}, ;                     // FOR
        {|| RLOCK() .and. EMPTY(name)}, ;           // WHILE
        NIL, NIL, .T.)                              // REST
? "deleted:", count
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

EVAL(), AEVAL()

DBF ()

Syntax:

retC = DBF ([exp1], [expL2])

Purpose:

Returns the file name of the current database.

Options:

<**exp1**> is an optional **alias** name given as a character expression or the working area number given as a numeric expression. If not specified, the current working area is used.

<**expL2**> is the requested output format of <retC>. If the argument is FALSE or not specified, the returned name does not include the **path specification**. Otherwise, the given path of USE, or the default SET PATH of the opened database preceding the file name is returned.

Returns:

<**retC**> is the true file name of the database USEd in the current working area, including the extension (usually .dbf) in upper/lowercase. The path output depends on the <expL2> argument. If no database is USEd in the working area, a null string "" is returned.

Description:

The function determines the file name USEd in the current or specified working area. The file name is case sensitive in UNIX.

Example:

```
#ifdef FlagShip
   FS_SET ("lower", .T.) ; FS_SET ("pathlower", .T.)
#endif
SET PATH TO \Data\Xyz;\usr\Data
SELECT 15
USE Address ALIAS addr
USE Invoice ALIAS xyz NEW
? DBF(), ALIAS()                  && invoice.dbf  XYZ
? DBF("addr"), addr->ALIAS()      && address.dbf  ADDR
? DBF(15, .T.)                    && /usr/data/address.dbf
```

Classification:

database

Compatibility:

Compatible to DB3 and DB4. The optional parameters are available in FS4 and VFS. In C5, the function returns the alias name only, equivalent to the ALIAS() function; there is no possibility of determining the file name.

Related:

ALIAS(), USED(), INDEXNAME(), INDEXDBF(), TRUEPATH(), oRdd:Name, oRdd:Info()

DBFINFO ()

Syntax:

`retA = DbfInfo ()`

Purpose:

Return information about all open working areas. This is simplified functionality of DbfUsed()

Returns:

<**retA**> is 2-dimensional array containing information about the work area and assigned database. If no database is open, empty array is returned.

retA[n,1] = area number
retA[n,2] = alias
retA[n,3] = open mode, combination of R/S/E read-only/shared/exclus
retA[n,4] = .dbf name, same as dbf(NIL, .T.)
retA[n,5] = number of indices, same as indexcount()
retA[n,6] = number of relations
retA[n,7] = array of parents in relation

Description:

This function can be used to determine which databases are currently open and to retrieve their characteristic.

Example 1:

```
local aRet := dbfinfo()
_displarrstd(aret,"aRet")    // display the array
*output:
* aRet[1] (A) = {1, 'ADDRESS', 'S', '/home/data/address.dbf',
*                 2, 0, NIL}
* aRet[2] (A) = {5, 'ORDER',   'S', '/tmp/order.dbf',
*                 1, 0, NIL}
* aRet[3] (A) = {7, 'ARTICLE', 'S', '/server/common/article.dbf',
*                 1, 0, NIL}
```

Example 2:

```
USE multi new shared
USE test  new shared
USE multi new shared    // multiple open

? "database 'test'  is open", myDbfCheckOpen("test") + "-times"
? "database 'multi' is open", myDbfCheckOpen("multi")+ "-times"
? "database 'foo'   is open", myDbfCheckOpen("foo")  + "-times"

// -----------------------------------------------------------------
// Determine whether a database is already open in current applic.
// Only the main dbf name is tested, without regard of case
// Returns the number of used work areas for the given database
// name, i.e. 0=not open, 1=open once, 2..n=multiple open
//
// Note: the IsDbMultipleOpen() function may work more precise
```

```
// since it checks the file inode/index number instead of the
// name only and hence considers also links and different paths
// (or drives).
//
FUNCTION myDbfCheckOpen(cTestName)
  local cDbfName, ii, iPos, iRet := 0
  local aUsed := dbfinfo()

  iPos := rat(PATH_SLASH, cTestName)         // look for "/" or "\"
  if iPos > 0
     cTestName := substr(cTestName, iPos +1) // extract dbf name
  endif
  if !("." $ cTestName)
     cTestName += ".dbf"
  endif

  for ii := 1 to len(aUsed)
     cDbfName := aUsed[ii,4]
     iPos := rat(PATH_SLASH, cDbfName)      // look for "/" or "\"
     if iPos > 0                            // extract dbf name
        cDbfName := substr(cDbfName, iPos +1)
     endif
     if lower(cTestName) == lower(cDbfName) // ignore path and case
        iRet++                              // found
     endif
  next
return iRet
```

Classification:

database, programming

Compatibility:

New in VFS

Related:

DbfUsed(), ALIAS(), DBF(), USED(), INDEXNAME(), INDEXDBF(), oRdd:Name, oRdd:Info()

DBFILTER ()

Syntax:

`retC = DBFILTER ()`

Purpose:

Retrieves the active filter expression of the selected working area.

Returns:

<**retC**> is a character string representing the filter condition defined for the selected working area. If a filter is not set, a null string "" is returned.

Description:

Since each working area can have an active filter, DBFILTER() can return the filter expression of any working area, if alias->(DBFILTER()) is used.

Example:

```
To change, and then restore the old filter condition.

SELECT magazine
old_filt = DBFILTER()
SET FILTER TO Country = "Canada"
LOCATE FOR trim(name)="Smith"           && Smith in Canada
:
SET FILTER TO &old_filt
LOCATE FOR trim(name)="Smith"           && all Smiths
```

Classification:

database

Related:

SET FILTER, DBRELATION(), DBRSELECT(), oRdd:Filter

DBFLOCK()

Syntax:

```
retL = DBFLOCK ()
```

Purpose:

Locks the current database selected to allow global write access in multiuser mode. This function is fully equivalent to FLOCK()

Returns:

The return value is equivalent to FLOCK(), see description there.

Description:

DBFLOCK() is a network function that locks the current database for a global write access. To lock other database, use aliasing, e.g. alias->(DBFLOCK()) Since DBFLOCK() is an equivalence for FLOCK(), see additional description there.

Example:

See FLOCK()

Compatibility:

Refer to the FLOCK() description.

Related:

FLOCK(), RLOCK(), USE...EXCLUSIVE|SHARED, SET EXCLUSIVE, UNLOCK, ISDBRLOCK(), ISDBFLOCK(), DBRLOCK(), DBUNLOCK()

DBFUSED ()

Syntax:

`retA = DbfUsed ([expN1], [expN2], [expC3])`

Purpose:

Return and/or print information about the specified or all open working areas.

Options:

<**expN1**> is optional number of the work area (select#) to be determined. If <expN1> is

> 0:	a full info (2-dimensional array) of the specified working area <expN1> is returned and/or printed. An empty array is returned if the area is not in use
= 0:	a full info (2-dimensional array) of all working areas in use is returned and/or printed. If no areas are in use, an empty array is returned
otherwise:	a short info (1-dimensional array) about all working areas currently in use is returned.

<**expN2**> is optional, numeric value specifying the kind of printing. Considered only when <expN1> is >= 0.

= 1:	print 1-line info
= 2:	print all data
= -1,-2:	same as above, sort on WA instd of ALIAS
else:	don't print

<**expC3**> is optional character value specifying the file name for print-out. Same as SET ALTERN with the ADDIT clause If NIL, the printout (if <expN2> # 0) goes to console

Returns:

<**retA**> is 1- or 2-dimensional array containing information about the work area:

When <expN1> is not given, or not numeric, or < 0: retA[n] contains the work area number, same as Select() function

When <expN1> is >= 0:

retA	[n,1]	=	(N)	WA (work area) number
	[n,2]	=	(C)	alias name
	[n,3]	=	(C)	dbf name incl. path
	[n,4]	=	(C)	open status, a combination of "S"/"E" shared/ exclusive and "R"/"W" readonly/read-write
	[n,5]	=	(N)	control.index (SET ORDER) from [n,6] or 0
	[n,6]	=	(A)	{C,C} names and keys of assigned indices (or an empty array if no indices are open)
	[n,7]	=	(C)	filter used (or an empty string)
	[n,8]	=	(A)	{N,C,C} WA, alias and relation string (or an

empty array)
[n,9] = (A) {N [,N,N,...]} rlock-ed records or -1 on Flock-ed file

On error, i.e. when the specified working area <expN1> is not open, or when no working areas are in use, an empty array {} is returned.

Description:

This function can be used to determine and/or print an info which databases are currently open and to retrieve their characteristic. It is a superset of DbfInfo().

Example:

```
// print/display all used databases, short info
DbfUsed(0, 1)

*output:
* WA=1 ADDRESS /home/data/address.dbf SHAR idx= 2/ 1
*                                         relat= 0 filt=[]
* WA=7 ARTICLE /home/data/article.dbf SHAR idx= 1/ 1
*                                         relat= 0 filt=[]
* WA=5 ORDER   /home/tmp/order.dbf    EXCL idx= 1/ 1
*                                         relat= 0 filt=[id>=234]

// print/display all used databases, long info
DbfUsed(0, -2)

*output:
* WorkArea=1 Alias=ADDRESS  SHARED Name=/home/data/address.dbf
*   *Index=/home/data/address1.idx
*      key=name+left(first,10)
*    Index=/tmp/address2.idx
*      key=id
* WorkArea=5 Alias=ORDER    SHARED Name=/home/tmp/order.dbf
*   *Index=/home/tmp/order.idx
*      key=id
*   Filter=id>=234
* WorkArea=7 Alias=ARTICLE  EXCLUSIVE Name=/home/data/article.dbf
*   *Index=/home/data/article.idx
*      key=artnum
```

Classification:

database, programming

Compatibility:

New in VFS

Related:

DbfInfo(), ALIAS(), DBF(), USED(), INDEXNAME(), INDEXDBF(), oRdd:Name, oRdd:Info()

DBGETLOCATE ()

Syntax:

retB = DBGETLOCATeBlock ( )

Purpose:

Returns the code block of the current LOCATE condition.

Returns:

<**retB**> is a code block created from the given LOCATE..FOR condition or specified by DBSETLOCATE(). If no locate condition has been specified in the current working area, NIL is returned.

Description:

DBGETLOCATE() returns the current LOCATE ... FOR condition as a code block, if one was specified in the current working area.

Example:

```
USE test index test1 NEW
LOCATE FOR "SMITH" $ UPPER(name)
// == DBSETLOCATE ({|| "SMITH" $ UPPER(name)})

? Found(), Eval(DBGETLOCATE())          // .T., .T.
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only.

Related:

DBSETLOCATE(), LOCATE, CONTINUE, EVAL(), oRdd:GetLocate(), oRdd:Info()

DBGOBOTTOM ()

Syntax:

retL = DBGOBOTTOM ( )

Purpose:

Moves the database pointer to the last logical record.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBGOBOTTOM() is equivalent to the standard GOTO BOTTOM command. For more information, refer to the GOTO command.

The function moves to the last record of the controlling index, if there is one, or to LASTREC(), if no index is in use. If there is a filter scope, DBGOBOTTOM() moves to the last record of the scope. This may be time-consuming on large databases, because the criteria fulfilled has to be looked for, skipping backwards from the last record.

If the database is empty, or all records have been filtered out, both BOF() and EOF() return .T.

To execute the function in a different working area than the current one, use alias->(DBGOBOTTOM()).

Multiuser:

In a multiuser environment, after executing DBGOBOTTOM() the internal and system buffers are refreshed. The changes are then available to other users. Refer to LNG.4.8 and SET COMMIT to modify this behaviour.

Example:

```
USE test NEW
? LASTREC()                              // 1234
USE address NEW
ZAP
test->DBGOBOTTOM()
? test->EOF(), rest->RECNO()             // .F.  1234
DBGOBOTTOM()
? EOF(), RECNO()                         // .T.  1
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

GOTO, SET COMMIT, DBGOTO(), DBGOTOP(), BOF(), oRdd:GoBottom()

DBGOTO ()

Syntax:

`retL = DBGOTO (expN)`

Purpose:

Moves the database pointer to the specified record.

Arguments:

<**expN**> is the record to which the record pointer is to be positioned. Positioning is done even if the record falls outside the FILTER scope, or SET DELETED is ON. Records not present in an index created with SET UNIQUE ON or INDEX...UNIQUE can be accessed too. If the <expN> is out of range, the database pointer is positioned to LASTREC() + 1; both EOF() and BOF() are set to true.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBGOTO() is equivalent to the standard GOTO command. The function moves to the specified physical record of the database.

To execute the function in a different working area from the current one, use alias->(DBGOTO()).

Multiuser:

In a multiuser environment, after executing DBGOTO() the internal and system buffers are refreshed. The changes are then available to other users. Refer to LNG.4.8 and SET COMMIT to modify this behaviour.

Example:

```
USE employee
? RECNO(), RECCOUNT()                    // 1  123
DBGOTO (25)
? RECNO(), EOF()                         // 25  .F.
DBGOTO (999)
? RECNO(), EOF()                         // 124 .T.
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

GOTO, SET COMMIT, DBGOBOTTOM(), DBGOTOP(), BOF(), EOF(), oRdd:GoTo()

DBGOTOP ()

Syntax:

`retL = DBGOTOP ( )`

Purpose:

Moves the database pointer to the first logical record.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBGOTOP() is equivalent to the standard GOTO TOP command. For more information, refer to the GOTO command.

The function moves to the first record of the controlling index, if there is one, or to record 1, if there is no index in use. If there is a filter scope, DBGOTOP() moves to the first record of the scope, just as the command LOCATE does.

If the database is empty, or all records have been filtered out, both BOF() and EOF() return .T.

To execute the function in a different working area from the current one, use alias->(DBGOTOP()).

Multiuser:

In a multiuser environment, after executing DBGOTOP() the internal and system buffers are refreshed. The changes are then available to other users. Refer to LNG.4.8 and SET COMMIT to modify this behaviour.

Example:

```
USE test NEW
SET INDEX TO test
USE address NEW
test->DBGOTOP()
? test->EOF(), rest->RECNO()            // .F.  25
DBGOTOP()
? EOF(), RECNO()                        // .F.  1
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

GOTO, SET COMMIT, DBGOTO(), DBGOBOTTOM(), BOF(), oRdd:GoTop()

DBOBJECT ()

Syntax:

`retO = DBOBJECT ([expN1|expC1])`

Purpose:

Retrieves the DataServer or DBserver object of a database opened in the current or specified work area, regardless of the way opened (USE, DBUSEAREA(), DBSERVERNEW(), instantiation).

Options:

<**expN1**>|<**expC1**> specifies the working area number (1 to 65535) or the alias name of the required database. Zero (0), NIL or omitting this argument assumes the currently selected working area.

Returns:

<**retO**> is the DataServer or DBserver object, which may be assigned to a regular FlagShip variable or to an array element. Before using the object, verify that the database was successfully opened and is still open, by using the obj:USED instance. If no database is open in the requested work area, and or in case of a parameter error, NIL is returned.

Description:

In view of the heterogeneous usage of database commands, functions and objects, FlagShip's run-time keeps internally record of the assigned databases to the working areas. This allows you to manage the database by the common, conventional commands and functions, and perform special RDD requests by using the class properties (instances and methods) described below. It also avoids the unintentional database closing when leaving the valid variable scope, e.g. by returning from an UDF which had instantiated only a LOCAL DataServer or DBserver object.

In other words: regardless of the opening method, the database remains open and assigned to a working area until it is explicitly closed, or closed implicitly by opening another database in the same area, or by terminating the application.

If the DBserver object is not available yet (due to opening the database by mean of USE or DBUSEAREA), or the variable becomes invisible, you may access, assign or restore it by using the DBOBJECT() function.

As with all objects, using TYPED variables (of the known RDD or DBSERVER type) will speed up the application significantly.

Example 1:

```
LOCAL oAdr AS DBSERVER
SELECT 54321
USE address SHARED
if LASTREC() > 5
   GOTO 5
endif
oAdr := DBOBJECT()
? oAdr:INFO(DBI_ACCESSRIGHTS)
```

```
SKIP 3
if oAdr:INFO(DBI_CANPUTREC)
   APPEND BLANK
endif
```

Example 2:

```
LOCAL oServ1, oServ2 AS DBSERVER
USED()                          // .F.
doUdf1()
? USED(), RECNO()               // .T. 2
oServ1 := DBOBJECT()

doUdf2()
? USED(), RECNO()               // .T. 5
oServ2 := DBOBJECT()
? oServ1:USED, oServ1:ALIAS     // .T. ADDRESS
? oServ2:USED, oServ2:ALIAS     // .T. ADDRESS
QUIT
FUNCTION doUdf1()
  LOCAL oAdr AS DBSERVER
  oAdr := DBSERVER {"address", DB_SHARED)
  oAdr:SKIP()
  ? recno(), dbf()              // 2  address.dbf
RETURN
FUNCTION doUdf2()
  LOCAL oTemp := DBOBJECT() AS DBSERVER
  oTemp:SKIP()
  SKIP 2
  ? oTemp:RECNO, ;
    oTemp:INFO(DBI_FULLPATH)    // 5  /usr/tmp/address.dbf
RETURN
```

Classification:

programming

Compatibility:

Available in FS4, VFS only, not available in C5 or in VO.

Related:

USE, DBUSEAREA(), ISOBJCLASS(), ISOBJPROPRETY(), (OBJ) DBSERVER-NEW()

DBRECALL ()

Syntax:

retL = DBRECALL ( )

Purpose:

Reinstates the DELETEd, current record in the current working area. If the record was not deleted, no action is taken.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBRECALL() is equivalent to the standard RECALL command processing the current record.

The deleted records are invisible when SET DELETED is ON and the database pointer was moved. To reach deleted records, use GOTO, DBGOTO() or SET DELETED OFF.

To execute the function in a different working area from the current one, use alias->(DBRECALL()).

Multiuser:

RLOCK() is required when reinstating one record, while FLOCK() when DBRECALL() is used in DBEVAL() to process multiple records. Otherwise, AUTORLOCK() is used automatically, if SET AUTOLOCK is active.

Example:

```
USE employee NEW
DBDELETE()
? DELETED()                                 &&   .T.
DBRECALL()
? DELETED()                                 &&   .F.
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

RECALL, DELETED(), DELETE, DBDELETE(), PACK, ZAP, SET DELETED, oRdd:Recall()

DBREINDEX ()

Syntax:

`retL = DBREINDEX ( )`

Purpose:

Rebuilds all open indices in the current working area.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBREINDEX() is equivalent to the standard command REINDEX. It performs the same action as INDEX ON... or DBINDEX() but uses the index criteria already stored in the index header. Therefore, should the index file be corrupted, the INDEX ON command should be used.

REINDEX() conforms to the UNIQUE and ASCEND/ DESCEND status as well as to the FOR <condition> of the first creation with INDEX ON. The current SET UNIQUE program status is not considered, but the stored status in the index header is used instead (see INDEX ON).

When REINDEX is finished, all current indices remain open, the ORDER is set to 1, and the database pointer is positioned to the first logical record. To execute the function in a different working area from the current one, use alias->(DBREINDEX()).

Refer to the REINDEX command for more information.

Multiuser:

The required database must be exclusively opened. In a multiuser environment, the time-consuming REINDEX() can be fully omitted, if all relevant index files are **always** assigned to the open databases. To select the required index file, use the SET ORDER command.

Example:

```
USE employee INDEX name
REPLACE ALL salary WITH salary + 100
SET INDEX TO id, birthdate, salary, name
REINDEX()
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

REINDEX, INDEX, PACK, SET INDEX, SET ORDER, USE, INDEXCHECK(), INDEXNAMES(), INDEXDBF(), oRdd:Reindex()

DBRELATION ()

Syntax:

`retC = DBRELATION ( [expN] )`

Purpose:

Retrieves the relational expression of the specified relation in the selected working area.

Options:

<**expN**> is the ordinal number of a relation in the relations list of the selected working area. The relations are numbered according to the order in which they were defined with SET RELATION.

If the argument is not specified, 1 is assumed.

Returns:

<**retC**> is a character string representing the relational expression of the relation specified by <expN>. If the specified relation does not exist, a null string "" is returned.

Description:

Using DBRELATION () with DBRSELECT () makes it possible for the user to save and restore relations to create and save environments.

Example:

```
USE article
SELECT 2
USE magazine INDEX magazine
SELECT 3
USE author INDEX author
SELECT Article
SET RELATION TO magazine INTO magazine, ;
    TO author INTO author
? DBRELATION(1)                         && Result:  magazine
? DBRELATION(2)                         && Result:  author
? DBRELATION(3)                         && Result:  ""
```

Classification:

database

Related:

SET RELATION, DbFilter(), DbRSelect(), DbRelCount(), oRdd:Relation()

DBRELCOUNT ()

Syntax:

`retN = DBRELCOUNT ( )`

Purpose:

Retrieves number of relations to child working areas from current working area.

Returns:

<**retN**> is a numeric value, representing the number of active relations from the current WA, or zero if there are no relations set by SET RELATION or DbSetRelation().

Example:

```
USE company
SELECT 20
USE departments INDEX comp_departm
SELECT 30
USE names INDEX comp_names
SELECT company
SET RELATION TO IdDepart INTO departments, TO IdName INTO names
? DbRSelect(1), DbRSelect(2)            // Result:  20   30

? "relations from current WA", ltrim(select()), alias()
FOR ii := 1 to DbRelCount()
   ? "relation #" + ltrim(ii) + " to WA " + ltrim(DbRselect(ii)), ;
     "=", alias(DbRselect(ii))
   if DbRelMultiple(ii)
      ?? " MULTIPLE = 1:n"
   endif
   ?? " by expression:", DbRelation(ii)
NEXT
```

Classification:

database

Compatibility:

Available in VFS7 and later.

Related:

SET RELATION, DbRelation(), DbRselect(), DbRelMultiple(), oRdd:RelationObject(), oRdd:SetRelation(), oRdd:Info(DBI_RELAT_COUNT)

DBRELMULTI ()

Syntax:

```
retL = DBRELMULTI ( [expN1], [expL2] )
retL = DBRELMULTIPLE ( [expN1], [expL2] )
```

Purpose:

Retrieves or set 1:n status of already set relation within current working area.

Options:

<**expN1**> is the ordinal number of a relation in the relations list of the selected working area. If the argument is not specified, 1 (the first relation) is assumed.

<**expL2**> is the new 1:n status to be set. If <expL2> is .T., the relation becomes 1:n, or 1:1 when <expL2> is .F. Default is NIL which does not change the status.

Returns:

<**retL**> is a logical value, representing the status of active relation from the current WA (before setting new value if <expL2> was given). .T. signals multiple 1:n relation, .F. is related 1:1. On error, .F. is returned.

Description:

SET RELATION links the active database (parent) with other opened databases (children) identified by INTO <alias>, see LNG.4.7. Each parent working area can be linked to unlimited number of child working areas. FlagShip supports per default 1:1 relations, upon request also 1:n or 1:n:n... which is set by the MULTIPLE clause of SET RELATION command or corresponding argument of DbSetRelation(). With DbRelMulti(), you may retrieve or change this status. For further details, see CMD.SET RELATION and LNG.4.7

Example:

```
USE company
SELECT 20
USE departments INDEX comp_departm
SET RELATION TO IdDepart INTO departments

? "relations from current WA", ltrim(select()), alias()
FOR ii := 1 to DbRelCount()
   ? "relation #" + ltrim(ii) + " to WA " + ltrim(DbRselect(ii)), ;
     "=", alias(DbRselect(ii))
   if DbRelMultiple(ii)
      ?? " MULTIPLE = 1:n"
   endif
   ?? " by expression:", DbRelation(ii)
NEXT
```

Classification: database
Compatibility: Available in VFS7 and later
Related:

SET RELATION, DbRelation(), DbRselect(), DbRelCount(), oRdd:RelationObject(), oRdd:SetRelation(), oRdd:Info(DBI_RELAT_COUNT)

DBRLOCK()

Syntax:

```
retL = DBRLOCK ( [expN] )
retL = DBRLOCK ( [expN1], [expN2,...] )
retL = DBRLOCK ( [expA] )
```

Purpose:

Locks the current or specified record(s) in the selected working area to enable exclusive write access in multiuser mode. This function is fully equivalent to RLOCK()

Options:

The parameters are equivalent to RLOCK(), see the description there

Returns:

The return value is equivalent to RLOCK(), see description there.

Description:

DBRLOCK() is a network function that locks the current or specified record(s), preventing other users from updating the record(s) until the lock is released. Since DBRLOCK() is an equivalence for RLOCK(), see additional description there.

Example:

See RLOCK()

Compatibility:

Refer to the RLOCK() description.

Related:

ISDBRLOCK(), DBRLOCKLIST(), DBRUNLOCK(), USE...EXCLUSIVE, SET EXCLUSIVE, UNLOCK, FLOCK(), ISDBFLOCK(), SET AUTOLOCK, SET MULTILOCK, RLOCKVERIFY(), oRdd:Rlock(), RLOCK()

DBRLOCKLIST ()

Syntax:

`retA = DBRLOCKLIST ( )`

Purpose:

Determines list of records locked by the RLOCK() function.

Returns:

<**retA**> is an array with the numbers of currently locked records. An empty array is returned if no records are locked.

Description:

DBRLOCKLIST() is used to obtain the list of record locks in the current working area. The LEN(<retA>) returns the number of all locked records in the area. To perform the function in a different working area from the current one, use alias->(DBRLOCKLIST()).

To determine if the current or specified record is locked, you may also use the ISDBRLOCK() function.

Example:

```
USE address SHARED
for ii := 1 to lastrec() STEP 5
   RLOCK (ii)
next
GOTO 6 ; replace ....
aLocks := DBRLOCKLIST()
aeval (aLocks, {|x| qqout(x)})    //  1, 6, 11, 16, ....
? len(aLocks)                     // 1050
DBRUNLOCK (1)
? len(DBRLOCKLIST())              // 1049
UNLOCK
? len(DBRLOCKLIST())              // 0
```

Classification:

database

Compatibility:

Available in FS4, VFS and VO only.

Related:

RLOCK(), ISDBRLOCK(), DBRUNLOCK(), DBUNLOCK(), DBUNLOCKALL(), UNLOCK, oRdd:RlockList()

DBRSELECT ()

Syntax:

`retN = DBRSELECT ( [expN] )`

Purpose:

Retrieves the child working area of the specified relation, defined in the selected working area.

Options:

<**expN**> is the ordinal number of a relation in the relations list of the selected working area. If the argument is not specified, 1 (the first relation) is assumed.

Returns:

<**retN**> is a numeric value, representing the working area to which the specified relation has been set. If the specified relation does not exist, zero is returned.

Example:

```
USE article
SELECT 2
USE magazine INDEX magazine
SELECT 3
USE author INDEX author
SELECT article
SET RELATION TO magazine INTO magazine, ;
   TO author INTO author
? DBRSELECT(1), DBRSELECT(2)            && Result:  2   3

? "relations from current WA", ltrim(select()), alias()
FOR ii := 1 to DbRelCount()
   ? "relation #" + ltrim(ii) + " to WA " + ltrim(DbRselect(ii)), ;
     "=", alias(DbRselect(ii))
   if DbRelMultiple(ii)
      ?? " MULTIPLE = 1:n"
   endif
   ?? " by expression:", DbRelation(ii)
NEXT
```

Classification:

database

Related:

SET RELATION, DbFilter(), DbRelation(), DbRelCount(), DbRelMulti(), oRdd:RelationObject(), oRdd:SetRelation()

DBRUNLOCK ()

Syntax:

`retL = DBRUNLOCK ([expN1])`

Purpose:

Releases RLOCK() of the specified record or all record locks.

Options:

<**expN1**> is the record number, of which the record lock should be released. If this record was not previously locked by RLOCK(), the function return FALSE. If <expN**1**> is not given or specified 0, all record locks are released.

Returns:

<**retL**> signals a successful release of the record lock.

Description:

DBRUNLOCK() is an opposite function of RLOCK(). If the argument <expN1> is not given, it is similar to DBUNLOCK() or the standard UNLOCK command, but frees only previously record locks, not a file lock. If the database is used EXCLUSIVE, no action is performed.

To perform the function in a different working area from the current one, use alias->(DBRUNLOCK()).

To determine the currently Rlock-ed records, use DBRLOCKLIST() or ISDBRLOCK() functions.

Tuning:

You may automatically perform DbCommit() or DbCommitAll() at the time of DbrUnlock() by assigning

```
  _aGlobSetting[GSET_N_UNLOCK_COMMIT] := num   // default = 0
```

where num=0 disables this feature, num=1 performs DbCommit() and num=2 invokes DbCommitAll() instead. Alternatively, SET AUTOCOMMIT ON has similar functionality. See further details and settings in SET COMMIT.

Example:

```
USE address SHARED
for ii := 1 to lastrec() STEP 5
   RLOCK (ii)
next
GOTO 6 ; replace ....
aLocks := DBRLOCKLIST()     //  1, 6, 11, 16, ....
? DBRUNLOCK (recno())       // .T.
? DBRUNLOCK (5)             // .F.
? DBRUNLOCK (11), RECNO()   // .T. 6
aLocks := DBRLOCKLIST()     //  1, 16, ....
? ISDBRLOCK (), RECNO()     // .F. 6
? ISDBRLOCK (16)            // .T.
```

Classification:

database

Compatibility:

Available in FS4, VFS and VO only.

Related:

SET COMMIT, RLOCK(), DBRLOCKOCKLIST(), ISDBRLOCK(), DBUNLOCK(), DBUNLOCKALL(), UNLOCK, oRdd:Unlock()

DBSEEK ()

Syntax:

`retL = DBSEEK (exp1, [expL2])`

Purpose:

Seeks through an index file until the first key matching the given expression is found.

Arguments:

<**exp1**> is an expression to be matched with the index key of the currently active index file (controlling index). The scope is ALL (the search starts with the first logical record).

Options:

<**expL2**> is an optional logical value that specifies whether a soft seek is to be performed. If the argument is not specified, the current SET SOFTSEEK state is used.

Returns:

<**retL**> is the equivalent to FOUND(), specifying if the specified key value was found.

Description:

DBSEEK() is equivalent to the standard SEEK command.

The searching of the controlling index starts from the first key. If a match is found, the record pointer is positioned to the record number found in the index and FOUND() returns TRUE, EOF() returns FALSE.

If the searched value is not found, the <expL2> argument or the current setting of SET SOFTSEEK affects the returning value of FOUND(), EOF() and the position of the record pointer; refer to the SEEK command.

DBSEEK() does not affect a working area with no active index. The SET DELETED and SET FILTER switch/condition is considered. The current setting of SET EXACT does not affect the search; the comparison is the same as SET EXACT OFF.

Tuning:

You may force DbCommit() on DbSeek() by setting

```
_aGlobSetting[GSET_L_DBCOMMIT_SEEK] := .T. // default is .F.
```

it behaves then same as VFS6 and Clipper. See further details and settings in SET COMMIT.

Example:

```
USE address NEW
DBSETINDEX ("adrname")                  // UPPER(name)
DBSETINDEX ("adrzip")                   // zipcode
? DBSEEK ("SMITH")                      // .T.
? FOUND(), EOF(), name                  // .T. .F. Smith
SET ORDER TO 2
? DBSEEK (1234, .T.)                    // .F.
? FOUND(), EOF(), zipcode               // .F. .F. 1255
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only.

Related:

SEEK, SET COMMIT, DBGOBOTTOM(), DBGOTOP(), DBSKIP(), EOF(), FOUND()

DBSELECTAREA ()

Syntax 1:

```
retL = DBSELECTAREA ( expN1|expC1 )
```

Syntax 2:

```
retN = DBSELECTAREA ( )
```

Purpose:

Changes the current working area or determines current WA number.

Arguments:

<**expN1**> is a numeric value between zero and 65534, inclusive, that specifies the SELECT() number of the working area being selected. A zero (0) argument selects the first unused working area, similar to the USE ... NEW clause.

<**expC1**> is a character value that specifies the **alias** of the working area being selected.

If no parameter is passed, DbSelectArea() behaves same as SELECT() function, i.e. returns the current working area number (1 to 65534).

Returns:

<**retL**> with syntax 1 signals success if TRUE, or is FALSE otherwise.

<**retN**> with syntax 2 returns the current working area number.

Description:

DBSELECTAR(param) is equivalent to the standard SELECT ... command and DBSELECTAR() is equivalent to the standard SELECT() function.

In FlagShip, 65534 working areas are available for simultaneously open databases. The ALIAS of a working area is automatically assigned when a database file is opened by the USE command.

For more information refer to the SELECT command.

Multiuser:

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Tuning:

When _aGlobSetting[GSET_L_DBCOMMIT_SELECT] switch is .T. (default is .F.), every SELEXT, DbSelectArea() or alias-> to foreign database will also commit (flush) the current work area physically to hard disk, same as COMMIT or DbCommit() - but this may decrease the speed significantly when you use the SELECT or DbSelectArea() or alias-> in a large loop with SHARED databases. See further details in SET COMMIT.

Example:

```
 USE address NEW                 // same as: DbSelectArea(0);
USE ...
 SELECT 155                      // same as: DbSelectArea(155)
 USE invoice INDEX invcust NEW
 DBSELECTAREA ("address")        // same as: SELECT address
 ? name
 DBSELECTAREA (155)              // same as: SELECT 155
 SEEK address->custno
 ? DBSELECTAREA()                // same as: SELECT()
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL and supports 250 to 255 working areas; other Xbase systems, like dBASE or FoxPro even less.

Related:

SELECT, SELECT(), USE, USED(), DBUSEAREA(), SET COMMIT, oRdd:Used

DBSETDRIVER ()

Syntax:

`retC = DBSETDRIVER ( [expC1] )`

Purpose:

Returns the default database driver and optionally sets a new driver.

Options:

<**expC1**> is an optional character value that specifies the name of the database driver RDD that should be used to activate and manage new working areas when no driver is explicitly specified.

Returns:

<**retC**> is the name of the current database driver. The default "DBFIDX" is returned, if not otherwise set.

Description:

DBSETDRIVER() is equivalent to RDDSETDEFAULT(), see also its description. The function sets the database driver to be used when activating new working areas without specifying a driver. If the specified driver is not available to the application, the call has no effect.

Example:

```
? DBSETDRIVER ()                          // dbfidx
DBFSETDRIVER ("dbfntx")
IF LOWER(DBSETDRIVER ()) != "dbfntx"
   ? "DBFnxt driver is not available"
ENDIF
USE address
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only.

Related:

USE, DBUSEAREA(), RDDSETDEFAULT(), oRdd:Driver

DBSETFILTER ()

Syntax:

`retL = DBSETFILTER (expB1, [expC2])`

Purpose:

Makes a database appear as if it contains only the records meeting the specified condition.

Arguments:

<**expB1**> is a code block that expresses the filter condition in executable form. The code block body is equivalent to the <condition> argument of the SET FILTER TO command.

Options:

<**expC2**> is an optional character value that expresses the filter condition in textual form. If not specified, the DBFILTER() function will return an empty string for the working area.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBSETFILTER() is equivalent to the standard SET FILTER command.

Each working area can have an active filter. When set, a filter is activated on the first movement of the record pointer in the corresponding working area, e.g. using the GOTO TOP command. The current filter condition can be returned as a character string using the DBFILTER() function.

For more information, refer to the SET FILTER command.

Example:

```
USE address NEW
DBSETFILTER ( {|| "SMITH" $ UPPER(name)}, ;
                  "'SMITH' $ UPPER(name)")
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

SET FILTER, SET DELETED, DBFILTER(), LOCATE, SEEK, oRdd:Filter, oRdd:FilterString

DBSETINDEX ()

Syntax:

`retL = DBSETINDEX (expC1 [, expC2 [,expCn]] [,expL2])`

Purpose:

Opens the specified index file and adds it to the list of open indices in the current working area.

Arguments:

<**expC1**> is the file name to be opened and added to the current working area. A file name resulting in either a null string "", NIL or spaces is ignored. If an extension is not specified, .idx is assumed.

Options:

<**expC2..expCn**> are additional index file names to be opened. If one the index files could not be open correctly, the return value is .F. and NetErr() returns .T.

<**expL2**> is a logical expression that specifies whether the index should be opened EXCLUSIVE to that application. If not specified, the default is FALSE, which opens the index in SHARED mode.

Returns:

<**retL**> is equivalent to .NOT. NETERR(). A TRUE value indicates successful index opening.

Description:

DBSETINDEX() performs the same function as the standard SET INDEX command or the INDEX clause of the USE command, except it does not clear already assigned indices, as SET INDEX do. To clear already assigned indices in the active work area, use DbClearIndex().

If the newly opened index is the first index associated with the working area, it becomes the controlling index; otherwise, the controlling order remains unchanged.

After the new index file is opened, the working area is positioned to the first logical record in the controlling order. For more information, refer to the SET INDEX command.

When the open fails, DbSetIndex() will return .F. When SET OPENERROR is ON (the default), an open failure will raise run-time error. For a full backward compatibility to FS 4.4, or to avoid RTE, use SET OPENERROR OFF and check the return status thereafter. Multiple assignment of the same index file into the same work area is not allowed and will be ignored, this will also raise developer warning when FS_SET("devel",.T.) is set.

Tuning:

FlagShip do not raise run-time error on failure, you should check the return value or NETERR() for success. You however may force run- time-error RTE 501 on failure by assigning

```
_aGlobSetting[GSET_L_DBSETINDEX_ERR] := .T. // default = .F.
```
which then behaves FoxPro conform.

Example:

```
USE address NEW
DBSETINDEX ("adrname")
DBSETINDEX ("adrzip")
? INDEXORD(), INDEXKEY()                // 1   UPPER(name)
SET ORDER TO 2
? INDEXORD(), INDEXKEY()                // 2   zipcode
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. The optional arguments are available in FlagShip only. C5 returns NIL value.

Related:

SET INDEX, CLOSE, INDEX, REINDEX, SET ORDER, USE, INDEXEXT(), INDEXORD(), NETERR(), DbClearIndex(), FS_SET(), oRdd:SetIndex()

DBSETLOCATE ()

Syntax:

`retL = DBSETLOCATeBlock (expB1)`

Purpose:

Sets the LOCATE...FOR condition.

Arguments:

<**expB1**> is a code block that specifies the FOR condition of the LOCATE command. The code block should return a logical expression. This code block is valid until it is overwritten by the next DBSETLOCATE(), LOCATE .. FOR, or the database is closed.

Returns:

<**retL**> signals the success, if TRUE.

Description:

DBSETLOCATE() allows you to change the LOCATE .. FOR condition of the current working area, e.g. to modify the CONTINUE searching. To execute the function in a different working area from the current one, use alias->(DBSETLOCATE(...)).

As opposed to the LOCATE or CONTINUE commands, DBSETLOCATE() does not perform any search action.

Example:

```
USE test NEW
LOCATE FOR "SMITH" $ UPPER(name)
if Found()
   DBSETLOCATE ({|| "SMITH" $ UPPER(name) .and. ;
                    "John"  $ first})
   WHILE found()
      ? name, first, eval(DbGetLocate())
      CONTINUE
   ENDDO
endif
```

Classification:

database

Compatibility:

Available in FS4, VFS and VO only.

Related:

DBGETLOCATE(), LOCATE, CONTINUE, EVAL(), oRdd:Info()

DBSETORDER ()

Syntax:

`retC = DBSETORDER (expN1 | expC1)`

Purpose:

Sets the specified index number as the (master) controlling index.

Arguments:

<**expN1**> is a numeric value that specifies which index number, according to the position in the list of open indices (1..n), will become the controlling index. Permissible values of <expN1> range from 0 to 15.

<**expC1**> specifies which index name will become the controlling index; the index must already be open by USE..INDEX or SET INDEX... The <expC1> is usually only the main index name, but may optionally contain path and/or the .idx file extension. Null-string "" behaves same as DbSetOrder(0), see below.

Returns:

<**retC**> is the name of the selected index/order, or null-string "" on error.

Description:

DBSETORDER() is equivalent to the standard SET ORDER command.

By using DBSETORDER() or SET ORDER, any of the previously assigned index files (using SET INDEX or DBSETINDEX()) can be declared as the controlling index. Changing the index order does not change the database record position.

DBSETORDER(0) deselects the controlling index, switches to the natural order of records in the database, but leaves all the indices open. This is useful for replacing an area of indexed records without having the index interfering with the position in the database.

For more information, refer to the SET ORDER command.

Example:

```
USE employees
SET INDEX TO persid, name
DBSETORDER (2)         // same as DBSETORDER("name")
SEEK "SMITH"
DBSETORDER("persid")   // same as DBSETORDER(1)
SKIP                   // next person ID after Smith
```

Classification: **d**atabase

Compatibility:

Available in FS4, VFS and C5 only, character values since VFS7. Clipper always returns NIL and accepts only numeric value.

Related:

SET ORDER, SET INDEX, DBSETINDEX(), DBCLEARIND(), INDEXORD(), INDEXEXT(), INDEXKEY(), INDEXCHECK(), oRdd:SetOrder()

DBSETRELATION ()

Syntax:

```
retL = DBSETRELATION (expN1|expC1, [expB2],
          [expC3], [expL4])
```

Purpose:

Adds a new relation from current (parent) work area to child work area (<expN1> or <expC1>) by using key expression <expB2>/<expC3> or record number.

Arguments:

<**expN1**> is a numeric value that specifies the working area number of the child working area. This is equivalent to the INTO clause of the SET RELATION command.

<**expC1**> is a character value that specifies the alias of the child working area. This is equivalent to the INTO clause of the SET RELATION command.

<**expB2**> is a code block that expresses the relational expression in executable form. The body of the code block is equivalent to the TO clause of the SET RELATION command.

Options:

<**expC3**> is an optional character value that expresses the relational expression in textual form. The string is equivalent to the TO clause of the SET RELATION command. If not specified, the DBRELATION() function returns an empty string for the relation.

<**expL4**> is optional logical value. If set .T., the child database is processed as 1:n relation, otherwise it is 1:1 relation. You may set or clear the 1:n relation also later by DbRelMultiple().

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DbSetRelat() is equivalent to the standard SET RELATION command. It links the active database (parent) with other opened databases (children) identified by <exp1>, see LNG.4.7. Each parent working area can be linked to as many as eight child working areas.

When the relational expression is evaluated, the parent working area is automatically selected as the current working area before the evaluation; the previously selected working area is automatically restored afterward.

For more information, refer to the SET RELATION command.

Example:

```
USE employee NEW ALIAS empl                 // child relat.
DBSETINDEX ("empl_id")
USE children NEW                            // parent relat.
DBSETRELATION ("empl", {|| id_numb}, "id_numb")
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL. 1:n relations are available in VFS7 and later.

Related:

SET RELATION, DbRelation(), DbClearRel(), DbRselect(), DbRelCount(), DbRelMultiple(), oRdd:SetRelation()

DBSKIP ()

Syntax:

`retN = DBSKIP ( [expN1] )`

Purpose:

Moves the record pointer in the specified working area relative to the current pointer position.

Option:

<**expN1**> specifies the number of records to move the record pointer from the current position. A positive value moves the pointer forward, while a negative value moves the pointer backwards. If <expN1> is not specified, 1 is assumed.

Returns:

<**retN**> is the number of records skipped.

Description:

DBSKIP() is equivalent to the standard SKIP command. It moves the record pointer to a new position relative to the record position in the current or specified working area. If an index file is in use, DBSKIP() moves the specified number of positions according to the index keys.

To skip in a working area other than the current one, use alias->(DBSKIP(n)). Refer to the SKIP command for more information.

Multiuser:

Any record movement command, including DBSKIP(), will make changes in the current working area visible to other applications, if the current file is shared and changes were made. DBSKIP(0) flushes the current working area and system buffers to disk, as does DBCOMMIT(). See more in chapter LNG.4.8.

Example:

```
USE employee NEW ALIAS empl
? RECNO(), name                          &&   1  Miller
DBSKIP ()
? RECNO(), name                          &&   2  Johnson
USE newdbf NEW
empl->(SKIP (1 + MAX(3,2)))
? empl->(RECNO()), empl->name            &&   5  Smith
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

SKIP, BOF(), EOF(), RECNO(), GOTO, oRdd:Skip()

DBSTRUCT ()

Syntax:

`retA = DBSTRUCT ( )`

Purpose:

Creates an array containing the structure of the current database file.

Returns:

<**retA**> is a two-dimensional array containing the structure of the current database file, whose length is equal to the number of fields in the database. Each element of the array is a sub-array containing information for one .dbf field:

Element	Type	#define	Description	Usage
retA [n,1]	expC	DBS_NAME	Field name	1..10 chars
retA [n,2]	expC	DBS_TYPE	Field type	C,N,D,L,M,V*
retA [n,3]	expN	DBS_LEN	Field length	1..65535
retA [n,4]	expN	DBS_DEC	Deci places	0..18

If there is no database file in USE in the current working area, DBSTRUCT() returns an empty array {}.

Description:

DBSTRUCT() is similar to COPY STRUCTURE EXTENDED in creating an array containing the structure information rather than a database. To perform the function in a working area other than the current one, use retA = alias->(DBSTRUCT()).

Using the function DBCREATE(), a new database can be created using the structure array information.

For Numeric fields, retA[n,DBS_LEN] reports the total size of digits including sign, deci point (if any) and the deci fraction, and retA[n,DBS_DEC] reports the number of decimal digits in the fraction.

On databases created by CREATE FROM, the character field length may be given as FIELD_DEC *256 + FIELD_LEN. This is considered by DbStruct() automatically, where retA[n,DBS_LEN] reports the total length and retA[n,DBS_DEC] reports 0. If you wish to determine the field characteristics as stored in the database header, you may use FieldLen(n,.T.) and FieldDeci(n,.T.) instead.

For fields of type Date, Logical, Memo, Variable character and Binary, retA[n,DBS_DEC] always reports 0 and retA[n,DBS_LEN] the number of bytes used in the .dbf file, i.e. 1 for Logical, 8 for Date, 10 for Memo and Variable data fields, and 2,4,8 for these binary fields.

Example:

See also the example in DBCREATE() for modifying the database structure using the DBSTRUCT() function.

```
#include "dbstruct.fh"
LOCAL aStruct
```

```
USE address NEW
aStruct := DBSTRUCT()
_displarrstd(aStruct, "address")             // display on screen
AEVAL (aStruct, {|x| QOUT(x[DBS_NAME])} )    // or list field names
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5. C5 supports up to 17 deci places. The binary fields 2,4,8 and variable data/blob are available in VFS only.

Include:

The #include constants are available in "dbstruct.fh"

Related:

COPY STRUCTURE EXTENDED, DBCREATE(), oRdd:DbStruct(), FieldName(), FieldType(), FieldLen(), FieldDeci()

DBUNLOCK ()

Syntax:

retL = DBUNLOCK ( )

Purpose:

Releases all locks for the current working area.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBUNLOCK() is equivalent to the standard UNLOCK command. It frees all record and file locking of the current databases used in the multiuser/multitasking mode. If the database is used EXCLUSIVE, no action is performed.

To perform the function in a different working area from the current one, use alias->(DBUNLOCK()). For more information, refer to the UNLOCK command.

Multiuser:

In multiuser mode (or when using a concurrent database access in the same application), SET EXCLUSIVE OFF or USE...SHARED is required to open the database in the current working area. Before any write database access, the record or the database must be locked using RLOCK() or FLOCK(). Otherwise, AUTOxLOCK() is used, when SET AUTOLOCK is enabled (the default). The only exception is APPEND BLANK or DBAPPEND(), which locks the new record automatically.

The DBUNLOCK() function in FlagShip implies updating the current working area buffers to UNIX (or Windows), which flushes buffers, and makes changes visible to other applications. If SET MULTILOCKS is ON, you may unlock specific record by using DBRUNLOCK() function.

Tuning:

You may automatically perform DbCommit() or DbCommitAll() at the time of DbUnlock() by assigning

```
_aGlobSetting[GSET_N_UNLOCK_COMMIT] := num   // default = 0
```

where num=0 disables this feature, num=1 performs DbCommit() and num=2 invokes DbCommitAll() instead. Alternatively, SET AUTOCOMMIT ON has similar functionality. See SET COMMIT for additional details and options.

Example:

```
WHILE !DBUSEAREA (.T., , "invoice", , .T.)
   ? "waiting to open invoice.dbf"
   inkey(1)
ENDDO
if lastkey() == K_ESC
  Quit
endif
WHILE !FLOCK()
  ? "waiting for file lock"
ENDDO
DELETE ALL FOR paid
DBCOMMIT()
DBUNLOCK()
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

UNLOCK, DBRUNLOCK(), DBUNLOCKALL(), SET COMMIT, USE, COMMIT, DBCOMMIT(), DBCOMMITALL(), DBCLOSEAREA(), oRdd:Unlock(), SET AUTOCOMMIT, SET COMMIT

DBUNLOCKALL ()

Syntax:

`retL = DBUNLOCKALL ( )`

Purpose:

Releases all locks for all working areas.

Returns:

<**retL**> signals success if TRUE, or is FALSE otherwise.

Description:

DBUNLOCKAL() is equivalent to the standard UNLOCK ALL command. It frees all record and file locks in all working areas used. If any of the databases is used EXCLUSIVE, no action is performed in that working area. For more information, refer to the UNLOCK command.

Multiuser:

In multiuser mode (or when using a concurrent database access in the same application), SET EXCLUSIVE OFF or USE...SHARED is required to open the database in the current working area. Before any write database access, the record or the database must be locked using RLOCK() or FLOCK(). Otherwise, AUTOxLOCK() is used, when SET AUTOLOCK is enabled (the default). The only exception is APPEND BLANK or DBAPPEND() both of which lock the new record automatically.

The DBUNLOCKALL() function in FlagShip implies updating all working area buffers to UNIX (or Windows), which flushes buffers, and makes changes visible to other applications.

Tuning:

You may automatically perform DbCommit() or DbCommitAll() at the time of DbUnlockAll() by assigning

```
  _aGlobSetting[GSET_N_UNLOCK_COMMIT] := num   // default = 0
```

where num=0 disables this feature, num=1 performs DbCommit() and num=2 invokes DbCommitAll() instead. Alternatively, SET AUTOCOMMIT ON has similar functionality. See SET COMMIT for additional details and options.

Example:

```
WHILE !DBUSEAREA (.T., , "custom", , .T.)  ; ENDDO
WHILE !DBSETINDEX("custno")                ; ENDDO
WHILE !DBUSEAREA (.T., , "invoice", , .T.) ; ENDDO
SET RELATION TO cust_no INTO custom
SELECT custom
DBSEEK (1234)
WHILE !RLOCK() ; ENDDO                      // lock custom
WHILE !(invoice->(RLOCK()) ; ENDDO          // lock invoice
REPLACE sum WITH sum + paidamount, invoice->paid WITH .T.
DBCOMMITALL()
DBUNLOCKALL()                               // unlock all
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. Clipper always returns NIL.

Related:

UNLOCK, DBUNLOCKALL(), SET COMMIT, USE, DBCLOSEAREA(), oRdd:Unlock()

DBUSEAREA ()

Syntax:

```
retL = DBUSEAREA ([expL1], [expC2], expC3, [expC4],
                  [expL5], [expL6])
```

Purpose:

Opens the specified database file, and its associated memo file when memo fields exist.

Arguments:

<**expC3**> specifies the name of the database file to open in the current or a next free working area. If no extension is specified, the default .dbf extension is assumed.

Options:

<**expL1**> is a synonym for the NEW clause of the USE command. If <expL1> is specified as a TRUE value, an unused working area is selected first, making it to the current one, and the database <expC3> is opened there. If the argument is FALSE or not given, the database is opened in the currently SELECTed working area, closing any active database occupying that working area.

<**expC2**> is a synonym for the VIA clause of the USE command. It defines the replaceable database driver (RDD) to process the current working area. If not specified, the default "DBFIDX" driver, or the one specified by RDDSETDEFAULT(), is used.

<**expC4**> is a synonym for the ALIAS clause of the USE command. It specifies the alias name to be associated with the working area. If not specified or using a NIL value, the main part of the file name <expC3> is assigned to the alias. When a null string is specified, the alias-> operator will be not applicable. If the given alias is invalid, DbUseArea() will try to generate a valid name and will display corresponding warning.

<**expL5**> is a synonym for the SHARED clause of the USE command. If specified TRUE, the database is open for shared use in a multiuser, multitasking network or concurrent mode. If the argument is FALSE, the database is opened in EXCLUSIVE mode. If not specified, the current SET EXCLUSIVE status is used.

<**expL6**> is a synonym for the READONLY clause of the USE command. If the argument is TRUE, the database is opened for read-only purposes. The UNIX access rights -r- (or read-only in Windows) are sufficient for the database and memo <file> (but not for index files, which must always be -rw- or read+write). In an attempt to REPLACE or APPEND a record, a run-time error is made to occur. If the argument is FALSE or not specified, the database is open in read and write mode.

Returns:

<**retL**> is equivalent to .NOT. NETERR(). A TRUE value indicates a successful database opening.

Description:

DBUSEAREA() is equivalent to the standard USE command. It opens an existing database .dbf file, and its associated memo .dbt file in the current or the next available working area.

After opening the database, the record pointer refers to the first physical record in the file which defaults to record 1 if no index file is specified. With active index, the record pointer is set to the first logical record. When SET DELETED is ON or SET FILTER is active, you will need to invoke GO TOP to set the pointer to the first visible record. Note the GO TOP is not done automatically for performance purposes and to allow you to check the indices via IndexCheck() before moving the record pointer.

If the database is empty, both BOF() and EOF() are set TRUE. When the open fails, DbUseArea() and Used() will return .F. When SET OPENERROR is ON (the default), an open failure will raise run-time error. If you wish to avoid RTE, use SET OPENERROR OFF and check the return status.

Additional warnings are available by using FS_SET("devel", .T.)

For more information, refer to the USE command.

Multiuser:

If a multiuser, multitasking and/or network access is required, database files can be opened EXCLUSIVE or SHARED, using the <expL5> argument or the SET EXCLUSIVE command.

Opening a database EXCLUSIVEly will succeed only if it is not already in use by some other user. Attempting to open a database SHARED will succeed only if the database is not opened exclusively by other user (or concurrently in another working area). Always check whether the database has been successfully opened using the return value <retL> or the NETERR() function.

For special purposes, FlagShip allows the same database to be used simultaneously in different working areas, when the given ALIAS names given by <expC4> are different. The handling of concurrent databases is the same, as the usage of shared databases in multi- user mode. You may check the concurrent use by IsDbMuplipleOpen()

When the global switch SET NFS is ON, additional actions are taken to ensure correct buffer flushing on the NFS mounted server to avoid corrupted indices in some over-optimized NFS versions. See further details in SET NFS.

Refer to the USE command and LNG.4.8 for more information about multiuser programming.

Tuning:

FlagShip do not raise run-time error on failure, checking the return value or USED() or NETERR() reports failure or success. You however may force RTE 501 on failure by assigning

```
_aGlobSetting[GSET_L_DBUSEAREA_ERR] := .T. // default = .F.
```

which then behaves FoxPro conform.

Example:

```
DBUSEAREA (,,"tempor")                      // minimal usage

DBUSEAREA (.T., "dbfidx", "address", "adr", .T., .T.)
** is equivalent to:
** USE address ALIAS adr SHARED READONLY NEW VIA "dbfidx"

WHILE ! DBUSEAREA (.T., , "invoice", NIL, .T.)
   ? "trying to open invoice.dbf in multiuser mode"
ENDDO
IF ! DBSETINDEX ("invoice")
   ? "cannot open invoice.idx in multiuser mode"
   QUIT
ENDIF
```

Classification:

database

Compatibility:

Available in FS4, VFS and C5 only. In C5, the function returns NIL.

Related:

USE, DBCLOSEAREA(), DBSETDRIVER(), SELECT(), DBFIDXNEW(), DBSETDRIVER(), RDDSETDEFAULT(), oRdd := DBFIDX {...}, oRdd := DBSERVER {...}

DEFAULT ()

Syntax:

`ret = Default (@var, exp2)`

Purpose:

Check and set default value for variable <var>

Arguments:

<**@var**> is the variable (passed by reference) to be tested.

Returns:

<**ret**> is either equal to <var> or to <exp2>.

Description:

This function is used to set default value to specified variable, if required. If the <var> is un-initialized, i.e. the <var> is NIL or the valtype(var) is not equal to valtype(exp2), the value of <exp2> is returned. Otherwise the value of <var> is returned.

Example:

```
check(1,"hello")
check(3)
check()
wait

function check(par1n, par2c)
  par1n := default(@par1n, 9)
  par2c := default(@par2c, "nothing")

  ? procstack(), ": par1n= " + ltrim(par1n),"- par2c= " + par2c
return
```

Output:

```
CHECK/9<-FUN_DEFAULT/1 : par1n= 1 - par2c= hello
CHECK/9<-FUN_DEFAULT/2 : par1n= 3 - par2c= nothing
CHECK/9<-FUN_DEFAULT/3 : par1n= 9 - par2c= nothing
```

Classification:

programming

Compatibility:

New in VFS

Related:

VALTYPE()

DELETED ()

Syntax:

retL = DELETED ()

Purpose:

Finds out whether the current record is marked for deletion.

Returns:

<**retL**> is a logical value. The return value is .T. if the record is marked for deletion, FALSE otherwise.

Description:

The "deleted" byte is a part of any database record. The DELETED() status can be examined in a different working area from the current one, using alias->(DELETED()).

Example:

```
USE article
? DELETED()                              &&   .F.
DELETE
? DELETED()                              &&   .T.
RECALL
? DELETED()                              &&   .F.
```

Example:

Create an index including non-deleted records only:

```
USE address
INDEX ON name FOR .not. DELETED() TO adrpure
```

Classification:

database

Related:

DELETE, RECALL, PACK, SET DELETED, oRdd:Deleted

DESCEND ()

Syntax:

`ret = DESCEND (exp)`

Purpose:

For creating and searching indices in reverse order.

Arguments:

<**exp**> is an expression of any data type.

Returns:

<**ret**> is the same type as <exp>, but in a complemented form.

Description:

DESCEND () can be used for creating descending order indices and to search them with SEEK. An alternative is to create the index using the DESCENDING clause; see the INDEX ON command.

When the DESCEND() function is used in INDEX ON, you need to SEEK by DESCEND() as well.

Note that the DESCEND(string) result may be affected by the LANG environment variable, and availability of the FSsortab.def (sorting) file or use of FS_SET("loadlang",num,file) + FS_SET("setlang",num). The DESCEND(numeric or logical or date) is not affected hereby.

Note, that the DESCEND clause may be used during the index creation. This however does not store key converted by DESCEND() in the index structure, but reverts the index sorting order. To store inverted field value into index key, use the DESCEND() function during INDEX ON..TO.. and for SEEK or SEEK EVAL, see example below.

Tuning:

When the input string contains binary 0 character, it is considered but specially handled. You may modify the return equivalence of the DESCEND(chr(0)) character by assigning num value of 0 to 255 to

```
_aGlobSetting[GSET_N_DESCEND_ZERO] := 255   // default setting
```

where 0 is compatible to Clipper and FS4, 255 is compatible to VFS6 and the byte order corresponds to it value - but is same as chr(1), except you set

```
_aGlobSetting[GSET_N_DESCEND_ADD] := 0      // default is 1
```

which then behaves differently to standard, but allows real descend sort for chr(0)..chr(255) in reverse order.

Example:

```
USE magazine
INDEX ON price TO price_up
SEEK 2.50                                   // found if available
SKIP ; ? price                              // >= 2.50
INDEX ON price TO price_down DESCENDING
SEEK 2.50                                   // found if available
```

```
SKIP ; ? price                              // <= 2.50
INDEX ON DESCEND(price) TO price_desc
SEEK DESCEND(2.50)                          // found if available
SKIP ; ? price                              // <= 2.50
```

Classification:

programming (used often for database)

Compatibility:

As opposite to FlagShip, Clipper returns numeric value for a "date" argument; in FlagShip the result remain "date" type. Embedded zero bytes are supported, see Tuning above.

Related:

INDEX, SEEK, DBCREATEINDEX(), oRdd:CreateIndex(), oRdd:Cre- ateOrder()

DEVOUT ()

Syntax:

NIL = DEVOUT (exp1, [expC2], [expC3])

Purpose:

Outputs a value to the current SET DEVICE.

Arguments:

<**exp1**> is an expression of any data type to display.

Options:

<**expC2**> is an optional argument that defines the display color of <exp1>. If the current DEVICE setting is not SCREEN, the argument is ignored. If not specified, the <exp1> is displayed using the current color setting. Only the first color pair (standard) of <expC2> is significant. See SET COLOR command or SETCOLOR() function for the format of <expC2> string. This argument is always considered in Terminal i/o mode. GUI mode check first for availability of <expC3> and if not set, uses <expC2> only if SET GUICOLOR is ON.

<**expC3**> is an optional argument that defines the display color of <exp1> used in GUI mode.If the current DEVICE setting is not SCREEN, or the AppIoMode() is not "G", the argument is ignored and <expC2> used instead.

Returns:

The function always returns NIL.

Description:

DEVOUT() is similar to the @ ROW(),COL() SAY.. command performing a full-screen display at the current cursor (or printhead) position. To set the cursor to other coordinates, use DEVPOS() function.

Example:

```
SET COLOR TO "W+/B"
CLS
@ 10,0 SAY "First part" COLOR "GR+/B"
DEVOUT(" second part ")
DEVOUT(" third part", "R+/B")
```

Classification:

screen oriented or coordinates oriented printer/file output.

Compatibility:

Available in FS4, VFS and C5 only. Embedded zero bytes in <expC1> are not supported. The <expC3> argument is available in VFS only.

Related:

@..SAY, DEVOUTPICT(), DEVPOS(), COL(), ROW(), SET COLOR, SETCOLOR()

DEVOUTPICT ()

Syntax:

NIL = DEVOUTPICT (exp1, expC2, [expC3], [expC4])

Purpose:

Outputs a value to the current SET DEVICE using a picture clause.

Arguments:

<**exp1**> is an expression of any data type to display.

<**expC2**> is the formatting rule for output of <exp1>. Refer to the PICTURE clause of the @...SAY command for more information. If <expC2> is not given, or is NIL or empty, DevOutPict() behaves same as DevOut().

Options:

<**expC3**> is an optional argument that defines the display color of <exp1>. If the current DEVICE setting is not SCREEN, the argument is ignored. If not specified, the <exp1> is displayed using the current color setting. Only the first color pair (standard) of <expC3> is significant. See SET COLOR command or SETCOLOR() function for the format of <expC3> string. This argument is always considered in Terminal i/o mode. GUI mode check first for availability of <expC4> and if not set, uses <expC3> only if SET GUICOLOR is ON.

<**expC4**> is an optional argument that defines the display color of <exp1> used in GUI mode.If the current DEVICE setting is not SCREEN, or the AppIoMode() is not "G", the argument is ignored and <expC3> used instead.

Returns:

The function always returns NIL.

Description:

DEVOUTPICT() is similar to the @ ROW(),COL() SAY... PICTURE... command performing a full-screen display at the current cursor (or printhead) position. To set the cursor to other coordinates, use DEVPOS() function.

Example:

```
SET COLOR TO "W+/B"
CLS
@ 10,0 SAY 123.3 PICTURE "999.9" COLOR "GR+/B"
DEVOUTPICT (234.5, "9999.9")
DEVPOS (ROW(), COL() +1)
DEVOUTPICT (DATE(), "99-99-99", "R+/B")
```

Classification: screen oriented or coordinates oriented printer/file output

Compatibility:

Available in FS4, VFS and C5 only. Embedded zero bytes in <expC1> are not supported. The <expC4> argument is available in VFS only.

Related:

@..SAY, DEVOUT(), DEVPOS(), COL(), ROW(), SET COLOR, SETCOLOR()

DEVPOS ()

Syntax:

NIL = DEVPOS (expN1, expN2, [expL3])

Purpose:

Moves the cursor or printhead to a new position.

Arguments:

<**expN1**> and <**expN2**> are the new row and column positions, starting with zero. If the DEVICE is SCREEN, the valid range is 0...MAXROW() and 0...MAXCOL(). If <expL3> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

<expL3> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

Returns:

The function always returns NIL.

Description:

DEVPOS() moves the screen or printhead depending on the current SET DEVICE status.

If SET DEVICE TO SCREEN is active (the default), DEVPOS() behaves like SETPOS(), moving the cursor on the screen and updating the COL() and ROW() values.

If SET DEVICE TO PRINTER is active, DEVPOS() moves the printhead instead. Movement forwards is issued by sending spaces, backwards by sending backspaces CHR(8) and downwards by sending linefeeds CHR(10). If the <expN1> is lower than the current PROW() value, a form-feed CHR(12) is executed first. If the current SET MARGIN value is greater than zero, it is added to <expN2>. The values of PROW() and PCOL() are updated according to the new printhead position.

If the printer is redirected to a file using the SET PRINTER command, DEVPOS() updates the file instead of the printer.

Example:

```
DEVPOS (10,5)                       // equivalent to:
DEVOUT ("my text")                  // @ 10,5 SAY "my text"
```

Classification:

screen oriented or coordinates oriented printer/file output

Compatibility:

Available in FS4, VFS and C5 only. The 3rd parameter is new in VFS.

Related:

@..SAY, SETPOS(), DEVOUT(), DEVOUTPICT(), COL(), ROW(), PCOL(), PROW(), SETPRC()

DIRECTORY ()

Syntax:

`retA = DIRECTORY ([expC1], [expC2])`

Purpose:

Creates an array of directory and file information.

Options:

<**expC1**> is a standard UNIX or Windows wildcard pattern specifying the path and file search. Wildcard characters "?", "*" (and on Unix regular expression like "[...]") can be used. If the argument is not specified, "*" or ".*", depending on the <expC2> argument, is assumed. The automatic path and file conversion using FS_SET() and the DOS drive substitution by x_FSDRIVE environment variable also applies to the <expC2> argument on Unix systems.

<**expC2**> specifies inclusion of files with special attributes in the information returned. <expC2> is a string containing one or more of the following characters:

Attr	FlagShip/Unix	FlagShip/MS-Windows
H	Include .* hidden files	Include hidden files
S	ignored	Include system files
D	Include directories	Include directories
V	ignored	Include DOS volume label (ignor)
A	Date,time = Last access	Date,time = Last access (not FAT)
C	Date,time = Creation	Date,time = Last access (not FAT)

If the "D" attribute is not specified in <exp2>, directories are not included in the search. If specified, also "." and ".." for the current and parent directories are reported in element 1 of the <retA> array. The "A" and "C" attribute is not supported on FAT file system, where the write/modification time is reported instead.

Returns:

<**retA**> is a two-dimensional array, containing information about each file matching <expC1>:

Element	Type	#define	Description
retA [n,1]	C	F_NAME	File name, case sensitive
retA [n,2]	N	F_SIZE	File size in bytes
retA [n,3]	D	F_DATE	Last modification date
retA [n,4]	C	F_TIME	Last modif. time hh:mm:ss
retA [n,5]	C	F_ATT	File attribute, see below

The first element (file name) is in usual convention, i.e. case sensitive on Unix. On MS-Windows, the file name is reported as passed from the system, but the case is irrelevant. Only the file part, without the path is stored in the array element.

The fifth element is filled with the UNIX file attributes (access rights). The elements are of character type in the syntax "drwxrwxrwx" for the directory, owner, group and

other rights; see section LNG.3.3. On MS-Windows, a combination of R (read only), H (hidden), S (system), D (directory), A (archive) or "" for regular file is reported.

If no files are found matching <expC1> or <expC2>, or if the given path is illegal, an empty { } array is returned.

Description:

DIRECTORY() returns information about files in the current or specified directory. It is similar to ADIR(), but returns a single array instead of adding values to a series of existing arrays passed by reference.

Example:

```
#include "directry.fh"
LOCAL dirstru
#ifdef FlagShip
   FS_SET ("lower", .T.)
   FS_SET ("pathlower", .T.)
#else
#  define TRUEPATH UPPER
#endif

? "all files in current directory, including hidden:"
dirstru := DIRECTORY ()
AEVAL (dirstru, {|elem| QOUT(elem[F_NAME], elem[F_SIZE])} )
? "all files t* in the parent directory " + ;
   TRUEPATH("..\") + ":"
dirstru := DIRECTORY ("..\t*", "D")
AEVAL (dirstru, {|elem| QOUT(PADR(elem[F_NAME],30), ;
                 elem[F_SIZE], elem[F_ATT]) } )
? "all index files in the parent directory " + ;
   TRUEPATH("..\") + " with access time:"
dirstru := DIRECTORY ("..\*" + INDEXEXT(), "A")
AEVAL (dirstru, {|elem| QOUT(PADR(elem[F_NAME],30), ;
                 elem[F_SIZE], elem[F_ATT]) } )
```

Classification:

system, file access

Compatibility:

Available in FS4, VFS and C5 only. Note the above differences on the UNIX and DOS file access (support of UNIX wildcard in <expC1> argument, slightly different <expC2> argument, case-sensitive file names in the first returned element, and the UNIX file attributes returned in the 5th element).

Note the differences in file names specified by wildcard on DOS and UNIX: on DOS "*." means "all files without an extension", as opposed to the UNIX interpretation "all files with a trailing dot".

Include:

The constants to #include are in the "directry.fh" file.

Related:

ADIR(), DIR, FILE()

DISKSPACE ()

Syntax:

retN = DISKSPACE ([expC])

Purpose:

Determines the number of bytes left free on the specified Unix file system or MS-Windows disk drive.

Arguments:

<**expC**> is the character string containing the name of the file system you want to determine the free space for.

The automatic path and file conversion using FS_SET() and the DOS drive substitution by x_FSDRIVE environment variable also applies to the <expC> argument. To substitute e.g. second diskette, enter "B:" in <expC> and ensure B_FSDRIVE is assigned to any valid UNIX path and exported before the application is started.

If the argument is not of the character type, or is not specified at all, the default is the current file system (not the file system set by SET DEFAULT).

Return value:

<**retN**> is a numeric value representing the number of free bytes remaining on the specified file system.

Description:

DISKSPACE() is used when you want to determine the amount of free space for the purpose of back-up, copying, sorting file, or any other purpose where disk space is needed.

Example:

```
FUNCTION backup (dbf_name)
LOCAL min
USE (dbf_name)
min = RECSIZE() * RECCOUNT() + HEADER() +1
IF DISKSPACE() < min
   ? "not enough space to copy " + dbf_name + ".dbf"
   WAIT "check the filesystem or use other one"
   USE
   RETURN .F.
ENDIF
COPY TO (dbf_name + ".bak")
USE
RETURN .T.
```

Classification:

system, file access

Compatibility:

C5 uses an optional **numeric** argument, which represents the drive number. On UNIX, there is no equivalent for the drive letter of DOS file system is used instead.

FlagShip supports the DOS drive letters substituting them with an UNIX directory by the environment variable x_FSDRIVE, see LNG.3.2 and LNG.9.5.

Related:

HEADER(), RECCOUNT(), RECSIZE(), FS2:DiskFree()

DISPBEGIN ()

Syntax:

NIL = DISPBEGIN ( )

Purpose:

Begins buffering screen output.

Returns:

The function always returns NIL.

Description:

DISPBEGIN() is a screen function that starts buffering the subsequent screen output instead of displaying it, until DISPEND() is executed. Applicable in Terminal i/o, ignored otherwise.

The output buffering using DISPBEGIN() and DISPEND() can be used to prepare a complex display in the "background". This is especially useful with a slow communication line.

The REFRESH command or depressing of the ^K (abort key) or ^O (debugger) will force the display of buffered data, but will not terminate buffering.

It is not recommended to output an user prompt (such as WAIT, READ, ACCEPT, INPUT etc.) using DISPBEGIN(). Ensure DISPEND() is issued prior to issuing such commands or functions.

Example:

```
DISPBEGIN()            // Start screen buffering
CLS
@ 10,10 say "This is my output"
SETPOS(11, 10)
DISPOUT("An another text")
?? " continued"
? "or printed at a new line
DISPEND()              // Display buffered screen data
```

Classification:

screen oriented output

Compatibility:

Available in FS4, VFS and C5 only. Nested calls of DISPBEGIN() are not supported by FS4.

Related:

DISPEND(), REFRESH, SYS.2

DISPBOX ()

Syntax:

```
NIL = DISPBOX (expN1, expN2, expN3, [expN4],
               [expCN5], [expC6], [expL7], [expL8],
               [expCN9], [expN10], [expN11], [expL12])
```

Purpose:

Draws a customized box on the screen.

Arguments:

<**expN1**> to <**expN4**> are the screen coordinates, upper, left, lower, and right respectively. The valid range is 0, 0, MAXROW(), MAXCOL(). If <expL7> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

Options:

<**expCN5**> specifies the box outfit and may be given as a character expression <expC5> or as a numeric expression <expN5>. If not specified, a single-line box is drawn.

<**expC5**> is a character string containing eight border characters and one fill character. The first character is used for the upper left corner, the next for the upper line, and so on, clockwise. The box is filled with the ninth character.

<**expN5**> is a numeric value of 1 which displays a single-line box or a value of 2 displays a double-line box. All other numeric values are assumed to be 1.

<**expC6**> is an optional color specification (according to SET COLOR). If not specified, the box is drawn using the current color setting.

<**expL7**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

<**expL8**> how to draw: .T. = draw only in GUI mode w/o considering SET GUITRANSL *, .F. = draw only in Terminal mode, NIL = draw in Terminal mode always, in GUI considering SET GUITRANSL BOX/LINES

<**expCN9**> color specification same as <expC5> but considered in GUI mode only

<**expN10**> is the frame type, considered in GUI mode only:

= 0: draw plain frame, same as @..BOX..PLAIN
= 1: draw sunken frame, same as @..BOX..SUNKEN
= 2: draw raised frame, same as @..BOX..RAISED

<**expN11**> is the line width, considered in GUI mode only, same as @..BOX..LINEWIDTH nn. If not specified, the default width (usually 2) is used. If specified 0, only the box inner area is filled.

<**expL12**> if specified .T., only the frame is drawn, the inner box area is not filled by the specified of default color. If .F. or not specified, the <expCN9>, <expC6> or the default background color is used to fill the inner box. Equivalent to @..BOX..FRAMEONLY command. Considered in GUI mode only.

Returns:

The function always returns NIL.

Description:

DISPBOX() is equivalent to the @...BOX command and is used for drawing boxes using a configurable border and filling it with a specified character.

After DISPBOX() executes, the cursor and ROW(), COL() are set into the boxed region at <expN1> +1, <expN2> +1.

In GUI mode, you may:

- use sunken, raised or plain box frame, when <expN10>, <expN11> or <expL12> are specified. The background color of <expCN9> is considered and overrides <expC5>, filling character of <expC5> is ignored.
- use the semi-graphic characters in <expC5>, simulated via line drawing, when SET GUITRANSL BOX is ON, and neither <expN10>, <expN11> nor <expL12> are specified. The background color and filling character in <expC5> is ignored, since almost unwanted results occurs with proportional fonts. This is the "old", backward compatible syntax.

In Terminal i/o mode, the box is always drawn by the <expC5> chars. The optional <expCN9> to <expL12> parameters are ignored. The color set by <expC6>, as well as the fill character of <expC5> are considered.

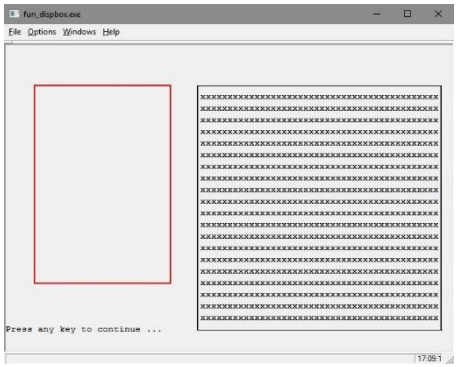
Note that DispBox() does not create new widget (control) but draws a rectangle filled by the specified color directly in the user window (or in current sub-window). It frame may therefore be overwritten by subsequent @..SAY, ?, ?? or Qout() output. If you wish to create new widget (sub-window) with protected frame, use either Wopen() from the FS2 Toolbox, or it subset MDIopen() for MDI based GUI application.

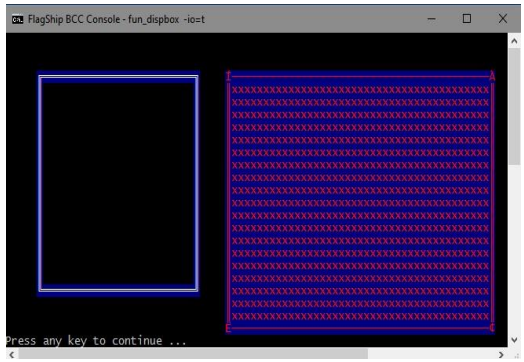
For more information, refer to the @...BOX command.

Example:

```
#include "box.fh"
set font "Courier", 10        // GUI font
oApplic:Resize(25,80,,.T.)    // resize screen to 25x80

SET GUITRANS LINE ON          // required for GUI
SET GUITRANS BOX  ON          // required for GUI

DISPBOX ( 3, 5, 20,30, 2, "W+/B", , , "R+")
DISPBOX ( 3,35, MaxRow()-1,MaxCol()-2, B_SINGLE_DOUBLE+"x", "R+/B")

setpos(23,0)
wait
```

Output:





Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND()

Compatibility:

Available in FS4, VFS and C5. The physical output on the screen depends on the chosen terminal description (environment variable TERM) and the ability of the terminal to output mapping applied via FSchrmap.def. See also LNG.5.1.4, section SYS, and FS_SET ("outmap"). All the 7th to 12th parameters are new in VFS and n/a in Clipper.

Include:

The #include-able box constants are available in the "box.fh" file

Related:

@...BOX

DISPCOUNT ()

Syntax:

`retN = DISPCOUNT ( )`

Purpose:

Returns the number of pending DISPEND() requests.

Returns:

<**retN**> is the number of DISPEND() calls required to restore the original display context.

Description:

DISPCOUNT() returns 1 if the output buffering using DISPBEGIN() is active, otherwise 0. Applicable in Terminal i/o, ignored otherwise.

Example:

```
IF DISPCOUNT() > 0         // output buffering active ?
   DISPEND()               // yes, terminate it
ENDIF
WAIT
```

Classification:

programming, screen oriented output

Compatibility:

Available in FS4, VFS and C5 only. Nested calls of DISPBEGIN() are not supported by FS4.

Related:

DISPBEGIN(), DISPEND()

DISPEND ()

Syntax:

NIL = DISPEND ( )

Purpose:

Displays buffered screen updates, terminate buffering.

Returns:

The function always returns NIL.

Description:

DISPEND() is a screen function that terminates the screen buffering of DISPBEGIN() and writes the "background" data to the screen. If buffering is not active, no action is performed. Applicable in Terminal i/o, ignored otherwise.

The output buffering using DISPBEGIN() and DISPEND() can be used to prepare a complex display in the "background". This is especially useful with a slow communication line.

The REFRESH command or depressing of the ^K (abort key) or ^O (debugger) will force the display of buffered data too, but will not terminate the buffering.

It is not recommended, to output an user prompt (like WAIT, READ, ACCEPT, INPUT etc.) using DISPBEGIN(). Ensure that DISPEND() is issued prior to such commands or functions.

Example:

```
DISPBEGIN()            // Start screen buffering
CLS
@ 10,10 say "This is my output"
SETPOS(11, 10)
REFRESH                // see the intermediate state
DISPOUT("An another text")
?? " continued"
? "or printed at a new line
DISPEND()              // Display buffered screen data
```

Classification:

screen oriented output

Compatibility:

Available in FS4, VFS and C5 only. Nested calls of DISPBEGIN() are not supported by FS4.

Related:

DISPBEGIN(), REFRESH

DISPOUT ()

Syntax:

NIL = DISPOUT (exp1, [expC2], [expC3])

Purpose:

Outputs a value on the screen.

Arguments:

<**exp1**> is an expression of any data type to display.

Options:

<**expC2**> is an optional argument that defines the display color of <exp1>. If not specified, the <exp1> is displayed using the current color setting. See SET COLOR command or SETCOLOR() function for the format of <expC2> string. Considered for Terminal i/o mode only. GUI mode check first for availability of <expC3> and if not set, uses <expC2> only if SET GUICOLOR is ON.

<**expC3**> is an optional argument that defines the display color of <exp1> used in GUI mode. This argument is considered in GUI mode only, otherwise <expC2> is used.

Returns:

The function always returns NIL.

Description:

DISPOUT() is similar to DEVOUT() function or the @ ROW(),COL() SAY.. command performing a full-screen display at the current cursor position. Unlike these, DISPOUT() ignores the SET DEVICE setting; output always goes to the screen. To set the cursor to other coordinates, use SETPOS() function.

Example:

```
SET COLOR TO "W+/B"
CLS
@ 10,0 SAY "First part" COLOR "GR+/B"
DISPOUT(" second part")
SETPOS (ROW(), COL() +2)
DISPOUT("third part", "R+/B")
```

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND()

Compatibility:

Available in FS4, VFS and C5 only.

Related:

@..SAY, SETPOS(), DEVOUT(), DEVPOS(), COL(), ROW(), SET COLOR, SETCOLOR()

DOSERROR ()

Syntax:

`retN = DOSERROR ()`

Purpose:

Determines the number of the last UNIX (or MS-Windows) operating system error.

Returns:

<**retN**> is a numeric value, representing the last error number.

Description:

DOSERROR () can be used to determine the exact cause of a file use error. For the error message text, use DOSERROR2STR().

Example:

```
Dos_err = DOSERROR()
DO CASE
   CASE Dos_err = 1
      <statements...>
   CASE Dos_err = 2
      <statements...>
   .
   .
   .
ENDCASE
```

Classification:

programming, system access

Compatibility:

DOS and UNIX operating systems may return different error numbers. FlagShip also supports return codes from the RUN, COPY TO and RENAME commands.

Related:

DOSERROR2STR(), FERROR()

DOSERROR2STR ()

Syntax:

`retC = DOSERROR2STR ()`

Purpose:

Determines the number of the last UNIX (or MS-Windows) operating system error and returns it textual description, if any.

Returns:

<**retC**> is a string, representing the description of last error number or "" if no error occurred.

Description:

DOSERROR() and DOSERROR2STR() can be used to determine the exact cause of a file use error.

Example:

```
COPY file1 TO file2
if DOSERROR() != 0            // on failure, display the reason
    ? "could not copy file1 to file2,"
    ? "error = " + ltrim(DOSERROR()) + ": " + DOSERROR2STR()
endif
```

Classification:

programming, system access

Compatibility:

Available in FlagShip VFS6 and newer. MS-DOS and UNIX operating systems may return different error codes and/or descriptions. FlagShip also supports return codes from the RUN, COPY TO and RENAME commands.

Related:

DOSERROR(), FERROR()

DOW ()

Syntax:

`retN = DOW (expD)`

Purpose:

Returns the ordinal number of the day of the week for a date value.

Arguments:

<**expD**> is the date value to convert.

Returns:

<**retN**> is a number between 1 and 7. The first day of the week is Sunday (one), the last day is Saturday (seven). If the date value is empty or invalid, 0 (zero) is returned.

Description:

DOW () can be used whenever the day of week is needed for calculations, or when the name of a day of the week is needed in other languages than reported by CDOW() function.

Example 1:

```
/* Determine the date of Monday in previous week and
 * the date of Wednesday in over-next week. Calculates
 * also dates begin-of-week (dBoW) and end-of-week (dEoW).
 */
local dCurr, dEoW, dBoW, dResult
dCurr := DATE()              // today's date
dEoW  := dCurr -1            // starting yesterday,
while DOW(dEoW) != 7         // find prev Saturday (= end of week)
   dEoW--
enddo
dResult := dEoW -5           // diff Saturday back to Monday
? "Today is",dCurr, "Monday of last week was the", dResult

dCurr := DATE()              // today's date
dBoW  := dCurr +1            // starting tomorrow,
while DOW(dBoW) != 1         // find next Sunday (= begin of week)
   dBoW++
enddo
dResult := dBoW +3 +7        // Sunday to Wednesday + 1 week
? "Today is",dCurr, "Wednesday in the week after next is", dResult
wait
```

Example 2:

See also example in CDOW() function

```
DECLARE aDays[7]
german = .T.
aDays[1] = if (german, "Sonntag",   "Sunday")
aDays[2] = if (german, "Montag",    "Monday")
aDays[3] = if (german, "Dienstag",  "Tuesday")
aDays[4] = if (german, "Mittwoch",  "Wednesday")
aDays[5] = if (german, "Donnerstag","Thursday")
aDays[6] = if (german, "Freitag",   "Friday")
```

```
aDays[7] = if (german, "Samstag",   "Saturday")
SET DATE GERMAN
? DATE()                                        // 27.09.13
? DOW(DATE())                                   // 6
? "Today is", aDays[DOW(DATE())]                // "... Freitag"
? "Yesterday was", aDays[DOW(DATE() -1)]        // "... Donnerstag"
cNextYear := ltrim(YEAR(DATE()) +1)             // "2014"
? "Next New Year " + cNextYear + " will be on " + ;
  aDays[DOW(CTOD("01/01/" + cNextYear))]        // "... Mittwoch"
```

Classification:

programming

Related:

CMONTH(), CTOD(), DATE(), DAY(), DOW(), DTOC(), DTOS(), MONTH(), YEAR()

DRAWLINE ()

Syntax:

```
NIL = DrawLine ([expN1], [expN2], expN3, expN4,
           [expN5], [expC6], [expL7])
```

Fully equivalent to GuiDrawLine(), see description for

```
NIL = GuiDrawLine ([expN1], [expN2], expN3, expN4,
           [expN5], [expC6], [expL7])
```

or for the command

```
@ <expN1>,<expN2> [GUI]
      DRAW [TO] <expN3>,<expN4>
      [COLOR <color>]
      [PIXEL|NOPIXEL]
      [WIDTH <wpix>]
```

DTOC ()

Syntax:

retC = DTOC (expD)

Purpose:

Converts a date value to a character string.

Arguments:

<**expD**> is the date value to convert.

Returns:

<**retC**> is a character string, representing the date value. The form of the returned string depends on the status of SET DATE, SET CENTURY and SET EPOCH. If a empty date is specified, a string of eight or ten spaces is returned, depending on the status of SET CENTURY.

Description:

DTOC () can be used when a date has to be displayed as a part of a character expression.

For indexing purposes, where a date is a part of a complex key, use DTOS () instead of DTOC () to build a proper index order.

Example:

```
? DATE()                                    &&  09/20/93
? DTOC(DATE())                              && "09/20/93"
? DTOS(DATE())                              && "19930920"
mytext := "today is " + DTOC(DATE())

SET DATE GERMAN
? DATE()                                    &&  20.09.93
? DTOC(DATE())                              && "20.09.93"
? DTOS(DATE())                              && "19930920"
mytext := "heute ist der " + DTOC(DATE())
```

Classification:

programming

Related:

SET CENTURY, SET DATE, SET EPOCH, CDOW(), CTOD(), CMONTH(), DATE(), DAY(), DOW(), DTOS(), MONTH(), YEAR()

DTOS ()

Syntax:

retC = DTOS (expD)

Purpose:

Converts a date value to a character string suitable for indexing.

Arguments:

<**expD**> is the date value to convert.

Returns:

<**retC**> is a character string, representing the date value. The form of the returned string is "yyyymmdd". If a null date is specified, DTOS () returns SPACE(8).

Description:

Strings returned by DTOS () preserve the relations between date values, which makes them suitable for use as parts of more complex indexing key expressions. The returned string is always formatted to eight characters as "YYYYMMDD" and is not affected by the current SET DATE and SET CENTURY format.

The complementary function is STOD().

Example:

```
? DATE()                                        &&  09/20/93
? DTOC(DATE())                                  && "09/20/93"
? DTOS(DATE())                                  && "19930920"
USE article
INDEX ON DTOS(publ_date) + STR(magazine) TO article
```

Classification:

programming

Related:

STOD(), SET CENTURY, SET DATE, SET EPOCH, CDOW(), CTOD(), CMONTH(), DATE(), DAY(), DOW(), DTOC(), MONTH(), YEAR()

EMPTY ()

Syntax:

`retL = EMPTY (exp)`

Purpose:

Determines if the result of an expression is empty.

Arguments:

<**exp**> is an expression of any type.

Returns:

<**retL**> is a logical value, which is TRUE in case of:

Data Type		Expression contents
Array	A	Zero-length { }
Character	C	"" or spaces and tabs only
Numeric	N,I,F	0
Date	D	Null date, e.g. CTOD("")
Logical	L	False value (.F.)
Memo	M	Same as character
Screen	S	empty
Code block	B	not code block
NIL	U	NIL

Description:

The EMPTY() function can be used to determine if the variable or field is properly filled, formal parameters passed or an array declared.

To determine if a PRIVATE, PUBLIC or FIELD variable is declared or visible, TYPE("varname") can additionally be used.

Example:

```
? EMPTY (0)                          &&    .T.
? EMPTY (.F.)                        &&    .T.
? EMPTY (.T.)                        &&    .F.
? EMPTY ("")                         &&    .T.
? EMPTY ("   ")                      &&    .T.
? EMPTY (CTOD(""))                   &&    .T.
? EMPTY ( {} )                       &&    .T.
```

Classification:

programming

Related:

LEN(), TYPE(), VALTYPE(), ISFUNCTION()

EOF ()

Syntax:

`retL = EOF ()`

Purpose:

Reports an attempt to move past the end of the current database file.

Returns:

<**retL**> is a logical value returning TRUE when the record pointer was moved beyond the last logical record, or when the database is empty or filtered out. In such a case, the record pointer is positioned beyond the database on a "phantom" record LASTREC() +1.

Description:

EOF() is used wherever a boundary condition test is needed when the record pointer is moving forward through a database. Any command that can move the record pointer can set EOF().

The EOF() condition check should be used to determine whether a SKIP, GOTO BOTTOM, SEEK, FIND, or LOCATE command failed. You should check it also before REPLACE-ing field values, since fields of not available records cannot be changed - except SET EOFAPPEND is ON. Once EOF() is set to TRUE, it retains its value until there is another attempt to move the record pointer.

On an empty or filtered out database, both EOF() and BOF() returns TRUE.

To examine the EOF() status in a working area other than the current one, use alias->(EOF()).

Example:

```
USE article NEW
COPY STRUCTURE TO emptydb
USE empty NEW
SELECT article
? RECNO(), LASTREC()                          &&   1 100
GOTO BOTTOM
? RECNO(), EOF()                              && 100 .F.
SKIP 5
? RECNO(), EOF()                              && 101 .T.
? emptydb->(RECNO()), emptydb->(EOF())  &&   1 .T.
```

Classification:

database

Related:

BOF(), FOUND(), SKIP, GOTO, LASTREC(), SET EOFAPPEND, oRdd:Eof

ERRBOX ()

Syntax:

`retN = ErrBox (expC1, [expC2], [expN3])`

Purpose:

Display an error message box dialog, similar to Alert() or InfoBox()

Arguments:

<**expC1**> is a string containing the displayed text, the lines are separated by semicolon ";". See additional details in Alert() and InfoBox() description.

<**expC2**> is an optional header text.

<**expN3**> is optional time-out in seconds. Zero or NIL specify to wait until user press key or mouse selection. When the time-out expires without user action, the box is closed and <retN> is set to 0.

Description:

The ErrBox() function is available for your convenience. It is equivalent to ErrorBox{NIL,cMessage}:Show() described in section OBJ. In GUI mode, error icon (red circle with cross) is displayed. In Terminal i/o mode, the color is specified in global variable

```
  _aGlobSetting[GSET_T_C_ERRBOXCOLOR] := "GR+/R,R/W"  // default
```

and can be re-assigned according to your needs.

Example:

```
_aGlobSetting[GSET_T_C_ERRBOXCOLOR] := "GR+/R,R/W"  // default
_aGlobSetting[GSET_T_C_ERRBOX     ] := B_SINGLE

 ErrBox (";Caution: unexpected error occured;" + ;
         "Please notify developer", "Unexpected Error")
```

Output:

Compatibility: New in VFS. The expN3 is supported since VFS7

Source:

The source code is available in <FlagShip_dir>/system/boxfunct.prg

Related:

Alert(), InfoBox(), WarnBox(), TextBox(), OBJ.MessageBox{}, InfoBox{}, ErrorBox{}, WarnBox{}, TextBox{}

ERRORBLOCK ()

Syntax:

retB = ERRORBLOCK ([expB])

Purpose:

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

Options:

<**expB**> is the code block to execute whenever a runtime error RTE occurs. When evaluated, the <expB> block is passed an error object as an argument by the system. If the argument is not specified, the default error handling block is evaluated on RTE.

Returns:

<**retB**> is a pointer to the current error handling code block. If no error handling block has been posted by the user, the pointer to the default error handling block is returned. The code block returned can be re-assigned later using ERRORBLOCK(), see example.

Description:

ERRORBLOCK() is an error function that defines an error handler to execute whenever a runtime error (RTE) occurs. The error block has the syntax:

```
{|<expO>| <exp>, ... }
```

where <**expO**> is an error object (see section OBJ) containing information about the error passed by the system. The <**exp**>, executed on RTE, is any single expression, expression list or a UDF. The code block must return TRUE to retry the failed operation or FALSE to terminate the process.

Within the UDF executed by the code block body, a BREAK can be invoked to pass the program control to the RECOVER (or the END) statement of the next BEGIN SEQUENCE ...END structure.

If the <expB> argument is not specified, the default error handling block is evaluated. It displays a descriptive message to the screen, sets the ERRORLEVEL() to 1, then QUITs the program.

Example:

```
LOCAL olderr
LOCAL newerr := {|errobj| my_error (errobj)}
olderr := ERRORBLOCK (newerr)

BEGIN SEQUENCE
   USE nonexist                          // will fail
   SKIP
RECOVER USING recovobj
   ? "sorry."
   QUIT
END
```

```
ERRORBLOCK (olderr)                    // restore
RETURN
FUNCTION my_error (err)
? err:description
BREAK                                  // jump to RECOVER
RETURN NIL
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

ERRORBOXNEW ()

Syntax:

```
retO = ErrorBoxNew ([oOwner], [cText])
```

Description:

This is alternative syntax for creator of ErrorBox class, see details in section OBJ

Related:

ErrorBox Class, ErrBox()

ERRORLEVEL ()

Syntax:

`retN = ERRORLEVEL ([expN])`

Purpose:

Sets error level (exit code) on return to the UNIX or Windows shell.

Arguments:

<**expN**> is a numeric value between 0 and 255, specifying the code setting. The default value for the normal exit is zero. Fatal or run-time error sets exit code 1 or 2, other hard errors 3 to 9. The InitIoQuit() function (or your re-defined UDF) also sets exit code on user abort, see details in <FlagShip_dir>/system/initiomenu.prg. If <expN> is not specified, ERRORLEVEL() reports the current setting without assigning a new value.

Returns:

<**retN**> is the current ERRORLEVEL() setting.

Description:

On program termination, the application can set a return code which is passed to the UNIX shell variable "$?" or in Windows to %ERRORLEVEL% shell variable. If the ERRORLEVEL() is not set, the $? or %ERRORLEVEL% examines 0.

Tuning:

Most of the default exit codes are re-definable by assigning

```
  _aGlobSetting[GSET_N_RETURN_DEFAULT ] := 0    // default exit
  _aGlobSetting[GSET_N_RETURN_FATAL   ] := 1    // fatal error
  _aGlobSetting[GSET_N_RETURN_RTE     ] := 2    // run-time error
  _aGlobSetting[GSET_N_RETURN_CTRL_K  ] := 11   // closed via ^K
  _aGlobSetting[GSET_N_RETURN_CLOSE_EV] := 12   // closed via event
  _aGlobSetting[GSET_N_RETURN_MENU    ] := 13   // closed via menu
  _aGlobSetting[GSET_N_RETURN_ABORT   ] := 14   // closed via abort
```

see also <FlagShip_dir>/system/initio.prg and initiomenu.prg

Example:

Application started by a simple shell script "run.sh":

```
#!/bin/bash
# *** file "run.sh", make it executable using "chmod +x" ***
./a.out           # or other executable
retcd = $?        # return code from a.out
if [ retcd -eq 0 ] ; then
   echo Program terminated normally
elif [ retcd -eq 1 ] ; then
   echo Run-time error occurred, contact programmer
elif [ retcd -eq 200 ] ; then
   echo Backup required
   echo Insert a backup tape, press any key when ready...
   read
   tar cvf /dev/tape *.dbf *.idx
   echo Done. Remove tape.
```

```
fi
# eof file "run.sh"

*** file test.prg ***
LOCAL answer := "n"
ACCEPT "backup required (y/n) ? " TO answer
IF UPPER(answer) == "Y"
   ERRORLEVEL (200)
ENDIF
QUIT
```

Classification:

programming

Compatibility:

Available in FlagShip and Clipper5 only.

Related:

QUIT, CANCEL, RETURN

EVAL ()

Syntax:

ret = EVAL (expB1, [exp2 [, exp3...]])

Purpose:

Evaluates a code block.

Arguments:

<**expB1**> is the code block (see LNG.2.3.3) to evaluate.

Options:

<**exp**2...> is an optional list of arguments to be passed to the code block.

Returns:

<**ret**> is the returned code block value. If an expression list (see LNG.2.8) is declared in the code block body, the value of the rightmost expression is returned.

Description:

Code blocks are functionally similar to UDFs. They are special **unnamed** inline functions, and are compiled to native code or evaluated at run-time as a macro, see LNG.2.3.3. The reference to the code block can be stored in usual the FlagShip variables or passed as parameter to standard or user defined functions and evaluated there.

Since the code block has no name, its execution cannot be invoked as a usual (named) call of a procedure or function. Instead, the EVAL() or AEVAL(), DBEVAL() invocation must be used to execute the code block body. The required parameters for the code block, if any, are passed by the second, third, etc. parameter of EVAL().

Example:

```
LOCAL txt := "my text"
LOCAL arr := {"elem 1", "elem 2"}
LOCAL blk := {|par| QOUT(par) }
LOCAL xxx := 0

EVAL (blk, txt)                                 // "my text"
EVAL (blk, arr[2])                              // "elem 2"
? EVAL ({|| xxx++, VALTYPE(txt)}), xxx          // "C"  1
? EVAL ({|temp| temp := 5, --temp })            // 4
? EVAL ({|par, temp| temp := par + 5}, 2)       // 7
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

AEVAL(), DBEVAL(), ASCAN(), ASORT()

EXECNAME ()

Syntax:

retC = EXECNAME ([expL1], [expL2])

Purpose:

Retrieves the name of the currently running executable.

Options:

<**expL1**> specifies the kind of the return value. If TRUE, the returned executable name includes the path, if it was given by the command line (or script) invocation. If <expL1> is not given or specified FALSE, only the file name is returned, regardless of the command line invocation.

<**expL2**> forces resolving symbolic links to return original destination. Considered in Unix/Linux only with enabled <expL1> = .T.

Returns:

<**retC**> is the file name of the currently running executable (e.g. "a.out", "test.exe"), optionally preceded by the path (e.g. "/usr/ data/myexe" or "/home/foo/a.out" or "C:\my data\applic\test.exe"). If the executable was invoked w/o path, the PATH environment is searched for, when <expL1> is TRUE. Relative paths are translated to absolute.

Description:

The function determines the name of the current executable.

Note: a Unix executable may carry any valid name, the access attribute is "x". The default name is "a.out", if the compiler option -o is not set. Windows executable usually has .EXE extension.

Example:

```
? ExecName()            // “a.out” or “myapplic.exe”
? ExecName(.T.)         // /usr/bin/myapplic (but can be symb.link)
? ExecName(.T., .T.)    // /home/john/data/myapplic  (original)
? FindExeFile(ExecName()) // C:\Program Files\MyApplic.exe or ""
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

ExecPidNum(), FindExeFile(), section FSC

EXECPIDNUM ()

Syntax:

`retN = EXECPIDNUM ( )`

Purpose:

Retrieves the PID (process id number) of the currently running executable.

Returns:

<**retN**> is the process identification number of the currently running executable.

Description:

Any process (including the running or sleeping executables) is automatically identified by Unix (and Windows/32) with a unique PID. This PID number is also used in FlagShip as the file name extension of the default spooler file; see also SET PRINTER and FS_SET("printfile").

Example:

```
? ExecName ()                       // "a.out" or “myapp.exe”
? ExecPIDnum ()                     // 1753
? fs_set ("printfile")              // "mytest.1753"
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

ExecName(), SET PRINTER, FS_SET(), LNG.3.4

EXP ()

Syntax:

`retN = EXP (expN)`

Purpose:

Calculates exponentiation expN (the base of natural logarithms).

Arguments:

<**expN**> is the numeric expression, specifying the natural logarithm value.

Returns:

<**retN**> is a numeric value, representing the result of exponentiation.

Description:

EXP() raises the base of natural logarithm, e by the given value <expN> (e = 2.7182818285...).

The inverse operation of EXP() is LOG().

The default number of decimals displayed depends on the current SET DECIMALS value.

Example:

```
? EXP (0)                                   && 1.00
? EXP (1)                                   && 2.72
SET DECIMALS TO 6
? EXP (1)                                   && 2.718282
? EXP (50)                                  && 518470552858704690000.
? EXP (LOG(60))                             && 60.000000
```

Classification:

programming

Related:

LOG(), SET DECIMALS, SET FIXED

FCLOSE ()

Syntax:

retL = FCLOSE (expN)

Purpose:

Closes an open binary file.

Arguments:

<**expN**> is the file handle number, previously returned by FOPEN () or FCREATE ().

Returns:

<**retL**> is logically FALSE if there was an error while writing the file buffers to disk, otherwise, TRUE is returned.

Description:

FCLOSE() is a low-level file function that closes binary files and writes the associated UNIX buffers to disk. If FCLOSE() fails, the function returns .F. and FERROR() can then be used to determine the reason for the failure.

Warning: FCLOSE() allows a low-level access to the file system, similar to the C function close(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Example:

```
art_handle = FCREATE("article.doc")
FWRITE (art_handle, "new text")
? FCLOSE(art_handle)                    && .T.
```

Classification:

programming

Related:

FCREATE(), FOPEN(), FERROR(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

FATTRIB ()

Syntax:

`retN = FATTRIB ([expN1])`

Purpose:

Retrieves the access rights associated to an open file or the global creation rights according to 'umask'.

Option:

<**expN1**> is the handle number, previously returned by FOPEN() or FCREATE(). If not given or NIL, the global creation rights according to 'umask' are returned.

Returns:

<**retN**> are the access rights associated to the given file, or set for newly created files, in a semi-octal notation. The semi-octal notation includes three digits (for the owner, group, world), each in the range 0..7. 0 digit specifies no permission, 1 = execute, 2 = write only, 4 = read only, 5 = read/ execute, 6 = read/write, 7 = read/write/execute. For example 664 = "rw-rw-r--".

Description:

Every file and directory in Unix has its own access rights (see section LNG.3.3), set during the file creation or via the 'chmod' Unix command.

FATTRIB() determines either the global rights, or specific access rights of an already opened file. It may be used to transfer permissions while creating a new file via FCREATE(). To determine the access rights of a file given by name, use the DIRECTORY() function.

Example:

Transfer access rights to newly created file

```
iHandle := FOPEN ("myfile.bin", 0)
if iHandle > 0
   ? "transferring rights =", iAttrib := FATTRIB (iHandle)
   FCLOSE (iHandle)
else
   ? "using default rights =", iAttrib := FATTRIB()
endif
iHandle := FCREATE ("newfile.bin", iAttrib)
```

Classification:

programming, file access

Compatibility:

Available in FlagShip only.

Related:

FOPEN(), FCREATE(), DIRECTORY(), Unix man pages 'umask' and 'chmod'

FCOUNT ()

Syntax:

`retN = FCOUNT ()`

Purpose:

Determines the number of fields in the current database file.

Returns:

<**retN**> is a numeric value, representing the number of database fields open in the current working area. If a database is not in use, FCOUNT () returns zero.

Description:

FCOUNT () is useful where serial operations on the fields are needed, or to operate data-independently on the database fields.

To execute the function in a different working area than the current one, use alias->(FCOUNT()).

Example:

```
USE article
? FCOUNT()                                  &&    10
COPY STRUCTURE EXTENDED TO artstruc
USE artstruc NEW
? RECCOUNT()                                &&    10
? article->(RECCOUNT())                     &&    10
```

Example:

```
LOCAL fldnum, fldarr, ii
USE address
fldnum := FCOUNT()
fldarr := ARRAY (fldnum)                    // create array
AFIELDS (fldarr)
FOR ii := 1 TO fldnum
   ? FIELDNAME(ii), fldarr[ii]              // field name
   ? TYPE (FIELDNAME(ii))                   // field type
NEXT
```

Classification:

database

Related:

FIELDNAME(), FIELDGETARR(), TYPE(), oRdd:Fcount

FCREATE ()

Syntax:

`retN = FCREATE (expC1, [expN2])`

Purpose:

Creates a new binary file, or truncates an existing file to zero length.

Arguments:

<**expC1**> is the name of the file to be created. If the file already exists, its length is truncated to zero without warning. An optional path and extension can be specified. The FS_SET() file and path translation is considered.

Options:

<**expN2**> are the resulting file access rights:

value	#define	FS access rights	(DOS attr)
not given		accord. to "umask"	normal
0	FC_NORMAL	rw- rw- ---	normal
1	FC_READONLY	r-- r-- ---	read only
2	FC_HIDDEN	rw- --- ---	hidden
4	FC_SYSTEM	r-- --- ---	system
100..777 *		accord.to value *	
other		rw- rw- rw-	

(*) The semi-octal notation, if given, includes three digits (for the owner, group, world), each in the range 0..7. 0 digit specifies no permission, 1 = execute, 2 = write only, 4 = read only, 5 = read/ execute, 6 = read/write, 7 = read/write/execute. For example 664 = "rw-rw-r--". See also LNG.3.3.

If <expN2> is not given, "rw-rw-rw-" (666) rights are used.

Returns:

<**retN**> is the file handle of the newly created and opened file, in the range from zero to 65535. If an error occurs, -1 is returned. The new file is left opened in the read/write mode with the selected access rights. For other open modes, see FOPEN().

Description:

FCREATE () returns the opened file's handle, which should be assigned to a variable, since it is needed to identify the file in other file functions. The new file is open in the read/write mode with the access rights selected. An additional read and/or write locking using FLOCKF() is then available.

If FCREATE() fails, -1 is returned, and FERROR() can then be used to determine the reason for the failure.

Like other low-level file functions, FCREATE() does not consider either the SET DEFAULT or SET PATH settings.

Warning: FCREATE() allows a low-level access to the file system, similar to the C function open(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Example:

```
#include "fileio.fh"
#include "error.fh"
art_handle = FCREATE("article.doc", FC_NORMAL)
IF art_handle = -1
   ? "The file cannot be created!"
   DO CASE
   CASE FERROR() = EX_EACCES
      ?? " (Permission denied)"
   CASE FERROR() = EX_EROFS
      ?? " (Read-only file system)"
   CASE FERROR() = EX_EMFILE
      ?? " (Too many files for process, adapt the kernel)"
   ENDCASE
   QUIT
ENDIF
```

Classification:

system, file access

Compatibility:

Note the UNIX access rights, not available in DOS. See also section SYS: tunable UNIX parameters to set the number of simultaneously open files.

Include:

<FlagShip_dir>/include/fileio.fh #define's for <expN2> <FlagShip_dir>/include/error.fh #define's of error codes

Related:

FATTRIB(), FCLOSE(), FERROR(), FERROR2STR(), FOPEN(), FREAD(), FREADSTR(),FSEEK(), FWRITE(), FLOCKF()

FEOF ()

Syntax:

`retL = FEOF ( expN )`

Purpose:

Determines if end-of-file was reached by textual/binary file read FREAD*() operations.

Parameter:

<expN> is a handle number, previously returned by FOPEN() or FCREATE(). If not given or is not numeric, the global status returned by FERROR() is used instead, which may not implicitly signal end of file.

Returns:

<**retL**> is logical value, where .T. signals reaching end-of-file by low-level read functions Fread(), FreadStr() or FreadTxt().

Description:

FEOF() is used to terminate read function after reaching end of the file. When the file handler is invalid, the return value is equivalent to FERROR() != 0. When the file handler is given, FEOF() returns .T. when the current file pointer is at or behind last byte of the file.

Example:

```
#include "fileio.fh"
LOCAL handle, text
handle = FOPEN("myfile.txt", FO_READ)
DO WHILE !FEOF(handle)
   ? text := FREADTXT (handle)
ENDDO
FCLOSE (handle)
```

Classification:

system, file access

Compatibility:

Available in FlagShip only, compatible to FoxPro

Related:

FOPEN(), FCREATE(), FERROR(), FREAD(), FREADSTR(), FREADTXT()

FERASE ()

Syntax:

retN = FERASE (expC, [expL2])

Purpose:

Delete a file from disk.

Arguments:

<**expC**> is the name of the file to be deleted from disk, including the extension. The full path may be specified. If omitted, only the current directory is searched; the SET PATH or SET DEFAULT setting is ignored. Standard UNIX wildcards using ?, *, [..] are supported. The FS_SET() file and path translation is considered.

Option:

<expL2> is optional logical value. If .T., run-time warning msg is displayed on failure. Default is .F. which signals failure by the return code only.

Returns:

<**retN**> is zero if the operation succeeds. If FCREATE() fails, the value returned is equivalent to the return code of FERROR() or DOSERROR(). It can then be used to determine the reason for the failure.

Description:

FERASE() is equivalent to the ERASE or DELETE FILE command but returns a value and can be specified within an expression.

The file will be deleted without any warning. The consequences are not recoverable. Only closed files can be deleted. The user must have at least the access rights "w" to the file and "x" in the directory.

Example:

```
IF FILE("abc[r-z]*.tmp")
   err := FERASE ("abc[r-z]*.tmp")
   IF err = 0
      ? "all files 'abc...tmp'deleted"
   ELSE
      ? "erase fails with return code",err
   ENDIF
ENDIF
```

Example:

```
AEVAL (DIRECTORY ("abc*.tm*"),
       {|file, x| x := FERASE(file[1]), ;
       QOUT ("File", file, if(x==0,"","not"), "deleted")} )
```

Classification:

system, file access

Compatibility:

Available in FS and C5 only. Use the #define identifiers EX_... in the error.fh file to ensure compatibility to the MS-DOS system error codes.

Include:

<FlagShip_dir>/include/error.fh #define's of error codes

Related:

DELETE FILE, ERASE, FERROR()

FERROR ()

Syntax:

`retN = FERROR ()`

Purpose:

Determines if an error occurred during textual/binary file FREAD*(), FWRITE() or FSEEK() operations.

Returns:

<**retN**> is a numeric value, representing the error number. If there was no error, zero is returned. See the file error.fh for error codes available. The most important error code identifiers are:

EX_ENOENT	File not found
EX_ENOTDIR	Path/directory does not exist
EX_EMFILE	Too many open files
EX_EROFS	Read-only file system
EX_EACCES	Access/permission denied
EX_EINVAL	Invalid access code
EX_EIO	I/O error

Note: the codes are system dependent. The above given EX_* values correspond to the standard SVR4 codes for the system "errno" value. Refer to /usr/include/errno.h and/or /usr/include/asm/errno.h for values corresponding to your operating system. You may determine or print the error description in clear text by FERROR2STR()

Description:

FERROR () returns the error code of the previous low-level function. The alternative DOSERROR() invocation returns the same code of the last file access command or function. To get the textual description, use FERROR2STR()

Example: *open file and check for failure*

```
#include "fileio.fh"
#include "error.fh"
handle = FOPEN ("mydata.bin", FO_READ)
IF handle = -1
   err := FERROR()
   ? "Access error " + LTRIM(STR(err))
   DO CASE
   CASE err = EX_EACCES
      ?? " (Permission denied)"
   CASE err = EX_ENOENT
      ?? " (File not available)"
   CASE err = EX_EMFILE
      ?? " (Too many files for process, adapt kernel settings)"
   OTHERWISE
      ?? " (" + FERROR2STR(err) + ")"
   ENDCASE
   QUIT
ENDIF
```

Example: *read/write text file line-by-line and change the text*

```
local fdIn, fdOut, iErr  as intvar
local cIn, cOut, cTxt    as char

cIn   := "infile.txt"                 // input file
cOut  := "outfile.txt"                // output file
fdIn  := fopen(cIn, 0)                // read-only
fdOut := fcreate(cOut, 0)             // truncate if exist
if fdIn <= 0 .or. fdOut <= 0
   ? "could not open " + cIn + " or " + cOut
   wait
   quit
endif
iErr := 0
while iErr == 0 .and. !feof(fdIn)     // until EOF reached
   cTxt := freadtxt(fdIn)             // read one text line w/o LF
   cTxt := strtran(cTxt, "#", "")     // remove all # chars
   fwrite(fdOut, cTxt + chr(13,10) )  // add CR+LF to text line
   if (iErr := ferror()) != 0         // write-error?
      ? "warning: write error", FERROR2STR(iErr), "in " + cOut
   endif
enddo
fclose(fdIn)                          // close input
fclose(fdOut)                         // close output
wait "Done, " + cOut + " created from " + cIn + " - any key..."
```

Classification:

system, file access

Compatibility:

Available in FS and C5 only. Use the #define identifiers EX_... in the error.fh file to ensure compatibility to the MS-DOS system error codes or use the textual description via FERROR2STR().

Include:

<FlagShip_dir>/include/error.fh #define's of error codes

Related:

FERROR2STR(), FEOF(), FCLOSE(), FCREATE(), FOPEN(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

FERROR2STR ()

Syntax:

retC = FERROR2STR ([expN1])

Purpose:

Determines if an error occurred during file operations and if any, returns it textual description.

Arguments:

<**expN1**> is an optional error-code value. If not specified, the last FERROR() or errno value is used per default.

Returns:

<**retC**> is a string, representing the description of last error number or "" if no error occurred.

Description:

FERROR2STR() is similar to DOSERROR2STR(). It is a counterpart of FERROR() and determines the textual string of the error cause. You may store the error code form FERROR() and display the message later using the stored value <expN1> instead of determining the current error code at this time.

Example:

```
#include "fileio.fh"
handle = FOPEN ("mydata.bin", FO_READ)
IF handle < 1
   alert("Access error for mydata.bin;error code =" + ;
         ltrim(FERROR()) + " = " + FERROR2STR()
ENDIF
```

Classification:

programming, system access

Compatibility:

Available in FlagShip VFS6 and later only. MS-DOS and UNIX operating systems may return different error codes and/or descriptions. FlagShip also supports return codes from the RUN, COPY TO and RENAME commands using DOSERROR2STR().

Related:

FERROR(), DOSERROR2STR()

FIELDBLOCK ()

Syntax:

retB = FIELDBLOCK (expC)

Purpose:

Returns a read/write code block for a given field.

Arguments:

<**expC**> is the name of the field to which the set-get block will refer.

The specified field variable need not exist when the code block is created but must exist before the code block is executed.

Returns:

<**retB**> is a code block that, when evaluated, reads (retrieves) or writes (assigns) values of the given field. If the <expC> field does not exist in the current working area, FIELDBLOCK() returns NIL.

Description:

FIELDBLOCK() is a database function. The returned code block can read or write the given database field and has the syntax (refer also to LNG.2.3.3):

```
retB := {|value| IF(value == NIL, expC, FIELD->expC := value) }
```

When the code block is later evaluated without an argument, i.e. EVAL(retB), the current field value is returned. When evaluating it with an additional argument, i.e. EVAL(retB, exp), it assigns (writes) the expression into the database field.

If you need to access another than the current database, use FieldWblock() instead.

Multiuser:

Before the code block is evaluated with an additional argument (write access), at least RLOCK() must be set for SHARED databases. See also LNG.4.8.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
 LOCAL fblck := FIELDBLOCK ("name")     // codeblock for field
"NAME"
 USE address NEW
 ? EVAL (fblck)                         // Miller
 USE custom NEW
 ? EVAL (fblck)                         // Smith & Co
 EVAL (fblck, "New data")
 ? EVAL (fblck)                         // New data
```

Classification: programming, database
Compatibility: Available in FS and C5 only
Related: FIELDWBLOCK(), MEMVARBLOCK(), EVAL()

FIELDDECI ()

Syntax:

`retN = FIELDDECI (expN1|expC1, [expL2])`

Purpose:

Determines the number of decimals in a numeric database field.

Arguments:

<**expN1**>|<**expC1**> is the ordinal field number, or the field name of the database in the current work area.

Options:

<**expL2**> is optional logical value. If .T., the number of decimals in a character field is reported exactly "as is", i.e. not corrected. Otherwise the output corresponds to DbStruct().

Returns:

<**retN**> is the number of decimal places (0..17) of numeric field, or zero on non-numeric fields. Equivalent to the DBS_DEC sub-array element of DBSTRUCT().

Description:

As opposed to DBSTRUCT(), which informs you about the database structure, FIELDDECI() provides you with the specific information about the selected field.

For Numeric fields, FieldLen() reports the total size of digits including sign, deci point (if any) and the deci fraction, and FieldDeci() reports the number of decimal digits in the fraction.

On databases created by CREATE FROM, the character field length may be given as FIELD_DEC *256 + FIELD_LEN. This is considered by FieldLen() and FieldDeci() automatically same as in DbStruct(), where FieldLen() reports the total length and FieldDeci() reports 0, except the <expL2> is .T. If specified so, the field characteristics are reported exact as available in database header, see example 2.

For fields of type Date, Logical, Memo, Variable character and Binary, FieldDeci() always reports 0 and FieldLen() the number of bytes used in the .dbf file, i.e. 1 for Logical, 8 for Date, 10 for Memo and Variable data field, and 2,4,8 for these binary fields.

Example:

```
USE mydbf
? FIELDNAME (2)                 // AMOUNT
? FIELDTYPE (2)                 // N
? FIELDLEN  (2)                 // 14
? FIELDDECI (2)                 // 2

? FIELDTYPE ("Amount")          // N
? FIELDLEN  ("Amount")          // 14
? FIELDDECI ("Amount")          // 2

aDbf := DbStruct()
```

```
? aDbf[2,DBS_NAME]              // AMOUNT
? aDbf[2,DBS_TYPE]              // N
? aDbf[2,DBS_LEN]               // 14
? aDbf[2,DBS_DEC]               // 2
```

Example 2:

```
#include "dbstruct.fh"
USE otherDbf
? FieldName(1)                  // LONGFIELD
? FieldType(1)                  // C
? FieldLen (1)                  // 1025
? FieldDeci(1)                  // 0

? FieldLen (1, .T.)             // 1  (1 + 4*256 = 1025)
? FieldDeci(1, .T.)             // 4

aDbf := DbStruct()
? aDbf[1,DBS_NAME]              // LONGFIELD
? aDbf[1,DBS_TYPE]              // C
? aDbf[1,DBS_LEN]               // 1025
? aDbf[1,DBS_DEC]               // 0
```

Classification:

database

Compatibility:

This function is available in FlagShip only.

Related:

DBSTRUCT(), FIELDNAME(), FIELDLEN(), FIELDTYPE(), oRdd:FieldInfo()

FIELDGET ()

Syntax:

`ret = FIELDGET (expN|expC)`

Purpose:

Retrieves the value of a field using the ordinal position of the field in the database structure.

Arguments:

<**expN**>|<**expC**> is the ordinal position of the field in the record structure for the current working area, or the field name.

Returns:

<**ret**> is the value of the specified field. If <expN> is out of range of FCOUNT(), the return value is NIL.

Description:

FIELDGET() is a database function that retrieves the value of field using its position within the database file structure, similar to ret = &(FIELDNAME(expN)).

Multiuser:

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
LOCAL xname, value
USE address
xname := FIELDNAME (3)
value := &xname
? xname, value                              // NAME  Smith

value := FIELDGET (3)
? value                                     // Smith
```

Classification:

database

Compatibility:

Available in FS and C5 only. Clipper supports <expN> only.

Related:

FIELDPUT(), FIELDNAME(), FCOUNT(), oRdd:FieldGet()

FIELDGETARR ()

Syntax:

`retA = FIELDGETARR ([expN1])`

Purpose:

Creates an array containing values of all fields of the current or specified record.

Options:

<**expN1**> is the required record number. The current database pointer remains unchanged after returning from the function. If not specified or zero, the current record is used.

Returns:

<**retA**> is an one-dimensional array containing field values of all fields in the current or specified record. On memo field, the array element is of character type.

Description:

FIELDGETARR() is a superset of the FIELDGET() function. After changing the array content, you may use FIELDPUTARR() for a global field replacement.

Example:

```
LOCAL aFields AS ARRAY
FIELD Amount
USE mydbf
goto 5
aFields := FieldGetArr ()       // retrieve curr.record

? len(aFields)                  // 8
? Amount                        // 1234.56
? FieldName (2)                 // AMOUNT
? aFields[2]                    // 1234.56

aFields := FieldGetArr (2)      // values of record 2
? recno()                       // 5
? aFields[2]                    // 8765.43
? Amount                        // 1234.56

goto 2
? Amount                        // 8765.43
aFields[2] := 5544.33
aFields[3] := "New Memo text"
aFields[1] := aFields[5] := NIL
while !rlock(); enddo
FieldPutArr (aFields)           // global replacement
Unlock
? Amount                        // 5544.33
```

Classification: d*atabase*

Compatibility: This function is available in FlagShip only

Related: FIELDGET(), FIELDPUTARR(), oRDD:GetArrFields()

FIELDLEN ()

Syntax:

`retN = FIELDLEN (expN1|expC1, [expL2])`

Purpose:

Determines the size of a selected database field.

Arguments:

<**expN1**>|<**expC1**> is the ordinal field number, or the field name of the database in the current work area.

Options:

<**expL2**> is optional logical value. If .T., the field length in a character field is reported exactly "as is", i.e. not corrected. Otherwise the output corresponds to DbStruct()

Returns:

<**retN**> is the length of the database field. Equivalent to the DBS_LEN sub-array element of DBSTRUCT().

Description:

As opposed to DBSTRUCT(), which informs you about the database structure, FIELDLEN() provides you with the specific information about the selected field.

For Numeric fields, FieldLen() reports the total size of digits including sign, deci point (if any) and the deci fraction, and FieldDeci() reports the number of decimal digits in the fraction.

On databases created by CREATE FROM, the character field length may be given as FIELD_DEC *256 + FIELD_LEN. This is considered by FieldLen() and FieldDeci() automatically same as in DbStruct(), where FieldLen() reports the total length and FieldDeci() reports 0, except the <expL2> is .T. If specified so, the field characteristics are reported exact as available in database header, see example 2.

For fields of type Date, Logical, Memo, Variable character and Binary, FieldDeci() always reports 0 and FieldLen() the number of bytes used in the .dbf file, i.e. 1 for Logical, 8 for Date, 10 for Memo and Variable data field, and 2,4,8 for these binary fields.

Example:

```
USE mydbf
? FIELDNAME (2)                 // AMOUNT
? FIELDTYPE (2)                 // N
? FIELDLEN  (2)                 // 14
? FIELDDECI (2)                 // 2

? FIELDTYPE ("Amount")          // N
? FIELDLEN  ("Amount")          // 14
? FIELDDECI ("Amount")          // 2

? FIELDNAME (3)                 // MYMEMO
? FIELDTYPE ("MyMemo")          // M
```

```
? FIELDLEN  ("MyMemo")          // 10

aDbf := DbStruct()
? aDbf[2,DBS_NAME]              // AMOUNT
? aDbf[2,DBS_TYPE]              // N
? aDbf[2,DBS_LEN]               // 14
? aDbf[2,DBS_DEC]               // 2
```

Example 2:

```
#include "dbstruct.fh"
USE otherDbf
? FieldName(1)                  // LONGFIELD
? FieldType(1)                  // C
? FieldLen (1)                  // 1025
? FieldDeci(1)                  // 0

? FieldLen (1, .T.)             // 1  (1 + 4*256 = 1025)
? FieldDeci(1, .T.)             // 4

aDbf := DbStruct()
? aDbf[1,DBS_NAME]              // LONGFIELD
? aDbf[1,DBS_TYPE]              // C
? aDbf[1,DBS_LEN]               // 1025
? aDbf[1,DBS_DEC]               // 0
```

Classification:

database

Compatibility:

This function is available in FlagShip only.

Related:

DBSTRUCT(), FIELDNAME(), FIELDDECI(), FIELDTYPE(), oRdd:FieldInfo()

FIELDNAME ()

Syntax:

retC = FIELD (expN)

Syntax:

retC = FIELDNAME (expN)

Purpose:

Returns the name of the specified field in the current database file. The function name can be abbreviated to FIELD.

Arguments:

<**expN**> is the field number as a position in the database structure. The valid range is 1 to FCOUNT().

Returns:

<**expC**> is a character string returning the name of the specified field. If <expN> is out of the valid range or no database is open in the current working area, null string "" is returned. Otherwise, the field names are returned in uppercase.

Description:

FIELD() or FIELDNAME() can be used in conjunction with FCOUNT () to determine a specific field name and for generic access to database fields. If detailed information on the database structure is required, use the AFIELDS() or DBSTRUCT() functions or COPY STRUCTURE EXTENDED command.

The inverse function of FIELDNAME() is FIELDPOS(). To execute the function in a different working area from the current one, use alias->(FIELD(n)).

Example:

```
USE article
? FCOUNT()                              && 10
? FIELD(1), FIELD(2)                    && IDNUM NAME
? &(FIELD(1)), FIELDGET(1)              && 1234  1234
x = FIELD(2)
? TYPE("x"), LEN(x)                     && C  25
```

Classification:

database

Related:

FIELDPOS(), AFIELDS(), DBSTRUCT(), COPY STRUCTURE EXTENDED, FCOUNT(), FIELDGET(), FIELDPUT(), TYPE(). oRdd:FieldName()

FIELDIS* ()

Syntax:

```
retL = FieldIsEq   (expC1, expN2)
retL = FieldIsEqGt (expC1, expN2)
retL = FieldIsEqLt (expC1, expN2)
retL = FieldIsNEq  (expC1, expN2)
retL = FieldIsGt   (expC1, expN2)
retL = FieldIsLt   (expC1, expN2)
```

Purpose:

Compares numeric field with numeric expression by using Round().

```
FieldIsEq  () compares: Round(Field(expC1),n) == Round(expN2,n)
FieldIsEqGt() compares: Round(Field(expC1),n) >= Round(expN2,n)
FieldIsEqLt() compares: Round(Field(expC1),n) <= Round(expN2,n)
FieldIsNeq () compares: Round(Field(expC1),n) != Round(expN2,n)
FieldIsGt  () compares: Round(Field(expC1),n) >  Round(expN2,n)
FieldIsLt  () compares: Round(Field(expC1),n) <  Round(expN2,n)
```

Arguments:

<**expC1**> is the name of a field in the current or specified working area, e.g. "fieldname" or "alias->fieldname". The value is rounded to FieldDeci() +1 decimals.

<**expN2**> is a numeric value to be compared with numeric field <expC1> The <expN2> value is rounded same as rounding of <expC1>

Returns:

<**retL**> is true (.T.) if the comparison succeeds, and is false (.F.) if the comparison fails, or when the <expC1> field is not numeric.

Description:

Because of the imprecise storage of floating point numbers (see LNG 2.6.4), the simple comparison of field == value may fail; comparing by Round() is here recommended. These FieldIs*() functions do this for your convenience: they rounds both arguments by using the size and deci places of field <expC1>, before comparing them. They simulates the manual comparison

```
deci   := FieldDeci(expC1) +1
value1 := Round(FieldGet(expC1), deci)
value2 := Round(expN2, deci)
return value1 == value2      // for FieldIsEq()
```

Example:

```
DbCreate("test", {{"num","N",10,2},{"int","N",6,0}} )
use test
append blank
for nVar := 0.01 to 9.99 step 1.01
   replace num with nVar, int with int(nVar)
   ? "num field=" + ltrim(num), "var=" + ltrim(nVar) + ;
     ": compare num == var", num == nVar, ;
```

```
     " FieldIsEq('num',var)=", FieldIsEq("num",nVar), ;
     " FieldIsEq('int'," + ltrim(int(nVar)) + ")=", ;
       FieldIsEq("int",int(nVar))
next
wait
```

Classification:

database, programming

Compatibility:

Available in FlagShip VFS7 and newer.

Related:

FieldName(), FieldDeci(), Round(), LNG.2.6.4, LNG.2.9

FIELDPOS ()

Syntax:

`retN = FIELDPOS (expC)`

Purpose:

Returns the position of a field in a database structure.

Arguments:

<**expC**> is the name of a field in the current or specified working area.

Returns:

<**retN**> is the position of the specified field within the database structure of the current working area. If the name <expC> is not found in the database structure or no database is open, zero is returned.

Description:

FIELDPOS() can be used in conjunction with FIELDGET() and FIELDPUT() to determine a specific field and for generic access to database fields. If full information on the database structure is required, use the AFIELDS() or DBSTRUCT() functions or COPY STRUCTURE EXTENDED command.

The reverse function to FIELDPOS() is FIELDNAME(). To execute the function in a directory other than the current one, use alias->(FIELDPOS(c)).

Example:

```
USE custom  NEW ALIAS cust
USE address NEW ALIAS adr
? FIELDPOS("name"), FIELDNAME(2)            // 2  NAME
? FIELDGET(FIELDPOS("name"))                // Smith
? cust->(FIELDGET(FIELDPOS("name")))        // Adams & Co
```

Classification:

database

Compatibility:

Available in FS and C5 only.

Related:

FIELDNAME(), AFIELDS(), DBSTRUCT(), COPY STRUCTURE EXTENDED, FCOUNT(), FIELDGET(), FIELDPUT(), oRdd:FieldPos()

FIELDPUT ()

Syntax:

`ret = FIELDPUT (expN1|expC1, exp2, [expC3])`

Purpose:

Replaces a database field using the position of the field in the database structure.

Arguments:

<**expN1**> is the field number as a position in the database structure. The valid range is 1 to FCOUNT(). <**expC1**> is the name of the field to replace.

<**exp2**> is the value to assign to the given field. The data type of <exp2> must match the data type of the designated database field.

<**expC3**> is the field sub-type, apply for variable fields.

Returns:

If successful <**ret**> is equivalent to the assigned <exp2> value. If the function fails, NIL is returned.

Description:

FIELDPUT() is a database function that replaces the specified field in the current working area. The function is similar to the REPLACE command, but uses the position within the database structure rather than the field name. To execute the function in a working area other than the current one, use alias->(FIELDPUT(n,x)).

Multiuser:

If the database is open in SHARED mode, the RLOCK() (or FLOCK()) locking must be performed first. See also LNG.4.8. Otherwise, AUTOxLOCK() is used, when SET AUTOLOCK is enabled (the default).

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
USE custom  NEW ALIAS cust
USE address NEW
? FIELDPOS ("name"), FIELDNAME(2)           // 2  NAME
? FIELDGET (FIELDPOS("name"))               // Smith
? FIELDPUT (FIELDPOS("name"), "Miller")     // Miller

temp = "name"                               // the "old" style
? &temp                                     // using macros
REPLACE &temp WITH "New name"
? FIELDGET (2)                              // New name
? cust->(FIELDPUT(2, "Other Comp."))        // Other Comp.
```

Classification:

database

Compatibility:

Available in FS and C5 only, whereby C5 does not support <expC1>. The AutoLock feature is available in FlagShip only. The 3rd parameter is new in FS5

Related:

REPLACE, FIELDPUTARR(), FIELDGET(), FIELDNAME(), SET AUTOLOCK, AUTOxLOCK(), oRdd:FieldPut()

FIELDPUTARR ()

Syntax:

retL = FIELDPUTARR (expA1, [expN2])

Purpose:

Stores the array content subsequently into fields of the current or specified database record.

Arguments:

<**expA1**> is the one-dimensional array which content should be subsequently stored in the database fields of the current or specified record. If the array size is larger than the number of fields, only the first FCOUNT array elements are affected. If the array size is smaller, the remaining fields will not be changed. In the array element contains NIL, the corresponding field remains unchanged. Otherwise, the type of the element and the corresponding field has to match, or a run-time error occurs. Numeric and character/memo conversion is done automatically similar to REPLACE.

Options:

<**expN2**> is the required record number. The current database pointer remains unchanged after returning from the function. If not specified or zero, the current record is used.

Returns:

<**retL**> signals the success.

Description:

FIELDPUTARR() is a superset of the FIELDPUT() function. You may e.g. retrieve the record content via FIELDGETARR(), change the array elements, and perform an global record replacement with the FIELDPUTARR() function. To execute the function in a working area other than the current one, use alias->(FIELDPUTARR(a,n)).

Multiuser:

If the database is open in SHARED mode, the RLOCK() (or FLOCK()) locking must be performed first, or SET AUTOLOCK must be active for automatic record lock. Otherwise, the replacement fails with a run-time error.

Example:

see example in FIELDGETARR()

Classification:

database

Compatibility:

This function is available in FlagShip only.

Related:

FIELDGET(), FIELDGETARR(), SET AUTOLOCK, oRDD:PutArrFields()

FIELDTYPE ()

Syntax:

retC = FIELDTYPE (expN1|expC1)

Purpose:

Determines the type of a selected database field.

Arguments:

<**expN1**>|<**expC1**> is the ordinal field number, or the field name of the database in the current work area.

Returns:

<**retC**> is the type of the database field. Equivalent to the DBS_TYPE sub-array element of DBSTRUCT():

C	character field
D	date field
F	floating point field
L	logical field
M	memo field (.dbt)
N	numeric field (integer or fixed decimal point)
V	variable data field (.dbv)

Description:

As opposed to DBSTRUCT(), which informs you about the database structure, FIELDTYPE() provides you with the specific information about the selected field.

Example:

```
USE mydbf
? FIELDNAME (2)                 // AMOUNT
? FIELDTYPE (2)                 // N
? FIELDLEN  (2)                 // 14
? FIELDDECI (2)                 // 2

? FIELDTYPE ("Amount")          // N
? FIELDLEN  ("Amount")          // 14
? FIELDDECI ("Amount")          // 2

? FIELDNAME ("MyMemo")          // MYMEMO
? FIELDTYPE ("MyMemo")          // M
? FIELDLEN  ("MyMemo")          // 10
aDbf := DbStruct()
? aDbf[2,DBS_NAME]              // AMOUNT
? aDbf[2,DBS_TYPE]              // N
? aDbf[2,DBS_LEN]               // 14
? aDbf[2,DBS_DEC]               // 2
```

Classification: database
Compatibility: available in FlagShip only. Field type 'F' and 'V' is not available in C5
Related: DBSTRUCT(), FIELDNAME(), FIELDLEN(), FIELDDECI(), oRdd:FiledInfo()

FIELDWBLOCK ()

Syntax:

`retB = FIELDWBLOCK (expC1, expN2)`

Purpose:

Returns a read/write code block for a given field in the specified working area.

Arguments:

<**expC1**> is the name of the field to which the set-get block will refer. The specified field variable need not exist when the code block is created but must exist before the code block is executed.

<**expN2**> is the number of the working area number where the field resides.

Returns:

<**retB**> is a code block that, when evaluated, reads (retrieves) or writes (assigns) values of the given field. If the <expC1> field does not exist in the working area <expN2>, FIELDWBLOCK() returns NIL.

Description:

FIELDWBLOCK() is a database function, similar to FIELDBLOCK(). The returned code block can read or write the given database field and has the general syntax (refer also to LNG.2.3.3):

```
  retB := {|par| IF(par==NIL, (expN)->expC, (expN)->expC := par)}
```

When the code block is later evaluated without an argument, i.e. EVAL(retB), the current field value is returned. When evaluating it with an additional argument, i.e. EVAL(retB, exp), it assigns (writes) the expression into the database field.

Multiuser:

Before the code block is evaluated with an additional argument (write access), at least RLOCK() must be set for SHARED databases. See also LNG.4.8.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
 LOCAL fblck := FIELDWBLOCK("name",3) // codeblock for "NAME" in WA
3
 SELECT 3
 USE address
 SELECT 5
 USE other
 ? EVAL (fblck)                       // Miller (in address.dbf)
 EVAL (fblck, "Smith")                // replace (3)->Name with
Smith
 ? EVAL (fblck)                       // Smith (in address.dbf)
```

Classification: programming, database
Compatibility: Available in FS and C5 only
Related: FIELDBLOCK(), MEMVARBLOCK(), EVAL()

FILE ()

Syntax:

`retL = FILE (expC1, [expL2])`

Purpose:

Determines whether a file exists in the defined path.

Arguments:

<**expC1**> is the name of the file or directory, including extension, optionally preceded by a path. Standard Unix wildcards using ?, *, and [..] to form the file name part are supported. In MS-Windows, * and ? are supported. The FS_SET() file and path translation to lower or upper case are considered, same as conversion of index extension and the substitution of DOS drive letter, see FS_SET("lower" or "upper" or "path*" or "trans") or #include "fspreset.fh", and x_FSDRIVE environment variable(s) in section FSC.3.3.

Options:

<**expL2**> is a logical value specifying whether the FS_SET("lower"| "upper" | "pathlower" | "pathupper") setting and hence the file and path conversion should be considered (.T.) or ignored (.F.). The default is .T. = the setting and auto translation is considered.

Returns:

<**retL**> is a logical value returning TRUE, if the file exists, or FALSE if a file matching the pattern <expC1> is not found.

Description:

FILE() determines whether any file matching a file specification pattern is found. If a path is not specified, FILE () first searches the current directory or the SET DEFAULT path when given. If the file is not found, the SET PATH is searched. When <expL2> is .F., only the current or given directory in <expC1> is considered as given.

The current user must have at least the "x" access right for the specified directory. In MS-Windows, the directory may not be hidden by "S" or "H" attribute. If the function fails, FILE() returns .F. and FERROR() returns the last system error code.

Example:

```
#ifdef FlagShip
   FS_SET ("lower", .T.)
   FS_SET ("pathlower", .T.)
#endif
? FILE("my.DBF")                          && .F.
? FILE("\usr\users/Joe/My.dbf")           && .T.
SET PATH TO /usr/users/joe                && or: \USR\users\Joe
? FILE("my.DBF")                          && .T.
```

Classification:

system, file access

Compatibility:

UNIX file names and paths are case sensitive, but FlagShip can convert the given pattern to lower or upper case automatically. Refer to the section LNG.9.5. The support of wildcards is new in FS4. Both forward "/" and backward "\" slashes are supported for path delimiters in Unix and MS-Windows environment.

Related:

SET DEFAULT, SET PATH, FS_SET(), #include "fspreset.fh"

FILESELECT ()

Syntax:

```
retCA = FileSelect ([expC1], [expCA2], [expC3],
           [expL4], [expL5])
```

or:

```
retCA = FileSelectDialog ([expC1],[expCA2],[expC3],
           [expL4],[expL5])
```

Purpose:

Opens file dialog and returns the selected or entered file(s). Available in GUI mode only. For Terminal i/o mode, see example 2 below.

Options:

<**expC1>** (optional) is the name of a directory, the dialog starts off in that directory. If not given or is empty, the dialog starts in current (or last selected) directory. Current directory can also be specified by "./" or CurDir().

<**expCA2**> (optional) is a filter specifying displayed files. You can specify a string with explanation and one or several wildcards separated by space) in parentheses eg. "Anything (*.*)" or "FlagShip sources (*.prg *.c *.fh *.ch)". When specifying different search criteria in an array, e.g. {"All files (*.*)", "Databases (*.dbf)", "Source files (*.prg *.fh)", "C sources (*.c *.cpp *.h)"}, the required search filter is then selected by user in the dialog combobox. If <expCA2> is not specified or is empty, "All files (*.*)" is the default. The search criteria is not case sensitive.

<**expC3**> (optional) is a string specifying caption of the dialog box, default is "Select File" or "Select File(s)" depending on <expL4>. If <expL5> is .T., the default is "Save file as (specify)".

<**expL4**> (optional) is a logical value, where .T. enables multi- selection which returns an array <retA>. Default is .F. returning <retC>. To select more than one file, use Ctrl- or Shift-key with left mouse button.

<**expL5**> (optional) is a logical value, where .T. enables entering the file name by user, which is then returned in <retC>. The returned string can be used to save a file according to users decision, see example 3 below. The <expL4> setting is not considered if <expL5> is .T. Default for <expL5> is .F. The by user entered file name should preferably contain only simple ASCII character set, since umlauts and UTF special characters may be interpreted differently.

Returns:

<**retC**> is a string representing the selected (or entered) file name including the path, or empty string if the user canceled the dialog.

<**retA**> with <expL4> = .T. is an array of strings representing all the selected file names including paths, or empty array if the user canceled the dialog.

Description:

FileSelect[Dialog]() opens a modal file dialog and returns the name (or file names) of the selected or entered file(s). The dialog is centered on the application window. The dialog starts in the directory <expC1> and displays files meeting search criteria <expCA2>. The selected (or entered) file(s) are returned as a string (or as an array of strings). The selected (or entered) file is however not opened by FileSelectDialog(), but the application should handle it using the return value, see examples below. This function is available in GUI mode only.

Example 1:

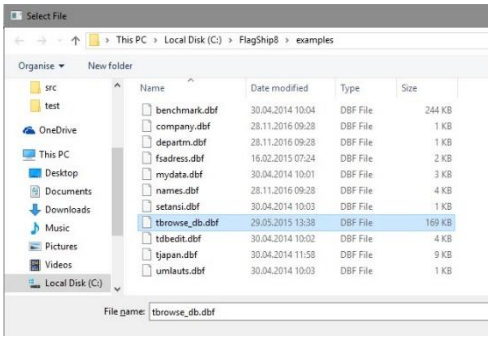
```
// Display the by user selected text file

if AppIoMode() == "G"              // GUI mode?
   cFile := FileSelectDialog("C:\mytext", "Text files (*.txt)" )
else
   cFile := TermFileSelect ("C:\mytext", "*.txt")  // see example 2
endif
if !empty(cFile)
   // RUN ("notepad " + cFile)     // execute external app
   cText := MemoRead(cFile)        // or process internally
   MemoEdit(cText,,,,,.F.)
endif
```

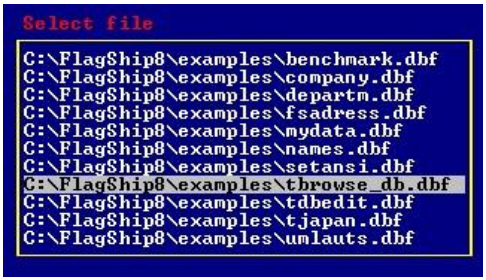
Example 2:

Select desired files in GUI or Terminal i/o mode:

```
set font "Courier",12
oApplic:Resize(25,80,,.T.)
SET COLOR TO "W+/B,N/W,GR+/B,GR+/B,W+/G,R+/B" // Terminal i/o
cls
local cFile

if AppIoMode() == "G"                          // GUI mode?
  cFile := FileSelect(curdir(), "Databases (*.dbf *.data)" )
else
  cFile := TermFileSelect(curdir(), "*.dbf")  // Terminal i/o
endif
? "Selected file:", cFile
?
Wait

/*************************************************************
 * TermFileSelect() emulates FileSelect() in Terminal i/o mode
 */
FUNCTION TermFileSelect(cPath, cFiles)
local aDir, aFiles, ii, iLen := 0, sScr, c1, c2, r2, cRet := ""
if !empty(cPath)
   cPath := alltrim(cPath)
   if !(right(cPath,1) $ "/\")
     cPath += PATH_SLASH    // slash (Linux) or backslash (Windows)
     cFiles := cPath + cFiles
   endif
endif
aDir := Directory(cFiles)               // determine desired files
if empty(aDir)
```

```
  return cRet
endif
aFiles := {}
for ii := 1 to len(aDir)                    // prepare for Achoice()
   aadd(aFiles, cPath + aDir[ii,1])
   iLen := max(iLen, len(aFiles[ii]))
next
asort(aFiles)
ii := max(0, maxcol() - iLen)               // prepare the window
c1 := max(0, int(ii / 2) -1)
c2 := min(c1 + iLen +2, maxcol())
r2 := min(len(aFiles) +4, maxrow() -2)
sScr := SaveScreen(2,c1,r2,c2)
@ 3,c1,r2,c2 BOX B_SINGLE COLOR "GR+/B"
@ 2,c1 clear to 2,c2
@ 2,c1+1 say "Select file" COLOR "R+/B"
ii := achoice(4,c1+1,r2-1,c2-1,aFiles)      // let user select file
if ii > 0
  cRet := aFiles[ii]
endif
RestScreen(2,c1,r2,c2,sScr)
return cRet
```





Example 3:

Displays the by user selected database in separate sub-window when clicking the corresponding push button. Another example is in <FlagShip_dir>/examples/ dbfstru.prg

```
set font "Courier",12
oApplic:Resize(25,80,,.T.)

local oButton1 := PushButton(20,50, "Inspect database", , ;
                             {|| funButton1() } )
oButton1:Show()
        // ...other statements...
wait    // or other wait state e.g. @..GET / READ etc.

FUNCTION funButton1()
local nWin, nSelect
local cFile := FileSelect(curdir(), "Databases (*.dbf *.data)" )
if !empty(cFile)
   nSelect := select()          // save current work area
```

```
   nWin := Wopen(3,2, 23,76)
   USE (cFile) NEW SHARED
   if used()
      DbEdit()                  // display selected database
      USE
   endif
   Wclose(nWin)
   SELECT (nSelect)             // restore current database
endif
return
```

Example 4:

Edit text and save it to user specified text file, considering default extension and (not-)overwriting of existing file

```
set font "Courier",12
oApplic:Resize(25,80,,.T.)

local nWin, nChoice, nLast, iPos, nRow := 23
local cFile, cExt, cOldText := "hello", cNewText := ""

nWin := Wopen(3,2, 24,75)       // optional in subwindow
cNewText := MemoEdit(cOldText)  // edit the text, exit by ^W or ESC
nLast    := lastkey()
Wclose(nWin)                    // close subwindow

if nLast == K_ESC .or. cNewText == cOldText
   @ nRow,0 say "Text unchanged, not saved"
else
   while .T.
      cFile := FileSelectDialog("./", "Text files (*.txt)", ;
                                "Enter file name to save", , .T.)

      if empty(cFile)           // user selected "Cancel"
         if alert("Loose text changes?", {"No", "Yes"} ) == 2
            @ nRow,0 say "Save aborted, text not saved"
            exit                // do not save, exit loop
         endif
         loop                   // repeat dialog
      endif
      iPos := rat(".", cFile)   // check extension
      if iPos == 0
         cFile += ".txt"        // add expected extension
      else
         cExt := substr(cFile, iPos)
         if !(lower(cExt) == ".txt")      // expected extension?
            cFile := left(cFile, iPos -1) + ".txt"  // no,change
         endif
      endif
      if file(cFile)            // file already exist?
         nChoice := alert(cFile + ";File exist, overwrite?", ;
                          {"No", "Yes"} )
         if nChoice != 2        // if overwrite is not desired,
            loop                // repeat loop to enter file name
         endif
      endif
```

```
      MemoWrit(cFile, cNewText) // save file
      @ nRow,0 say "Text saved to file " + cFile
      exit                      // exit loop
   enddo
endif
wait
```

Classification:

system, file access

Compatibility:

Available in FlagShip VFS8 and newer.

Related:

FILE(), DIRECTORY()

FINDEXEFILE()

Syntax:

`retC = FindExeFile (expC)`

Purpose:

Returns the complete path of an executable.

Arguments:

<**expC**> is the name of executable to be located. It is either the name self (like "myExe" or "MYEXE.EXE"), or the exe name with relative path, e.g. "../../hallo/myExe". You may use ExecName() to determine full path of your executable - but only if the directory of your executable is available in the PATH environment or invoked with a relative/absolute path.

Returns:

<**retC**> is absolute path with the executable or ""

Description:

This function searches the PATH environment variable for given executable <expC>, similar to the shell "which" function. If found, returns the absolute path if found and the file has execute permission. Relative paths are accepted and converted to absolute. If not found, an empty string "" is returned.

Example:

```
? getenv("PATH")                 // /usr/bin:/bin:/usr/local:/data
? FindExeFile("a.out")           // /usr/local/a.out
? FindExeFile("FlagShip")        // /usr/bin/FlagShip
? FindExeFile("../mayer/a.out")  // /home/mayer/a.out
? FindExeFile(ExecName())        // C:\data\MyApplic.exe or ""
```

Classification:

programming

Compatibility:

Available in FlagShip VFS5 and newer.

Related:

File(), TruePath()

FKLABEL ()

Syntax:

`retC = FKLABEL (expN)`

Purpose:

Returns the name of the specified function key.

Arguments:

<**expN**> is the numeric value of the FN key in the range 1 to 40.

Returns:

<**retC**> is a character string in the format Fn or Fnn representing the name of the function key specified by the argument <expN>. If the argument <expN> is out of range, the function returns null string "".

Description:

FKLABEL() is a compatibility function to dBASE and is equivalent to retC = "F" + LTRIM(STR(expN)).

Example:

```
? FKLABEL(1)                                    // "F1"
? FKLABEL(20)                                   // "F20"
? FKLABEL(99)                                   // ""
```

Classification:

programming

Compatibility:

FKLABEL() in FS4 and C5 does not check the availability of the function key specified. In FlagShip, up to 48 function keys may be defined in the FStinfo.src (or other similar terminfo description).

Related:

SET FUNCTION, SET KEY

FKMAX ()

Syntax:

`retN = FKMAX ( )`

Purpose:

Returns number of function keys as a constant.

Returns:

<**retN**> is constant 40.

Description:

FKMAX() is available only for compatibility purposes to dBASE.

Example:

```
? FKMAX()                                   // 40
```

Classification:

programming

Compatibility:

FKMAX() in FS4 and C5 does not check the availability of the function key specified. In FlagShip, up to 48 function keys may be defined in the FStinfo.src (or other similar terminfo description).

Related:

SET FUNCTION, SET KEY

FLAGSHIP_DIR ()

Syntax:

retC = FlagShip_Dir ()

Purpose:

Returns the path of FlagShip installation directory.

Returns:

<**retC**> is the path where the FlagShip development package was installed, e.g. "/usr/local/FlagShip8" in Unix/Linux or "C:\ FlagShip8" in MS-Windows. Assigned only if the installation directory was found during the compilation. If not so, empty string "" is returned otherwise.

Description:

The FlagShip installation directory is determined during the compilation phase and is also displayed by using the -v switch.

Example:

```
? FlagShip_Dir()                        // /usr/local/FlagShip8
```

Example:

```
// set polish keyboard mapping in GUI
m->oApplic:Font:CharSet(FONT_ISO8859_2)
FS_SET("guikey", FlagShip_dir() + PATH_SLASH + "terminfo" +;
                 PATH_SLASH + "FSguikey.latin2_pl")
```

Classification:

programming

Compatibility:

Available in FlagShip VFS6 and newer.

Related:

-v compiler switch

FLOCK ()

Syntax:

```
retL = FLOCK ()
```

Purpose:

Locks the current database selected to allow global write access in multiuser mode.

Returns:

<**retL**> is a logical value, which signals success or error. If TRUE is returned, the current database is successfully locked; otherwise the file cannot be locked. FLOCK() will return FALSE when another user:

- has successfully locked the database using FLOCK() or RLOCK().
- has APPENDed new records without an additional UNLOCK being performed.

Description:

After a successful FLOCK(), any record may be changed using REPLACE, DELETE, RECALL or @..GET../READ, including multi-record changes (e.g. with ALL, NEXT, FOR or WHILE clause) or using the DBEVAL() function. If FLOCK() is successful, the file lock remains in place until UNLOCK, CLOSE, or DBUNLOCK() is executed.

FLOCK() may also be used to avoid database changes by other user executing database wide commands like TOTAL, COUNT or INDEX ON.

For each invocation of FLOCK(), there is one attempt to lock the database, and the result is returned as a logical value. If a record of the database is locked from the current application by RLOCK() or APPEND BLANK, the previous record is UNLOCKed before attempting to lock the whole file. If the user has locked the database with FLOCK(), the file lock remains active and FLOCK() return TRUE. An attempt to APPEND BLANK or RLOCK() a database already FLOCKed will succeed, but leaves FLOCK unchanged.

FLOCK() is effective only on the database in the working area selected (and the associated .dbt files) only, not on related databases. To execute the function in a different working area from the current one, use alias->(FLOCK()).

To determine if the record or file is locked, without actually activating locking, use ISDBRLOCK() or ISDBFLOCK() functions.

Multiuser:

In multiuser mode (SET EXCLUSIVE OFF or USE... ...SHARED), the record or file must be locked to allow a **write** access to the record or the whole file. If not set by the programmer and SET AUTOLOCK is ON, FlagShip locks the record or file automatically by using the AUTOxLOCK() function.

Locking is not necessary for a read access such as var=field, SKIP, SEEK, GOTO, LOCATE etc. Other users may also read the locked file or record, but they cannot write it. Refer to section LNG.4.8 for more information.

FLOCK() provides a shared lock, allowing other users to read the locked databases, but they cannot write in them.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
SET EXCLUSIVE OFF                       && multiuser mode
USE address INDEX name                  && you should check it
APPEND BLANK                            && automatic RLOCK
replace NAME with "Smith"               && on write access to one
UNLOCK                                  && record only, use RLOCK

GOTO TOP
DO WHILE .not. FLOCK()                  && wait until FLOCK
   INKEY (0.5)                          && come trough,
ENDDO                                   && with small delay
DELETE ALL FOR EMPTY(name)
UNLOCK
```

Classification:

database

Compatibility:

FlagShip uses Unix and Windows compatible locking methods (via fcntl). The usage of RLOCK(), FLOCK() and UNLOCK is nevertheless source- and logic-compatible to other Xbase dialects. FlagShip, Clipper, dBase, and Foxbase/FoxPro locking is incompatible to each other. The check using ISDBRLOCK() and the **AutoLock** feature is available in FS only.

Refer to section SYS (tunable UNIX parameters) to set or increase the number of maximal available locks.

Related:

USE...EXCLUSIVE|SHARED, SET EXCLUSIVE, UNLOCK, RLOCK(), ISDBRLOCK(), ISDBFLOCK(), DBFLOCK(), DBRLOCK()

FLOCKF ()

Syntax:

```
retN = FLOCKF (expN1, [expN2], [expN3])
```

Purpose:

Locks or unlocks a part or the whole of a binary file, which has already been opened, for a read and/or write access.

Arguments:

<**expN1**> is the file handle number previously returned by FOPEN() or FCREATE().

Options:

<**expN2**> is the required locking mode. If not specified, zero is assumed.

value	#define	Description
0	FF_UNLOCK	Unlock the whole file
1	FF_READ	Lock to read only by curr. user
2	FF_READWRITE	Lock to read/write by curr. user

<**expN3**> is a positive byte count to be locked from the current position. If omitted, or zero is specified, the rest of the file is locked until end-of-file.

Returns:

<**retN**> is set to zero, if the required locking is successful; non- zero otherwise.

Description:

FLOCKF() locks for low-level system calls Fxxx(). It has some similarity to high-level database locking using RLOCK() or FLOCK(). Unlike these, FLOCKF() does not protect the write access by another user, but merely signals that the other user may or may not **lock** and therefore may access the same file.

It is the programmer's responsibility, to perform and check the required locking using FLOCKF() before the read or write access. As opposed to the high-level database write access using REPLACE, no error messages from FlagShip are generated by low-level access when FLOCKF() fails.

FLOCKF() can be used to control multiuser access to text or binary files. Locking a file does not protect the file from being read or written by any other user, but another user cannot lock the same portion of the file.

Unlike the database FLOCK() function, FLOCKF() allows to either lock the file, or a part of it, for reading or both reading and writing. A file locked for reading cannot be locked for writing by another user, but all other users may lock the file for reading, too. A file locked for reading and writing cannot be locked by another user, neither for reading nor writing.

Locking and unlocking with FLOCKF() should always be done at the same byte offset, preferably zero, representing begin-of-file. After executing FLOCKF(), the file pointer

remains unchanged. Note that unlike FLOCKF(), FLOCKF() allows to lock different parts of a file at the same time.

When the binary file is closed using FCLOSE(), all FLOCKF() locks set by the current user will be cleared.

Warning: FLOCKF() allows a low-level access to the file system, similar to the C function fcntl(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Multiuser:

On simultaneously used binary files, issue FLOCKF() and check the return code at least before a write attempt.

Example:

```
#include "fileio.fh"
#define LINEFEED CHR(10)
LOCAL handle, ii := 0
handle := FOPEN ("myfile.txt", FO_READWRITE)
IF handle > 0
   WHILE FLOCKF (handle, FF_READWRITE) # 0 .AND. ii <= 10
      ? "Waiting to access myfile.txt"
      INKEY(1)
      ii++
   END
   IF ii > 10
      ? "Sorry, cannot lock. Try later."
      FCLOSE (handle)
      RETURN
   ENDIF
   FWRITE (handle, "Line 1" + LINEFEED)
   FWRITE (handle, "Line 2" + LINEFEED)
   FSEEK  (handle, 0)              // go back to file begin
   FLOCKF (handle, FF_UNLOCK)
   ? "Try now to read the file by other user or press..."
   INKEY(0)
   FCLOSE (handle)
ENDIF
```

Classification:

system, file access

Compatibility:

Available in FS only. Refer to section SYS: tunable UNIX parameters to set or increase the number of maximal locks are available.

Include:

<FlagShip_dir>/include/fileio.fh #define's for <expN2> <FlagShip_dir>/include/error.fh #define's of error codes

Related:

FCLOSE(), FERROR(), FOPEN(), FCREATE(), FREAD(), FREADSTR(), FSEEK(), FWRITE(), UNIX: fcntl

FONTDIALOG ()

Syntax:

`retL = FONTDIALOG (expO1)`

Purpose:

Lets the user modify attributes of the font object by a modal dialog. Supported in GUI mode only, other i/o modes do nothing and returns .F.

Arguments:

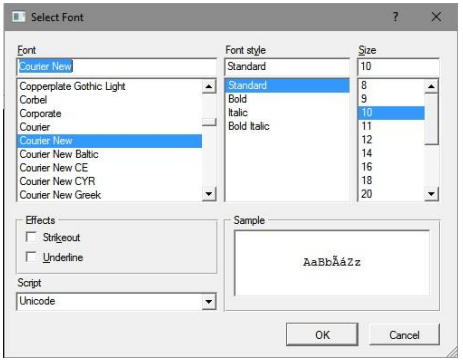
<**expO1**> is a font object

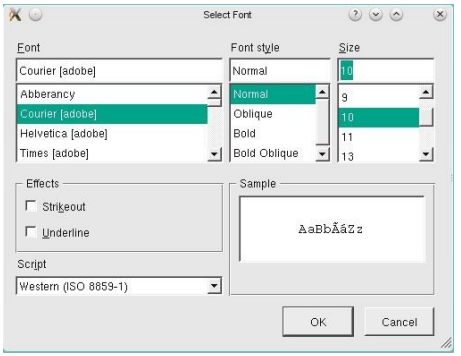
Returns:

<**retL**> is T. and the modified font object <**expO1**> if the user clicked "OK" button. Otherwise .F. and the original unmodified object is returned if the user clicked "Cancel" or closed the dialog window.

Description:

The **FontDialog()** function is a shortcut for, and equivalent to oFont:Dialog(), see section OBJ.Font for further details.





Example 1:

```
oFont1 := FontNew("Courier", 10, "B")
ok := FontDialog(oFont1)              // == oFont1:Dialog()
```

Example 2:

Set/Modify the standard input/output font

```
ok := FontDialog(oApplic:Font)        // == oApplic:Font:Dialog()
```

Compatibility:

Available in FS8 (and newer)

Related:

OBJ:Font class, oFont:Dialog()

FONTNEW ()

Syntax:

retO = FONTNEW ([expC1], [expN2], [expC3])

Purpose:

Instantiates the Font object of a Font class. Supported in GUI mode only, other modes returns standard font.

Options:

<**expC1**> is the desired font family name, see details in oFont:FontFamily. The given family name is case insensitive. If <expC1> is not given, is NIL or empty, the default desktop font is used.

<**expN2**> is the desired font size in Points. You may specify or change the font size later by oFont:SizePoint( ) or oFont:SizePixel( ). If <expN2> is not given, or is NIL or 0 (zero), the default desktop size is used.

<**expC3**> is optional font attributes, any combination of letters "N" = normal, "B" = bold, "I" = italic, "U" = underline, "S" = striked thru. You may check or change the font attribute later by oFont:Attrib() or oFont:AttribChar()

Returns:

<**retO**> is T. and the modified font object <**expO1**> if the user clicked "OK" button. Otherwise .F. and the original unmodified object is returned if the user clicked "Cancel" or closed the dialog window.

Description:

The **FontNew ()** function creates new font object and is equivalent to `oFont := Font{expC1,expC2,expC3}`. You may set other font attributes thereafter, see section OBJ.Font for further details.

Example:

```
oFont1 := FontNew("Courier", 10, "B")
? "Hello World" FONT (oFont1)

oFont2 := FontNew()
oFont2:Name := "Arial"          // Font = Arial (or Helvetica)
oFont2:Size := 12               // 12pt
oFont2:AttribChar("BI")         // bold italic
SetFont(oFont2)                 // use as default font
? "oFont2=", oFont2:FontFamily, oFont2:Size, oFont2:AttribChar()
```

Compatibility:

Available in FS7 (and newer)

Related:

OBJ:Font class, FontDialog()

FOPEN ()

Syntax:

`retN = FOPEN (expC1, [expN2], [expNL3] )`

Purpose:

Opens the specified text or binary file.

Arguments:

<**expC1**> is the file name. An optional path and extension can be specified. The FS_SET() file and path translation is considered. The SET DEFAULT or SET PATH is not evaluated by the low-level function.

Option:

<**expN2**> is the open mode of the file. The open mode can be:

value	#define in fileio.fh	Description
0	FO_READ	Open for read-only access
1	FO_WRITE	Open for write-only access
2	FO_READWRITE	Open for read/write access

If <expN2> is not specified or out of range, read-only access is assumed.

<**expNL3**> is optional logical or numeric value specifying creation of new file, if <expC1> does not exist

value	#define in fileio.fh	Description
0 or .F.	FO_NOCREATE	Do not create new file
1	FO_CREATE	Creates new file if not exist
2 or .T.	FO_CREATEWRITE	Creates new file if not exist but not in read-only mode

If <expNL3> is not specified, the global setting

```
_aGlobSetting[GSET_N_FOPEN_CREATE]  // default = 0 = FO_NOCREATE
```

is used. For backward compatibility to older FlagShip versions, set

```
_aGlobSetting[GSET_N_FOPEN_CREATE] := 2
```

Returns:

<**retN**> is the file handle of the opened file, in the range from zero to 65535. If an error occurs, -1 is returned. The file is left opened in the requested mode until FCLOSE() is executed.

Description:

FOPEN () opens the specified file in no-delay-mode and returns the opened file handle. This should be assigned to a variable, since it is needed to identify the file to the other file functions. On success, the additional read and/or write locking using FLOCKF() is available.

If FOPEN() fails, -1 is returned, and FERROR() can then be used to determine the reason for the failure.

Warning: FOPEN() allows a low-level access to the file system, similar to the C function open(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Example 1:

```
#include "fileio.fh"
#include "error.fh"
art_handle = Fopen("article.doc", FO_READWRITE, .T.)
IF art_handle = -1
   ? "The file cannot be created!"
   DO CASE
   CASE FERROR() = EX_EACCES
      ?? " (Permission denied)"
   CASE FERROR() = EX_EROFS
      ?? " (Read-only file system)"
   CASE FERROR() = EX_EMFILE
      ?? " (Too many files for process, adapt the kernel)"
   ENDCASE
   QUIT
ENDIF
```

Example 2:

Opens or create new file in mixed case (Linux/Unix) when the automatic conversion to lowercase is on (e.g. in fspreset.fh)

```
#include "fileio.fh"
#include "fspreset.fh"                    /* note */
local fd, lConvPath, lConvFile, cFile

cFile := "/home/myName/Test/MyFile.txt"

/* this will fail, when /home/myname/test/myfile.txt
 * does not exist, because of lower case translation of
 * <cFile> (for path and file) in #include "fspreset.fh"
 */
fd := fopen(cFile, FO_READ, FO_NOCREATE) // may fail
? "fd=",fd                               // -1

/* this will fail only when /home/myName/Test/MyFile.txt
 * does not exist (and could not be created)
 */
lConvPath := FS_SET("pathlower", .F.)    // save status
lConvFile := FS_SET("lowerfile", .F.)    // save status
fd := fopen(cFile, FO_READWRITE, FO_NOCREATE)
if fd < 0
   ? "File", cFile, "does not exist"
   fd := fopen(cFile, FO_READWRITE, FO_CREATE)
   if fd > 0
      ?? " but was created"
   else
      ?? " and could not be created"
   endif
endif
FS_SET("pathlower", lConvPath)           // restore status
FS_SET("lowerfile", lConvFile)           // restore status
```

Example 3:

Check the user permission prior open database or file

```
found := file(mysubdir + "myfile.dbf")
if !CHECK_ACCESS (mysubdir, "myfile.dbf", !found)
   ? "access denied to " + mysubdir + "myfile.dbf"
   quit
else
   USE (mysubdir + "myfile")     // or MYUSE() from CMD.USE
endif

function CHECK_ACCESS (dir, file, create)
*--------------------------------------
* check the access rights to directory or file
* [dir   ] = the specified directory, or "./" as default
* [file  ] = any file name. If not specified,
*            temp is crea/deleted.
* [create] = check r/w access if .T., otherwise r only

LOCAL handle
if empty(file)
   file := FS_SET("print")
   file := "$$temp$$" + substr(file,at(".",file))
   create := .T.
endif
create := if (valtype(create) == "L", create, .F.)
dir    := if (empty(dir), ".", trim(dir))
if !(SLASH == substr(dir, -1))
   dir = trim(dir) + SLASH
endif

if !file(dir + file)
   handle := fcreate(dir + file)
else
   handle := fopen(dir + file, if (create, 2, 1))
endif
if handle < 1 .or. ferror() > 0
   return .F.
endif
fclose (handle)
if create
   ferase(dir + file)
endif
return ferror() == 0
```

Classification:

system, file access

Compatibility:

Note the UNIX access rights, not available in DOS. See also section SYS for tunable UNIX parameters to set the number of simultaneously open files. The third parameter is available in VFS6 (and newer) only. Clipper and VFS6 behaves like <expNL3> = 0, FS4 and VFS5 like <expNL3> = 2

Include:

<FlagShip_dir>/include/fileio.fh #define's for <expN2>
<FlagShip_dir>/include/error.fh #define's of error codes

Related:

FCLOSE(), FERROR(), FCREATE(), FREAD(), FREADSTR(), FSEEK(), FWRITE(), FLOCKF()

FOUND ()

Syntax:

`retL = FOUND ()`

Purpose:

Determines the success of a previous SEEK, FIND, LOCATE or CONTINUE.

Returns:

<**retL**> is TRUE if the last search operation was successful.

Description:

FOUND () can be used when the success of a search operation is important for determining what to do next.

The value of FOUND() is retained until another record movement command (except SKIP 0) is executed or the index order is changed. When performing another search, FOUND() reflects the last status. On record movement, or when changing the current index, FOUND() is set to FALSE. The init status is FALSE, and remain so until a SEEK, FIND or LOCATE is executed in this working area.

Each working area (or the RDD object) has a FOUND() flag, so if you search in a SET RELATION's child area, the results of the search can be queried in the child area. To execute the function in a working area other than the current one, use alias->(FOUND()).

Tuning:

You may achieve Clipper 5.x compatibility by assigning
`_aGlobSetting[GSET_L_FOUND_SETORDER] := .T.   // default is .F.`
to let the Found() status unchanged during SET ORDER TO <n>

Example:

```
USE personal INDEX name
LOCATE FOR name = space(5)
? FOUND(), EOF()                        && .F. .T.
LOCATE FOR TRIM(name) = "Smith"
? FOUND(), EOF(), RECNO()               && .T. .F.  25
SKIP
? FOUND(), EOF(), RECNO()               && .F. .F.  26
CONTINUE
? FOUND(), EOF(), RECNO()               && .T. .F.  39

SEEK "SMITH"
? FOUND(), EOF(), RECNO()               && .T. .F.  25
```

Classification:

database

Related:

CONTINUE, FIND, LOCATE, SEEK, SET RELATION, SET SOFTSEEK, EOF(), oRdd:Found

FREAD ()

Syntax:

retN = FREAD (expN1, @varC2, expN3, [expL4])

Purpose:

Reads any count of bytes from the specified file into a character buffer.

Arguments:

<**expN1**> is the file handle previously returned by FOPEN() or FCREATE().

<**varC2**> is an already existing and initialized memory variable of character type used as a buffer. Its length must be at least <expN3>. This variable must be passed by reference to FREAD().

<**expN3**> is the number of characters to be read into the buffer.

<**expL4**> is optional logical value. If true (.T.), FREAD() will retry the read on failure. The default is .F. = no retry.

Returns:

<**retN**> represents the number of bytes that were read successfully. This value should be equal to <expN3>. If the return value is smaller than <expN3> or equals zero, it means that the end of file has been reached or an error has occurred, FERROR() is set. Exception: when the passed <expN3> is <= 0 or the length of <expC2> is 0, zero is returned and FERROR() is set to zero. When <expN1> is not numeric or is < 0, or when <expC2> is not a string, or if the length of <expC2> is less than <expN3>, -1 is returned and FERROR() remain unchanged.

Description:

FREAD () is a low-level file function that reads characters from any binary file or device (in no-delay-mode) into an existing character buffer. It starts reading at the current file pointer position, and reads the specified number of characters, or less, if the end of file is encountered. FREAD() reads all characters, including CR, LF and binary zero, regardless of the setting FS_SET("zerobyte").

For positioning the file pointer without reading, FSEEK () should be used.

Warning: FREAD() allows a low-level access to the file system, similar to the C function read(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Example:

```
#define BUFFSIZE 512
LOCAL buff, handle, text
buff = SPACE(BUFFSIZE)                  // the best performance
text = ""
handle = FOPEN("help.doc")
IF FERROR() == 0
   DO WHILE FREAD(handle, @buff, BUFFSIZE) = BUFFSIZE
      text = text + buff
   ENDDO
```

```
   FCLOSE (handle)
ENDIF
```

Example:

Uses FREAD() to read data from serial port:

```
LOCAL handle, out := "", device := "/dev/tty02"
if (handle := FOPEN(device, 2)) < 1
   ? "cannot open", device, "because", ferror()
   quit
endif
* RUN ("stty 19600 -echo < " + device) // etc, on request
while inkey() != K_ESC
   ?? out := ReadWithDelay (handle, 80, 10.5)
   if len(out) == 0                   // there was no input
      exit                            // within 10.5 sec
   endif
enddo
FCLOSE (handle)
quit

FUNCTION ReadWithDelay (handle, nSize, maxDelay)
*********************************************
LOCAL start := secondscpu(), out := "", buff := " "
while .T.
   while FREAD(handle, @buff, 1) < 1   // wait for char
      if secondscpu() > (start + maxDelay)
         return out
      endif
   enddo
   out += buff
   if len(out) >= nSize
      exit
   endif
   start := secondscpu()
enddo
return out
```

Classification:

system, file access

Related:

FCLOSE(), FCREATE(), FEOF(), FERROR(), FOPEN(), FREADSTR(), FREADTXT(), FSEEK(), FWRITE(), FLOCKF()

FREADSTDIN ()

Syntax:

retN = FREADSTDIN (@varC1, expN2, [expN3])

Purpose:

Reads data from stdin into character buffer and optionally performs additional filtering and translation.

Arguments:

<**varC1**> is an already existing and initialized memory variable of character type used as a buffer. Its length must be at least <expN2>. Note: FlagShip support character variables up to 2 GB of size. This variable must be passed by reference to FREADSTDIN().

<**expN2**> is the max. number of characters to be read into the buffer. <expN2> must be <= len(expC1).

Options:

<**expN3**> is a optional filter and translation flag. The values below may be added together for a multiple choice.

0	default, no filter
1	ignore all CR and LF at the end of input (or buffer)
2	ignore all CR (0x0D)
4	ignore all LF (0x0A)
8	ignore all ^Z (0x1A)
16	ignore chars < " " (0x20)
32	replace TAB (0x07) by space
64	replace all chars < " " (0x20) by space

Additional translations are possible in the application by e.g. STRTRAN().

Returns:

<**retN**> represents the number of bytes that were read successfully. This value should be equal or less to <expN2>. If the return value is smaller than <expN2> or equals zero, it means that the entered string / input stream was smaller than the buffer size, the end of file has been reached or an error has occurred. A negative value signals an error:

-1	less than 2 arguments passed
-2	wrong type of argument 1 or 2
-3	wrong type of argument 3 if given
-4	varC1 is not passed by reference
-5	len(varC1) < 1 or expN2 < 1

Description:

FREADSTDIN() is a low-level i/o function that reads characters from standard input into an existing character buffer. It is a special case of FREAD(1, @varC1, expN2) with additional functionality.

When reading data from the console, the system transmit it when the RETURN or ^D key is pressed. When the buffer size is too small for storing the received data, the input stream is truncated to the size of <expN2>.

When reading data from a file (e.g. by an input redirection on the command line), the data are read until EOF or the <expN2> size is reached - whichever comes first.

Note: since the input may include null-bytes \0, you may use STRLEN() or <retN> to extract or print the significant part of the buffer in your application.

Note: since the Curses or IOCTL initialization is not required (see SYS.2.7), you may freely use this function also for background or CGI processing. In fact, WebGetFormData() uses it (with filter 1) for POST tags.

Example:

```
local buff := space(10000)
nread := FReadStdin(@buff, 10000, 63)  // large filter
if nread >= 0
   ? "Data read: ", left(@buff,nread)
else
   ? "error", ltrim(nread)
endif
```

Compatibility:

Available in FS only

Related:

FREAD(), FREADSTR(), FREADTXT(), LEN(), STRLEN(), WebGetFormData(), SYS.2.7

FREADSTR ()

Syntax:

retC = FREADSTR (expN1, expN2, [expL3])

Purpose:

Reads character strings from the specified file into a character variable.

Arguments:

<**expN1**> is the file handle previously returned by FOPEN() or FCREATE().

<**expN2**> is the maximum number of characters to read, beginning at the current file pointer position.

<**expL3**> is optional logical value. If true (.T.), FREADSTR() will retry the read on failure. The default is .F. = no retry.

Returns:

<**retC**> is the returned character string. If the end of file is encountered or an error occurs, a null string is returned.

Description:

FREADSTR () is low-level function similar to FREAD(). It starts reading at the current file pointer position and reads the number of characters specified. The string returned may be shorter than <expN2>, if a null character (zero byte CHR (0)) was encountered and FS_SET("zerobyte") is not set. The file pointer is moved forward <expN2> bytes (or to end-of-file), regardless of the string length returned.

Example:

```
LOCAL handle, text
text = ""
handle = FOPEN("myfile.txt")
DO WHILE .not. FEOF(handle)
   ? text := FREADSTR (handle, 50), FERROR()
ENDDO
FCLOSE (handle)
```

Classification:

system, file access

Compatibility:

In C5, the FERROR() is not set reaching the end-of-file. Also, the returned string is always shortened, if zero byte is encountered. FS supports zero bytes, if FS_SET("zero") is set. Parameter 3 is new in FS6.

Related:

FCLOSE(), FCREATE(), FEOF(), FERROR(), FOPEN(), FREAD(), FREADTXT(), FSEEK(), FWRITE(), FLOCKF()

FREADTXT ()

Syntax:

retC = FREADTXT (expN1, [expN2], [expL3])

Purpose:

Reads text lines from the specified ASCII file into a character variable.

Arguments:

<**expN**> is the file handle previously returned by FOPEN() or FCREATE().

<**expN2**> is optional maximum size of the text line. If not specified, 512 bytes is assumed.

<**expL3**> is optional logical value. If true (.T.), FREADTXT() will retry the read on failure. The default is .F. = no retry.

Returns:

<**retC**> is the returned character string containing the text line. If the end of file is encountered or an error occurs, a null string "" is returned.

Description:

FREADTXT () is low-level function to read ASCII files, similar to MEMOREAD() and MEMOLINE(), but it reads one text line only. It starts reading at the current file pointer position until the end- of-line mark CR/LF or LF is encountered or the end-of-file is reached.

The end-of-line mark is not stored in the returned string <retC>. The file pointer is then positioned on the byte following the end- of-line mark. Upon reaching the end-of-file, FERROR() is set.

If you wish to read the file at-once, use MemoRead() or Fread() or FreadStr().

Example:

```
#include "fileio.fh"
LOCAL handle, text
handle = FOPEN("myfile.txt", FO_READ)
DO WHILE ! FEOF(handle)
   ? text := FREADTXT (handle)
ENDDO
FCLOSE (handle)
```

Classification:

system, file access

Compatibility:

Available in FlagShip only. Zero bytes are supported, when FS_SET("zero") is set. Parameter 2 and 3 are new in FS6.

Related:

FCLOSE(), FCREATE(), FEOF(), FERROR(), FOPEN(), FREAD(), FSEEK(), FWRITE(), FLOCKF(), FS_SET()

FRENAME ()

Syntax:

retN = FRENAME (expC1, expC2, [expL3])

Purpose:

Changes the name of a file.

Arguments:

<**expC1**> is the old name of the file to rename. Standard UNIX wildcards are supported.

<**expC2**> is the new name of the file. If only the directory (except the dot . alone) is specified, the file(s) from <expC1> are **moved** to the directory given in <expC2>.

Both <expC1> and <expC1> file names have to include the extension and can optionally be prefaced by a path designator. SET PATH and SET DEFAULT are not considered.

Options:

<**expL3**> is a optional logical expression. If specified .T., possible errors are reported as "General Warning". Default is .F. which do not repport errors, only <retN> is set accordingly.

Returns:

<**retN**> is zero if the operation succeeds, -1 in the case of failure. FERROR() can be used to determine the nature of the error.

Description:

FRENAME() is an equivalent function to the RENAME command. If the destination file exists, it is deleted first to avoid access error by rename. If rename fails (e.g. for different drives or file systems), FRename() tries to copy source to destination and on success deletes the source thereafter.

The <expC1> file must be closed before renaming, otherwise failure or unpredictable errors may occur. When a database file .dbf is renamed, it is necessary to RENAME the associated memo file .dbt as well.

Example:

```
IF FRENAME ("file1.txt", "/usr/peter/file2.txt") < 0
   ? "error", FERROR(), "occurred"
ENDIF
```

Classification:

system, file access

Compatibility:

The FRENAME() function and RENAME command are equivalent to the UNIX command "mv" or the similar DOS command "RENAME". The usage of wildcards and the destination directory is available in FlagShip only.

FRENAME() considers the automatic path and/or conversion using e.g. FS_SET("pathlower") and FS_SET ("lower"), the extension replacement using FS_SET ("translext") and the drive substitution using the environment variable x_FSDRIVE.

The 3rd parameter is available in FlagShip6 and newer only.

Related:

RENAME, FERROR(), FILE(), ERASE, FERASE(), COPY TO, FCOPY(), FS_SET(), UNIX: man mv, DOS: rename /?

FSEEK ()

Syntax:

retN = FSEEK (expN1, expN2 [,expN3])

Purpose:

Moves the file pointer to a new position.

Arguments:

<**expN1**> is the file handle previously returned by FOPEN () or FCREATE ().

<**expN2**> is the number of bytes to move the file pointer from the position specified by <expN3>. The number of bytes specified can be positive or negative, depending on the direction of movement through the file.

Options:

<**expN3**> specifies the method of moving the file pointer. If this argument is not specified, zero is assumed.

value	#define	The file pointer is moved from
0	FS_START	the beginning of file
1	FS_RELATIVE	the current pointer position
2	FS_END	the end of file

Returns:

<**retN**> is the position of the file pointer after the operation, counting from the beginning of file, regardless of the moving mode <expN3>.

Description:

FSEEK () is low-level function to move the file pointer forward and backward in an open binary file.

Warning: FSEEK() allows a low-level access to the file system, similar to the C function lseek(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Example:

```
LOCAL actpos, maxpos, handle, txt
handle := FOPEN("text.asc")
IF handle # 0
   RETURN
ENDIF
maxpos := FSEEK (handle, 0, FS_END)    // last available
FSEEK (handle, 0, FS_SET)              // go to file begin
FREADTXT (handle)                      // skip the line 1
actpos := FSEEK(handle, 0, 1)          // set actual posit.
txt := FREADTXT (handle)               // read next line
FSEEK (handle, actpos)                 // go back
? FREADTXT (handle)                    // re-read again
FCLOSE (handle)
```

Classification:

system, file access

Compatibility:

In UNIX, it is legal to move the file pointer beyond the end-of- file. Be very careful when doing this, since it is possible to write one byte, move one gigabyte forward and write another byte. When finished, the file is two bytes long, but you can reach the second byte only by issuing the gigabyte seek again.

Clipper defines FS_SET instead of FS_START for the 0 value.

Include:

<FlagShip_dir>/include/fileio.fh #define's for <expN3>
<FlagShip_dir>/include/error.fh #define's of error codes

Related:

FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FWRITE(), FREADTXT()

FS_SET ()

The function FS_SET is an extension to the Clipper or dBASE settings (with the command SET ...). We decided to use only one universal additional function instead of creating new extended SET commands to ensure backwards compatibility to other DOS systems. However, you may define shortcuts for your most frequently used settings using the user-defined-commands, e.g.

```
#command SET LOWERFILE => FS_SET("lower", .T.)
#command SET LOWERPATH => FS_SET("pathlower", .T.)
```

etc. and use the alternative e.g. SET LOWERFILE instead. The typical use of this FS_SET() function is:

```
public FlagShip, Clipper        |   // or using the FS preprocessor:
:                               |
if FlagShip                     |   #ifdef FlagShip
   ok = FS_SET (....)           |      ok = FS_SET (....)
endif                           |   #endif
SET PATH ....                   |   SET PATH ....
```

On the DOS system (e.g. Clipper'87), all you need is to create a dummy function FS_SET (defined in the file <FlagShip_dir>/system/fstodos.prg) such as:

```
* File FSTODOS.PRG :   DOS compatibility to FlagShip

function FS_SET
parameters p1,p2,p3
return .T.
```

which will be linked to your application by:

```
PLINK86|RTLINK|BLINKER fi myprog, fstodos  [lib clipper, extend]
```

Using the alternative #ifdef FlagShip ... #endif syntax and Clipper 5.x, no additional changes are necessary.

The required action is always passed to the FS_SET function as a string in the first parameter. The second and third parameters specify the options. The action string may be given either in upper or lowercase and may be shortened up to the first 3 characters. So all the statements FS_SET("dev"), FS_SET('DEVEL') and FS_SET ("Developer") are identical, but the statement FS_SET ("Develxxx") is wrong.

Reference of the FS_SET() functions in logical groups:

FS_SET ("devel")	Sets/checks the developer/end-product running mode
FS_SET ("break")	Sets the "break" key
FS_SET ("debug")	Sets the "debug" activation key
FS_SET ("lower")	Converts file names to lowercase

FS_SET ("upper")	Converts file names to uppercase
FS_SET ("pathlow")	Converts given path to lowercase
FS_SET ("pathupp")	Converts given path to uppercase
FS_SET ("pathdeli")	Sets additional SET PATH separators
FS_SET ("shortnam")	Truncates file names to become compatible to DOS
FS_SET ("translext")	Translates the file extension to another
FS_SET ("print")	Determines the name of the printer spooler file
FS_SET ("prset")	Determines or sets the printer behavior
FS_SET ("term")	Determines the active terminal or mapping
FS_SET ("escdelay")	Sets/checks the delay time of the ESC key
FS_SET ("typeah")	Controls the type-ahead capability for curses
FS_SET ("inmap")	Sets/checks the character mapping for keyboard input
FS_SET ("outmap")	Sets/checks the character mapping for screen output
FS_SET ("loadlang")	Loads a user defined sorting and language table
FS_SET ("setlang")	Sets/checks the loaded sorting and language table
FS_SET ("speckeys")	Translates special key esc-sequence to Inkey value
FS_SET ("ansi2oem")	Sets new Ansi2oem() and Oem2Ansi() translation table
FS_SET ("guikeys")	Sets/checks the loaded GUI keyboard table
FS_SET ("intvar")	Enables the differentiating between "N" and "I" var types
FS_SET ("memcomp")	Sets full compatibility of the .mem files to Clipper
FS_SET ("zerobyte")	Enables the usage of chr(0) within a string

Note: when using manifests (defines) from the <FlagShip_dir>/include/ fileio.fh file for the low-level file access, it is recommended to specify the FS_SET() function in lower case to avoid conflicts with the FS_SET manifest.

The following description of FS_SET() functions is given in alphabetical order.

FS_SET ("ansi2oem")

Syntax:

`retC = FS_SET ("ansi2oem" [, expC2])`

Purpose:

Redefines the used translation table for Ansi2oem() and Oem2Ansi(), i.e. for translation from Ansi (ISO, Latin) to OEM (ASCII, PC8) and vice versa. These functions are also used when SET GUITRANS ASCII and/or SET SOURCE ASCII/ISO and/or SET GUITRANS TEXT ON and/or SET ASCII are set. See also LNG.5.4 for discussion about national character set and internationalization.

Options:

<**expC2**> is a character expression specifying the file name with the translation mapping (see section 5.4). If the path is not given, the search order is:

a. the current directory,
b. the path in environment var SCRMAP=/path/path
c. the path in envir. variable FSTERMINFO=/path/path
d. the path in envir. variable TERMINFO=/path/path
e. the FlagShip path according to SET PATH TO ...
f. the FlagShip_Dir()/terminfo directory

The default file is <FlagShip_dir>/terminfo/FSansi2oem.def which corresponds to IBM PC-8 <-> ISO-8859-1 (Latin1) translation; those equivalence is used internally if not explicitly redefined.

If no parameter <expC2> is given, or the file is not found, only the current status is returned, errors are reported when FS_SET("devel") is active.

Returns:

<**retC**> is a character value specifying the file name previously set by FS_SET("ansi2oem"). An empty string "" is returned, when the default build-in table is used.

Classification:

programming, internationalization

Compatibility:

Available in FlagShip 5 (VFS) only.

Related:

SET ANSI, SET GUITRANS ASCII, LNG.5.4

FS_SET ("break")

Syntax:

`retN = FS_SET ("breakkey" [,expN])`

Purpose:

Determines or changes the default FlagShip "break" hotkey (^K). Usable for terminals with "hard coded" special keys.

Arguments:

<**expN**> is an optional numeric expression representing the new hotkey. The default is 11, which is the decimal equivalent of the control-K key, or 0 in FlagShip/Windows.

Returns:

<**retN**> is a numeric value, representing the current "break" hotkey (see also section FSC). Before resetting with the <expN> argument, 11 will be returned.

Description:

Reset or determine the special interrupt key. All the other FlagShip special keys (cursor movement, function keys etc.) are defined by the terminfo description and can therefore be modified by the user. See also section SYS for description of FStinfo.src and FSkeymap.def.

In FlagShip for Windows, the break key is disabled by default (for performance reasons), you need to activate it by e.g. FS_SET("break", K_CTRL_K) or FS_SET("break", K_ALT_C) The break key is then recognized at any input status.

At redefinition be careful in order to avoid conflicts with other keys. See Appendix A3..A7 for the default control keys.

Example:

```
term = FS_SET ("terminal")        && determine act.terminal
if terminal == "FSxyz"            && is it a special one?
   FS_SET ("break", 16)           && yes, change break key
   wait "Note: hotkey for break is now ctrl-P"
else
   ? "Program break is possible with ctrl-" + ;
     chr(FS_SET("break") + 64)
endif
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FS_SET("debugkey")

FS_SET ("debug")

Syntax:

`retN = FS_SET ("debugkey" [,expN])`

Purpose:

Determines or changes the default FlagShip "debug" activation key (^O). Usable for terminals with "hard coded" special keys.

Options:

<**expN**> is an optional numeric expression representing the new activation key. The default is 15, which is the decimal equivalent of the control-O key.

Returns:

<**retN**> is a numeric value, representing the current activation key for the FlagShip debugger (see also section FSC). Before resetting with the <expN> argument, 15 (or 0) will be returned.

Description:

Reset or determine the special interrupt key. All the other FlagShip special keys (cursor movement, function keys etc.) are defined by the terminfo description and can therefore be modified by the user. See also section SYS for description of FStinfo.src and FSkeymap.def.

In FlagShip for Windows, the debug key is disabled by default (for performance reasons), you need to activate it by e.g. FS_SET("debug", K_CTRL_O) or FS_SET("debug", K_ALT_D) The debug key is then recognized at any input status.

At redefinition, be careful to avoid conflicts with other keys. See Appendix A3..A7 for the default control keys.

Example:

```
term = FS_SET ("terminal")        && determine act.terminal
if terminal == "FSxyz"            && is it a special one?
   FS_SET ("debug", 14)           && yes, change debug key
   wait "Note: debugger activation changed to ctrl-N"
endif
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

ALTD(), FS_SET("breakkey")

FS_SET ("develop")

Syntax:

retL = FS_SET ("developer" [,expL2])

Purpose:

Sets/decides if the FlagShip application runs in the "developer" or in the "final" mode.

Options:

<**expL2**> logical TRUE for "test" mode, FALSE for the "final" mode. Default: FALSE .F. (final mode).

Returns:

<**retL**> is a logical value, representing the current status of this mode when the FS_SET() function is entered.

Description:

The FlagShip error system can decide whether the application is running for debugging and testing or is "finished". In test mode FlagShip will also show minor programming errors and warnings (like a wrong PICTURE clause etc.), but the user will not receive any minor message errors in the "final" mode.

Errors

will be displayed only during "developer" mode. With wrong parameters, the return code becomes FALSE.

Example:

```
* Program test
parameters cmd
public FlagShip, Clipper
if pcount() > 0
   if "/TEST" $ upper(cmd)
      FS_SET ("devel", .T.)
   endif
endif
test_mode = FS_SET ("developer") .and. FlagShip
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FlagShip error system

FS_SET ("escdelay")

Syntax:

retN = FS_SET ("escdelay" [, expN2])

Purpose:

Determines the delay between the moments the ESC key is pressed and transmitted.

Options:

<**expN2**> is a numeric value representing the delay time in tenths of seconds (1/10 sec). The valid range is 1 to 100 (i.e. 0.1 to 10 seconds), the default value is normally 10 (1 second). When the argument is omitted, the current delay value is returned.

Returns:

<**retN**> is the current delay time in tenth of second prior to executing the FS_SET() function.

Description:

On UNIX, the special keys (like cursor movement, function keys etc.) are represented by an escape sequence string, see also terminfo and section SYS.

Therefore, when the keyboard driver detects the ESC character, it waits a while to check if an additional sequence follows. If so, the whole sequence (string) is compared with the terminfo entry to determine the special key. Otherwise, the Escape key is returned.

The FS_SET("esc") function changes the delay which follows after the ESC key is pressed. Be careful to set the value properly considering the current transmission speed of the terminal (if below 300 baud).

Note: on some systems, a delay time of 10 == 1 second will disable the trapping of the ESC key. On these systems, the delay is set to 9 = 0.9 seconds per default.

Refer also to the section REL for differences listed for your operating system and hardware, if any.

Example:

```
? FS_SET ("esc")                    // 10
FS_SET ("esc", 3)                   // set delay to 0.3 sec.
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

terminfo

FS_SET ("guikeys")

Syntax:

`retC = FS_SET ("guikeys" [, expC2])`

Purpose:

Determines and/or sets the used keyboard translation table. Apply for GUI mode only, ignored otherwise. See also LNG.5.4 for national character set and internationalization.

Options:

<**expC2**> is a character expression specifying the file name with the GUI keyboard mapping (see section 5.4). If the path is not given, the search order is:

a. the current directory,
b. the path in environment var SCRMAP=/path/path
c. the path in envir. variable FSTERMINFO=/path/path
d. the path in envir. variable TERMINFO=/path/path
e. the FlagShip path according to SET PATH TO ...
f. the FlagShip_Dir()/terminfo directory

The default file is <FlagShip_dir>/terminfo/FSguikeys.def

If no parameter <expC2> is given, or the file is not found, only the current status is returned, errors are reported when FS_SET("devel") is active.

Returns:

<**retC**> is a character value specifying the file name previously set by FSGUIKEYS environment variable or by FS_SET("guikeys",file). An empty string "" is returned, when the default build-in table is used.

Description:

FlagShip supports a user defined character mapping for different keyboards both in GUI and Terminal i/o mode.

On program beginning in GUI mode, the main module looks if the environment variable FSGUIKEYS is set and if so, uses the keyboard mapping file specified with it. If not defined, the build-in table corresponding to the FSguikeys.def file is used.

When the application is running in Terminal i/o mode, FSkeymap.def table is used instead, see FS_SET("inmap") below and LNG.5.4.

Classification:

programming

Compatibility:

Available in FlagShip 5 (VFS) only.

Related:

FS_SET("inmap")

FS_SET ("inmap")

Syntax:

```
retL = FS_SET ("inmap" [, expC2 [, expC3]])
```

Purpose:

Determines, enables or disables an additional mapping for the keyboard input. Apply for terminal i/o mode, see also LNG.5.4

Options:

<**expC2**> is a character expression specifying the file name for the character mapping (see section SYS). Default: "FSkeymap.def". The search order, if no path is given is:

a. the current directory,
b. the path in environment var SCRMAP=/path/path
c. the path in envir. variable TERMINFO=/path/path
d. the FlagShip path according to SET PATH TO ...
e. the default 'terminfo' path
f. the FlagShip_dir()/terminfo directory

If no parameter <expC2> is given, only the current status is returned. An empty string "" disables the user defined character setting. A 1:1 mapping is then used.

<**expC3**> is a character expression specifying the terminal name. If the argument is omitted, FS_SET looks for the terminal name as given in the UNIX environment variable TERM.

Returns:

<**retL**> is a logical value. If argument <expC2> and <expC3> are not given, the current status of this mode on entry of the FS_SET() function is returned. Otherwise <retL> gives the current status after the setting. A TRUE value means that the character mapping is currently active.

Description:

FlagShip supports a user defined character mapping for different keyboards, such as ISO or 7-bit. For more information, refer to the section SYS and LNG.5.4.

On program beginning of Terminal i/o based application, the main module looks for the file "FSkeymap.def" and an entry there for the current terminal (TERM). You can change these defaults using the "inmap" option setting.

When the application is running in GUI i/o mode, the FSguikeys.def table is used instead, see FS_SET("guikeys") above and LNG.5.4.

Additional information is available via SET(_SET_TERM_DATA)

Example:

```
act_out_file = ""
act_out_term = ""
act_in_file = ""
act_in_term = ""
IF FS_SET ("mapping")
   FS_SET ("terminal", @act_out_file, @act_out_term, ;
                       @act_in_file,  @act_in_term )
   ? "current mapping from", act_in_file, "with", act_in_term
   IF FS_SET ("inmap")
      ? "input-mapping is set to ", act_term
   ENDIF
ELSE
   ? "no character mapping active"
ENDIF
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FS_SET("guikeys"), FS_SET("term"), FS_SET("outmap"), SET(_SET_TERM_DATA)

FS_SET ("intvar")

Syntax:

```
retL = FS_SET ("intvar" [,expL2])
```

Purpose:

Enables or disables differentiating between "N", "F" and "I" variable types.

Options:

<**expL2**> is a logical expression. A TRUE value enables the TYPE() and VALTYPE() functions to distinguish between a floating point "N" (NUMERIC type) and integer "I" (INTVAR type), or "F" (float numeric field) variables. The default is FALSE, which displays "N" for all three types.

Returns:

<**retL**> is a logical value, representing the current, or previously set displaying status.

Description:

FlagShip internally distinguish between usual numeric (floating point) variables and integer variables. To be compatible to former (and Clipper, VO) applications, the TYPE() and VALTYPE() functions usually returns "N" for all the types "N", "F" and "I".

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

TYPE(), VALTYPE(), LOCAL..AS, INT2NUM(), NUM2INT()

FS_SET ("loadLang")

Syntax:

`retL = FS_SET ("loadlanguage", expN2, expC3)`

Purpose:

Loads the sorting and names table specified from an ASCII file to select it later using FS_SET("setlang").

Arguments:

<**expN2**> is a numeric expression specifying the ordinal number of the current language to select it later. The valid range is 1 to 9.

<**expC3**> is a character expression specifying the name of the ASCII file, containing the language sorting table (see section SYS). The search order for the file, if no path is given, is:

a. the current directory,
b. the path in environment var SCRMAP=/path/path
c. the path in envir. variable TERMINFO=/path/path
d. the FlagShip path according to SET PATH TO ...
e. the default 'terminfo' path
f. the FlagShip_dir()/terminfo directory

During the FlagShip installation, you obtain at least one language description in <FlagShip_dir>/terminfo/FSsortab.xxx; see GEN.3.2.

Returns:

<**retL**> contains TRUE if the table was found and read correctly.

Description:

FlagShip supports user defined sorting tables, including also the upper/lower conversion and standard output messages, for any foreign human language. The data is defined in an external ASCII file; see sections LNG.9.4 and SYS. On program start, the default ASCII sorting order and English messages are active.

The programmer must specify which sorting file (or files) should be read into the internal FlagShip buffer. Up to 9 different language tables may be read in the memory and selected later by the function FS_SET("setLanguage").

Warning: Attempting to use the database with the language other than the one it was filled with, may lead to unpredictable results.

The current, active sorting table is stored in the index header by executing the INDEX ON command. Using this index later with another sorting table setting will result in a run-time warning while the index file is open, since some of the index keys may not be found using SEEK, SKIP, LOCATE or may not be found in the expected order.

Example:

```
ok = FS_SET ("load", 1, "FSsortab.ger")
ok = FS_SET ("load", 2, "espan.dat")
if .not. FS_SET ("load", 3, "france.dat")
   ? "no french output possible"
endif
FS_SET ("setLang", 1)                    && german
IF pcount() > 0
   IF "/ESP" $ upper(cmd_line)           && spanish
      FS_SET ("set", 2)
   ELSEIF "/FR" $ upper(cmd_line)        && french
      FS_SET ("set", 3)
   ELSEIF "/ENGL" $ upper(cmd_line)      && default: english
      FS_SET ("set", 0)
   ENDIF
ENDIF
act_lang = FS_SET ("set")
? "current language = " + ltrim(str(act_lang))
? CDOW(DATE())                    &&  Monday or Montag etc.

USE address
INDEX ON name TO name                    // setting in .idx header
USE custom NEW
FS_SET ("setlang", 0)
INDEX ON name TO custom                  // setting in .idx header
```

Classification:

programming

Compatibility:

Available in FlagShip only. Storing the sorting table in the index header is available in FS for the default "dbfidx" driver only.

Related:

FS_SET ("setlang"), INDEX ON, SEEK, ISALPHA(), ISLOWER(), ISUPPER(), DESCEND(), LOWER(), UPPER(), ASORT(), ASCAN()

FS_SET ("lower")
FS_SET ("upper")

Syntax:

```
retL = FS_SET ("lowerfiles" [, expL2])
retL = FS_SET ("upperfiles" [, expL2])
```

Purpose:

Translates the given file names to lower or upper case during the create, open or search file access.

Options:

<**expL2**> is TRUE (.T.) to enable automatic conversion of file names to lower or uppercase, FALSE (.F.) to disable it. The switch FS_SET("upper",.T.) disables the current setting of FS_SET("lower",.T.) and vice versa. At program start the <expL2> is set to FALSE.

Returns:

<**retL**> is a logical value, representing the current status of this mode when the FS_SET() function is entered.

Errors

will be displayed only during "developer" mode. With wrong parameters, the return code becomes FALSE.

Description:

Because of the upper/lowercase sensitivity of the file names on the UNIX system, there is big difference in coding "use address" and "use ADDRESS". On UNIX, the file names are generally given in lowercase. DOS, however, converts all file names to uppercase.

To create programs, which are fully portable between DOS and UNIX, FlagShip can be directed to automatically convert file names to upper or lower letters, regardless of how they are written in the program. If you enter FS_SET ("lower", .T.), FlagShip automatically converts the <file> names given in your program to lowercase during the file open processes for the statements

```
USE <file> ...
SET INDEX TO <file>
INDEX ON ... TO <file>
COPY TO <file>
APPEND FROM <file>
PRINT TO <file>
FCREATE (<file>)
FOPEN (<file>)
DIR [<files>]
FILE (<file>)
ADIR (<files>)
DIRECTORY ( [<files>])
```

The extensions (.dbf, .dbt, .dbv, .idx) will be automatically added in lowercase and also in uppercase. The conversion of "\" to "/" (see LNG.9.4) and path names is not affected by the setting of FS_SET("lower"| "upper").

Note, that the conversion affects the file names and extension only, but not the path which may have be given. To translate the given path, use FS_SET ("pathlower").

The translation is omitted for files starting with the /dev/* path

Example:

```
old_status = FS_SET ("lower", .T.)
stat_check = FS_SET ("lower")
? "The lower file name conversion was", old_status
? "is now set to", stat_check

FS_SET ("pathlower", .F.)
file name = "\Data\Addresses\ADDRESS"
USE &file name       && opens "/Data/Addresses/address.dbf"
set path to \xxx\Data  && per default: case sensitive
if FlagShip
   subdir = "/usr/Data/"
else
   subdir = "C:\globdata\DATA\"
endif
USE &subdir.ADRESS     && opens /usr/data/address.dbf
FS_SET ("lower", .T.)
? FILE("Abcd.EFG")     && looks for "abcd.efg" in
                       && act. directory or SET PATH
FS_SET ("low", old_status)
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Include:

<FlagShip_dir>/include/fspreset.fh mostly used settings

Related:

FS_SET ("pathlower") and section LNG.9.4, 9.5

FS_SET ("memcomp")

Syntax:

retL = FS_SET ("memcompat" [,expL2])

Purpose:

Enables or disables the full compatibility of the .mem files to other dialects.

Options:

<**expL2**> is a logical expression. A TRUE value enables the full .mem file compatibility, FALSE disables it. If <expL2> is not specified, the current status is returned. The default setting on startup is FALSE.

Returns:

<**retL**> is a logical value, representing the current .mem file compatibility.

Description:

In addition to the standard variables of the type character, numeric, logical and date, FlagShip can also store and restore the array contents and screen variables into .mem files.

Normally, the additional variable types are stored at the end of the .mem file and are therefore ignored by most of the xBase dialects (like Clipper or dBASE). If problems occur or you need to avoid the storing/restoring of PRIVATE and PUBLIC arrays or screen variables using SAVE TO, RESTORE FROM commands, set <expL2> to TRUE.

FS_SET("memcompat", .T.) should also be used for restoring FoxPro memory variables of C, N, D, L types.

Example:

```
PRIVATE varC, varN, varS, varA
varC := "any string"
varN := 1234.567
varS := SAVESCREEN (1,2, 20,30)
varA := {1,2, {"text", DATE()}}

* FS_SET ("memcomp", .T.)       // avoid SAVE varS, varA
SAVE TO mymem ALL LIKE var*

CLEAR MEMORY
* FS_SET ("memcomp", .T.)       // avoid RESTORE varS, varA
RESTORE FROM mymem
```

Classification: **p**rogramming

Compatibility:

Available in FlagShip only. Note, that the setting has no affect on the incompatibility between FS and C5 of the screen contents from SAVE SCREEN, REST SCREEN.

Related:

SAVE TO, RESTORE FROM, SAVESCREEN(), SAVE SCREEN

FS_SET ("outmap")
FS_SET ("mapping")

Syntax:

```
retL = FS_SET ("mapping" | "outmap" , [expC2] ,
              [expC3] )
```

Purpose:

Determines, enables or disables additional mapping for the screen output. Apply for terminal i/o mode, ignored otherwise. See also LNg.5.4 for internationalization.

Options:

<**expC2**> is a character expression specifying the file name for the character mapping (see section SYS). Default: "FSchrmap.def". The search order, if no path is given is:

a. the current directory,
b. the path in environment var SCRMAP=/path/path
c. the path in envir. variable TERMINFO=/path/path
d. the FlagShip path according to SET PATH TO ...
e. the default 'terminfo' path
f. the FlagShip_dir()/terminfo directory

If no parameter <expC2> is given, only the current status is returned. An empty string "" disables the user defined character setting. 1:1 mapping is then used.

<**expC3**> is a character expression specifying the terminal name. If the argument is omitted, FS_SET looks for the terminal name as given in the UNIX environment variable TERM.

Returns:

<**retL**> is a logical value. If argument <expC2> and <expC3> are not given, the current status of this mode on entry of the FS_SET() function is returned; otherwise <retL> gives the current status after the setting. A TRUE value means that the character mapping is currently active.

Description:

FlagShip supports user-defined character mapping for different terminals. For more information, refer to the section SYS. When the program starts up, the main module looks for the file "FSchrmap.def" and an entry there for the current terminal (TERM). You can change these defaults using this option setting.

Additional information is available via SET(_SET_TERM_DATA)

Example:

```
* Program test.prg
act_out_file = ""
act_out_term = ""
act_in_file  = ""
act_in_term  = ""

if FS_SET ("mapping")
   FS_SET ("terminal", @act_out_file, @act_out_term)
   ? "current mapping from", act_out_file, "with", act_out_term
endif
if FS_SET ("mapping", "mymap.dat")
   FS_SET ("term", @act_out_file, @act_out_term, ;
                   @act_in_file,  @act_in_term )
   ? "new mapping from", act_out_file, "with", act_out_term
else
   act_term = FS_SET ("term")
   ? "Terminal", act_term, "not in 'mymap.dat'"
   if FS_SET ("map", "mymap.dat", "ansi")
      FS_SET ("term", @act_out_file, @act_out_term, ;
                      @act_in_file,  @act_in_term )
      ? "new mapping from", act_out_file, "with", act_out_term
   else
      ? "no character mapping active"
   endif
endif
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FS_SET ("term"), FS_SET ("inmap") SET(_SET_TERM_DATA), section SYS.

FS_SET ("pathdelim")

Syntax:

retC = FS_SET ("pathdelim" [, expC2])

Purpose:

Determines or enables user-defined path separators for use when deleting multiple path names in the SET PATH command.

Options:

<**expC2**> is a character expression up to four characters specifying the new path separator. The default settings are the DOS separator characters, comma and semicolon ",;". If the <expC2> string exceeds four characters, only the first four are used.

Returns:

<**retC**> is a string specifying the current path separators. On program start or upon redefinition ",;" is returned.

Description:

In the SET PATH command, multiple path names to search for can be specified. The default separators between the path list are usually comma or semicolon.

If a UNIX path list should be included in the path list, using e.g. the GETE("PATH") function, an additional colon (":") and/or space (" ") separator is to be specified.

Example:

```
#include "set.fh"
LOCAL path1 := "\data;\address\data,\mydata", path2
FS_SET ("pathdel", ",; :")              && add space and colon
path2 := GETENV ("PATH")
SET PATH TO (path1 + "," + path2)
IF .not. FILE("address.dbf")
   ? "install the full package,"
   ? "data not available in directories " + SET(_SET_PATH)
   QUIT
ENDIF
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

SET PATH, FS_SET ("pathlow"), section LNG.9.3.

FS_SET ("pathlower")
FS_SET ("pathupper")

Syntax:

```
retL = FS_SET ("pathlower" [, expL2])
retL = FS_SET ("pathupper" [, expL2])
```

Purpose:

Translates the given paths to lower or upper case during the create, open or search file access.

Options:

set <**expL2**> to TRUE (.T.) to enable the automatic conversion of the path names to lower or uppercase, or to FALSE (.F.) to disable it. The switch FS_SET("pathupper",.T.) disables the current setting of FS_SET("pathlower",.T.) and vice versa. The <expL2> on program beginning is set to FALSE.

Returns:

<**retL**> is a logical value, representing the current status of this mode on entry of FS_SET() function.

Description:

Because of the upper/lowercase sensitivity of the file and path names in UNIX, coding "use \xyz\ address" means something completely different from "use \XYZ\ADDRESS". In UNIX, the file and path names are generally given in lowercase. DOS, however, converts all file names to uppercase.

To create fully compatible programs for DOS and UNIX, FlagShip can be directed to automatically convert path names to upper or lower letters, regardless of how they appear in the program. If you enter FS_SET ("pathlower", .T.), FlagShip automatically converts the path names given in your program to lowercase during the file open processes.

The conversion of a backslash "\" to a slash "/" is not affected by the setting of FS_SET ("pathlower" | "pathupper"), since FlagShip automatically converts it to the UNIX convention (refer also to LNG.9.4).

Note, that this conversion affects the path names only, but not the file names given. To translate the file names and extensions given, use FS_SET("lower").

The translation is omitted for files starting with the /dev/* path

Example:

See example in FS_SET ("lower")

Example:

Using the predefined settings from the include file:

```
#ifdef FlagShip
#include "fspreset.fh"
#endif
USE C:\XYZ\abc\DAta                    && .../xyz/abc/data
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Include:

<FlagShip_dir>/include/fspreset.fh most used settings

Related:

FS_SET("lower"), FS_SET("pathdelim"), x_FSDRIVE and section LNG.9.4

FS_SET ("print")

Syntax:

`retC = FS_SET ("printfile", [expLC1])`

Purpose:

Determines the current name of the printer-spooler file.

Options:

<**expLC1**> is optional logical or string value, controlling the SET PRINTER behavior.

If <expLC1> is a string, it specifies the name of default spooler file (used by SET PRINTER ON without SET PRINTER TO..), i.e. it re- defines the default "name.pid" spool file. The previously used file is deleted, if yet empty. The file name <expLC1> may contain absolute or relative path, and is used "as is", i.e. without appending ".prn" extension. The directory path, if given, must exist and must be read/write, otherwise RTE occurs.

If <expLC1> is .F., the default spooler file name will be disabled and deleted if yet empty. Subsequent SET PRINTER ON will raise RTE, but you may freely print to file or device specified by SET PRINTER TO <fileOrDevice>. To enable the default spooler file again, use FS_SET("printfile","yourFileName"), where the file name can include path (and Windows drive) as well.

Returns:

<**retC**> is a string containing the current printer output file including path (and drive), e.g. "/usr/data/address.2345" in Linux, or "D:\mydata\address.2345" in MS-Windows.

Description:

For printer output, FlagShip is designed to support the UNIX and Windows spooling mechanism, refer to LNG.3.4 and the SET PRINTER command.

On program start, FlagShip creates a file "**name.pid**" in the current directory (or in the directory specified by the environment variable FSOUTPUT). The "name" part reflects the main program module, and "pid" gives the current process ID number. If the resulting file name is longer than 14 characters, the program name will be shortened. You may redefine this standard spooler file at any time later by FS_SET("printfile","yourFileName"), see above.

Using SET PRINTER ON and/or SET DEVICE TO PRINT, the output is written to the output file instead of printing it directly to the printer device, unless SET PRINTER TO <device> is specified. To omit garbage being printed by several users printing simultaneously, the output should be done later using e.g.

```
   $ lp -dmatrix test.1234   ## or
   $ lpr -dlaser1 test.1234
or in Windows
   C:> copy test.1234 PRN:
```

or directly from the application using the RUN command. Refer to SET PRINTER command for a detailed description.

To create the spooler file in other than local directory at program start, use FSOUTPUT environment variable, see FSC.3.3, or re-define it later by FS_SET("printfile","yourFileName").

There is also Printer class available in FlagShip, which allows you to select printer via _oPrinter:Setup() and sent the output directly to it via _oPrinter:Exec() or _oPrinter:ExecFormatted(), see also <FlagShip_dir>/examples/printer.prg. Another alternative to print to GDI printer same as on screen is SET GUIPRINT ON or PrintGui(.T.) see <FlagShip_dir>/examples/printergui.prg.

Example 1:

```
SET PRINT ON
? "Text 1"
? "Text 2"
SET PRINT OFF
? "text to screen only"

SET PRINT ON
? "Text 3"
? "being in directory", CURDIR()            // e.g. /home/joe
SET DIRECTORY TO /tmp
? "and now changed to", CURDIR()            // now in: /tmp
SET PRINT OFF                               // print file unchanged

prnfile = FS_SET ("printfile")              //
"/home/joe/test.1234"
? "printing/spooling the file", prnfile
#ifdef FS_WIN32
   RUN ("copy " + prnfile + " PRN:")        // = Windows
#else
   RUN ("lp -dlaser2 -m -s " + prnfile)     // = Linux
*  RUN cp &prnfile /dev/lp1                 // alternative
*  RUN ("cat " + prnfile + " > /dev/lp0")   // alternative
#endif

SET PRINT ON NEW                            // deletes old file
? "the next output"                         // and writes new data
SET PRINT OFF                               // to
/home/joe/test.1234
```

Example 2:

```
// re-define the standard spooler file to "/tmp/applic.ext.1234"
// and delete old file, if yet empty
oldFile := FS_SET("print")
FS_SET("print", "/tmp/" + ExecName() + "." + ltrim(ExecPidNum()) )
? "default spooler file", oldFile, "is now", FS_SET("print")
```

Classification: programming
Compatibility: available in FlagShip only
Related:

SET PRINTER, SET GUIPRINT, PrintGui(), _oPrinter:Exec(), Exec*(), UNIX: man lp or man lpr, ls /dev/lp*

FS_SET ("prset")

Syntax:

```
retA = FS_SET ("prset", [aData])
```

Purpose:

Determines and/or sets the printer sequences for a fine output tuning.

Options:

<**aData**> is an optional array containing zero to nine elements with strings of printer sequences.

All settings apply for device printer output (@...SAY, devpos() and SET DEVICE TO PRINT), some of them also the sequential output (?, qout(), eject and SET PRINTER ON, SET ALTERNATE, SET EXTRA). Does not affect the SET CONSOLE, SET DEVICE TO SCREEN.

Each element may contain up to 50 characters or escape-sequences, oversized strings will be truncated. To reset the corresponding element, pass null-string. Elements of other than char type are ignored.

aData[1] = cNewLine: printer sequence for movement to new line. Used for movement to the next line (row). The default sequence is chr(10) = LineFeed in Unix/Linux and chr(13)+chr(10) in MS-Windows. When the printer or lp filter does not accept the LF alone but requires CR+LF as in DOS, set it to chr(13) + chr(10). Affects both the sequential and device printer driver. For SET ALTERNATE and SET EXTRA new-line sequence, see element 8 and 9 below.

aData[2] = cNewPage: printer sequence for movement to new page. Used to feed the next page. The default sequence is chr(12) = Form- Feed. Affects the sequential and device printer driver.

aData[3] = cLineMove: printer sequence executed before moving to the specified column. When set (default is unset, i.e. null string ""), the driver sets pcol() to zero and executes the given sequence before each column movement. Used e.g. to move the printer "head" to the leftmost position and subsequently nMarg-times nMargin + nCol-times the <cSpace> sequence to reach the specified column according to the optional SET MARGIN TO <nMarg> and the current @ <nRow>,<nCol> SAY commands. Usable when the SAY string contains also ESC sequences eg. for char set or font switching, which will confuse the pcol() and would require correction by setprc() otherwise or when using proportional character set. If the <cLineMove> is set, aData[4] = <cLineHome> and aData[5] = <cPos2left> are ignored. Affects only the device printer driver.

aData[4] = cLineHome: printer sequence to move the print "head" to the leftmost column. As opposite to <cLineMove>, this sequence is executed only for "backward" moving in the same row i.e. the new <nColumn> is lower than the current pcol() value and the aData[3] = <cLineMove> is not specified. You may set the value to e.g. chr(13) when the <cPos2Left> sequence is not applicable for your printer. If set

(default is unset = null string "") and the printer head should move to left, the driver resets pcol() to 0, executes this <cLineHome> sequence and moves the head to the right by <nMarg> * <cMargin> plus <nCol> * <cPos2right> sequences. Affects only the device printer driver.

aData[5] = cPos2left: printer sequence to move one char to left. The default is set to chr(8) = Backspace. This sequence is executed by default for backward movement in the same row, but only when both aData[3] and aData[4] are not specified. Affects only the device printer driver.

aData[6] = cPos2right: printer sequence to move one char to the right. The default is chr(32) = Space, but you may set it e.g. to Esc sequence corresponding to a movement of a fix number of points when a proportional font is used. Affects only the device printer driver.

aData[7] = cMargin: sequence for printing one margin column, if such is set in the application by SET MARGIN TO <nMarg>. Default is chr(32) = Space. Affects both the sequential and device printer driver.

aData[8] = cNewLineAltern: sequence for new-line with SET ALTERNATE TO.. and SET ALTERNATE ON. The default is chr(10) = LineFeed; you may set it upon request e.g. to chr(13,10) for DOS/Windows CR+LF.

aData[9] = cNewLineExtra: sequence for new-line with SET EXTRA TO.. and SET EXTRA ON. The default is chr(10) = LineFeed; you may set it upon request e.g. to chr(13,10) for DOS/Windows CR+LF.

Returns:

<**retA**> is an array containing the current printer settings at the time the FS_SET() function is entered. The array elements corresponds to above described aData[].

Description:

For printer output, FlagShip usually uses default settings, i.e. spaces or backspaces for column positioning, LineFeed for new line movement etc. These are in the most cases sufficient for both the sequential and direct positioned output and are acceptable for any kind of printers.

However, when you use special escape sequences (e.g. for "bold" or "italics", font selection etc.), these sequences of course also counts as "usual characters" for PCOL() and therefore for the subsequent @...SAY output to printer. To synchronize the driver with the printer, you will need to reset the column by SETPRC() to the "real" printed characters in such a case.

Also when you use proportional fonts, the column setting by @..SAY or DISPOUT() will most probably produce insufficient results, especially for tables, since the column movement is done per default by space (or backspace) where the character width of space is very different e.g. to the "X" or "M" letter.

With FS_SET("prset") you can modify the behavior of your printer so, that no additional corrections in the source are required.

Example:

```
// #define PCL6
#define ESC  chr(27)

SET DEVICE TO PRINT
SET PRINTER ON
SET CONSOLE OFF
cSeqBold := ESC + "(s3B"  // here sequences for HL LJ III+
cSeqNorm := ESC + "(s0B"
cSeqItal := ESC + "(s1S"
#ifdef PCL6                 // e.g. HPLJ2100
  cSeqCour := ESC + "(10U" + ESC + "(s0p12h0s0b24579T"
  cSeqTime := ESC + "(10U" + ESC + "(s1p12v0s0b25093T"
  cSeqUprg := cSeqCour
#else
  cSeqCour := ESC + "(s3T"  // e.g. HPLJ3
  cSeqTime := ESC + "(s1T"
  cSeqUprg := ESC + "(s0S"
#endif

printme(1, "Default setting")      // print by default settings

aDefPrint := FS_SET("prset", { chr(13)+chr(10) } )
printme(10, "Using CR+LF for newline")

FS_SET("prset", {NIL, NIL, "", chr(13) } )
printme(20, "Using CR+LF and CR before backwise movement")

FS_SET("prset", {NIL, NIL, chr(13) + cSeqCour, "" } )
printme(30, "Using CR+LF and CR + " + ;
            "Courier10 before any column movement")

SET CONSOLE ON
SET PRINTER OFF
SET DEVICE TO SCREEN

? "printing file " + fs_set("print") + " via 'lpr'"
* RUN ("lpr -Php3 " + fs_set("print"))  // run spooler
wait "Done, any key..."
quit

FUNCT printme(line, text)
local ii, nPcol
@ line,0 say text
@ line +1, 0 say cSeqCour + ;
  "AaBb......BBB.......XXX.......fghi......|"
for ii := 2 to 7
   @ line+ii, 0 say cSeqCour + "AaBb"
   @ line+ii,10 say cSeqTime + repli("B",ii) + cSeqCour
   @ line+ii,20 say cSeqTime + cSeqItal + repli(chr(71+ii), ii+2)
   @ line+ii,30 say cSeqTime + cSeqBold + "fghijklm" + cSeqCour
   @ line+ii,40 say cSeqNorm + cSeqUprg + cSeqCour   // reset
   nPcol := pcol()
   ?? "|<- assumed=40, pcol()=" + ltrim(str(nPcol))
next
return
```

Classification:

programming

Compatibility:

Available in FlagShip only

Related:

SET PRINTER, SET DEVICE, @..SAY, ?, QOUT(), EJECT

FS_SET ("setenvir")

Syntax:

`retC = FS_SET ("setenvir", expC2, [expC3], [expL4])`

Purpose:

Determines or sets an Unix or Windows environment variable for the current process, and optionally, reinitializes the screen.

Arguments:

<**expC2**> is the name of the Unix/Windows shell environment variable and is case-sensitive.

Options:

<**expC3**> is the new value, to be assigned to the environment variable <expC2>. If the argument is not given or NIL is specified, the contents of the environment variable remain unchanged. Otherwise, if the variable <expC2> does not exist, a new one is created.

<**expL4**> specifies if the curses (and the screen, character mapping) are to be reinitialized. Any value other than TRUE leaves the initialization unchanged.

Returns:

<**retC**> is the current contents of the specified Unix or Windows environment variable after executing the function.

Description:

If only the argument <expC2> is specified, the function determines the current environment setting, as the GETENV() function.

Otherwise, this FS_SET() function sets a new value for a specific Unix/Windows environment variable creating this variable if it does not exist. The new setting is active during the execution of the current process (application) and its recently activated child processes (e.g. using the RUN command).

Tech info: if you invoke any process, the shell (sh, bash, csh,ksh in Linux/Unix or CMD in Windows) passes a COPY of the current environment to the child process (child = your executable or script). So changing the environment variable within the process changes it in the local environment copy - but neither Unix/Linux nor Windows can change parent's environment. See also "man bash" for Linux: "Changes made to the subshell (or subprocess) environment cannot affect the shell's execution environment." or the Microsoft docu: http://msdn.microsoft.com/en-us/library/ms682009%28VS.85%29.aspx "A process can never directly change the environment variables of another process that is not a child of that process."

If a new terminal variable TERM, COLUMNS or LINES is set or changed in the Terminal i/o mode, the curses and the screen should be reinitialized using the fourth argument. This will automatically reset the input/output mapping and clear the screen. Refer also to MAXCOL(), MAXROW() and GETENV() functions for more information.

Example 1:

```
LOCAL choice := 0
LOCAL old_term := new_term := GETENV ("TERM")
LOCAL term_file := "", term_map := ""
LOCAL terms := {"FSansi", "FSansi-50", "FSVT100", ;
                "FSvt100-50", "FSAT386"}
IF EMPTY(old_term) .or. SUBSTR(old_term,1,2) != "FS"
   ? "Select the correct terminal:"
   ? "1: ANSI"
   ? "2: ANSI, 50 lines"     // The alternative ACHOICE()
   ? "3: VT100, 200"         // may yet cause a screen
   ? "4: VT100/50 emulation" // garbage for wrong terminals
   ? "5: PC console"
   INPUT "Your choice (1..5 or ESC): " TO choice
   IF choice > 0 .and. choice <= 5
      new_term = FS_SET ("setenv","TERM", terms[choice], .T.)
   ENDIF
ENDIF
IF FS_SET ("terminal", @term_file, @term_map) != new_term
   ? "error on terminal setting, check TERMINFO environ"
   QUIT
ENDIF
? "The current terminal set is : " + new_term
? "The current mapping file is : "
IF empty(term_file) .or. ! FS_SET("inmap")
   ?? "inactive"
ELSE
   ?? term_file, "using", term_map
ENDIF
```

Example 2:

```
LOCAL newdrive := GETENV("D_FSDRIVE")
WHILE .T.
   IF EMPTY(newdrive)
      ACCEPT "Enter UNIX path for substituting the " + ;
             "DOS drive D:" to newdrive
   ENDIF
   IF !EMPTY(newdrive) .and. FILE(newdrive)
      EXIT
   ENDIF
   IF LASTKEY() == 27
      QUIT
   ENDIF
   ? "Error, try again" ; newdrive := ""
ENDDO
FS_SET ("setenv", "D_FSDRIVE", newdrive)
```

Classification: programming

Compatibility**: a**vailable in FlagShip only. On DOS, SETMODE() may be used to redefine the terminal.

Related:

GETENV(), GETE(), SETENV(), FS_SET("term"), FS_SET("outmap"), FS_SET("inmap"), MAXROW(), MAXCOL(), environment variables in section FSC and SYS, section REF: predefined terminals, UNIX: env, printenv, putenv, man term, man terminfo

FS_SET ("setLang")

Syntax:

`retN = FS_SET ("setlanguage" [,expN2])`

Purpose:

Enables, disables or determines the specific, already read language sorting and names table; see FS_SET("load").

Options:

<**expN2**> is a numeric expression specifying the ordinal number of the language table read by FS_SET ("load"). Valid values range from 1 to 9. If 0 (zero) is entered, the current sorting translation will be disabled. If the parameter <expN2> is not specified or is NIL, the current sorting status is returned.

Returns:

<**retN**> is the number of the current language currently set. If zero is returned, no language sorting is currently set, the sorting is ASCII and the messages in English.

Description:

FS_SET("setlang") is used to activate one of the required sorting tables, already read by FS_SET("load"). The function is also used to determine or disable the current sorting table.

Warning: Attempting to use the database with a language other than the one it was filled with, may lead to unpredictable results. When a new index file is opened (SET INDEX or USE..INDEX) with other sorting criteria than the current ones, a run-time warning occurs

Example:

```
LOCAL names := {"Myname", "Müller", "Muller"}
? FS_SET ("setl")                                    // 0
ASORT (names)
AEVAL(names, {|x| QOUT(x)} )      // Muller, Myname, Müller
? FS_SET ("load", 1, "FSsortab.ger")                 // .T.
? FS_SET ("setl", 1)                                 // 1
ASORT (names)
AEVAL(names, {|x| QOUT(x)} )      // Muller, Müller, Myname
? FS_SET ("load", 3, "france.dat")                   // .T.
? FS_SET ("setl", 3)                                 // 3
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FS_SET ("load"), INDEX ON, SEEK, ISALPHA(), ISLOWER(), ISUPPER(), DESCEND(), LOWER(), UPPER(), ASORT(), ASCAN()

FS_SET ("shortname")

Syntax:

```
retL = FS_SET ("shortname" [, expL2])
```

Purpose:

Translates long UNIX file and path names to the DOS naming convention during the create, open or file search access.

Options:

<**expL2**> logical TRUE to enable the DOS naming translation, FALSE to disable it. The default value is FALSE.

Returns:

<**retL**> is a logical value, representing the current status of this mode on entry of FS_SET() function.

Description:

On DOS, the full file name consists of four parts: the optional directory path, the main part of the file name (one to eight characters), a period (.) and an optional file extension (one to three characters). In UNIX, no special parts of the file name are used; the file name can include any number of periods. The file name can contain at least 14 characters, but may be longer on some UNIX systems. For more information refer to LNG.3.1.

Note: To maintain compatibility with DOS programs, FlagShip supports the DOS naming convention by default on access to databases, indices and text files.

When a FlagShip programmer creates long file names, the application may lose portability to other systems.

If the FS_SET("short") conversion is active, FlagShip abbreviates the given file name during the create, open or file search access to the minimal base, using the DOS naming convention, e.g.:

- All embedded spaces and special characters (other than A-Z, a- z, 0-9, # @ ! % ~ _ - . ? *) are removed.
- The main part of the file name (up to the first period ".") is abbreviated to eight characters, if necessary.
- When the extension consists of characters other than digits only (used e.g. for printer files), the extension is abbreviated up to three characters.
- Additional "extensions" (beyond the second period) are removed.

If the additional lower/upper translation is required, use also FS_SET("lower") and/or FS_SET("pathlower").

Note: The file names given are automatically left and right trimmed, regardless of the FS_SET("short") setting. Also with some database or file access commands, where the default extension may be omitted, FlagShip TRIMs the given filename on the left and on the right first, prior to adding the default ".dbf", ".idx", ".txt" or ".prn" extension.

Example 1:

```
file1 := this_is.unixfile"
file2 := "This Is A Valid UNIX Filename. On Some Systems"
file3 := "abc,defg=hij.1234"
SET PRINTER TO (file1)                  // creates long name
FS_SET ("short", .T.)
SET PRINTER TO (file1)                  // creates "this_is.uni"
FCREATE (file2)                         // creates "ThisIsAV.OnS"
FS_SET ("lower", .T.)
? FILE (file2)                          // search  "thisisav.ons"

SET EXTRA TO (file3)                    // creates "abcdefgh.1234"
```

Example 2:

Note that only the leading and trailing spaces are trimmed automatically. Both "mydata.dbf" and "mydata .dbf" are valid names of two different files:

```
LOCAL dbf_file := "  mydata  "
USE (dbf_file)                          // "mydata.dbf"
USE (dbf_file + ".dbf ")                // "mydata  .dbf"

FS_SET ("short", .T.)
USE (dbf_file + ".dbf ")                // "mydata.dbf"
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FS_SET("lower"), FS_SET("transl")

FS_SET ("speckeys")

Syntax 1:

```
retL = FS_SET ("speckeys", expC2, [expL3])
```

Syntax 2:

```
retC = FS_SET ("speckeys")
```

Purpose:

Determines and/or sets translation of special Esc-sequences to Inkey value. Apply for Terminal i/o mode in Unix/Linux, ignored otherwise.

Arguments:

<**expC2**> is a character expression specifying the file name with the Esc mapping table. If a path is not given, the search order is:

a. the current directory,
b. the path in environment var SCRMAP=/path/path
c. the path in envir. variable FSTERMINFO=/path/path
d. the path in envir. variable TERMINFO=/path/path
e. the FlagShip path according to SET PATH TO ...
f. the FlagShip_Dir()/terminfo directory

The default file is <FlagShip_dir>/terminfo/FSspeckeys.def

If the parameter <expC2> is not given (**syntax 2**), only the currently used file name is returned. If the parameter <expC2> is empty (""), the mapping is reset to default, corresponding to FSspeckeys.def.

Options:

<**expL3**> is logical value specifying optional protocol printout. If .T., the new setting (and possible errors) are printed to default SET PRINTER file. If <expL3> is .F., the output goes to stderr. If the argument is not given or is not logical, no protocol is printed.

Returns:

<**retL**> in syntax 1 reports success or failure. In development mode, i.e. with FS_SET("devel", .T.), errors are also reported as warning.

<**retC**> in syntax 2 reports the currently used file with translation table. Empty string ("") signals default setting.

Description:

In Terminal i/o mode, FlagShip uses [n]curses which handles keyboard input. Unfortunately is the curses capability limited for handling special keystrokes like Alt-key, Alt-Cursor etc. This FS_SET() table extends the [n]curses capability to handling any special key. For syntax, see the file <FlagShip_dir>/terminfo/FSspeckeys.def

This keyboard translation apply for Terminal i/o mode in Unix/Linux and is ignored otherwise. FlagShip port for **MS-Windows** uses pdcurses library which has predefined, fix keyboard translation.

Technical note:

For non-Unix gurus, here a rough description how the key-press comes to FlagShip's Inkey() and it related commands/functions such as READ, WAIT, ACCEPT, MENU TO etc. (in Terminal i/o mode):

- With active X11 (Gnome, KDE etc), keypress generates event stored in event queue; w/o X11, the scan code is stored in kernel buffer.
- X11 looks in xrdb table if the key should be translated to Escape sequence, and/or kernel looks in similar table set by loadkeys. This system translation table is set per default, or by loadkeys command, and/or by FSXTerm.xrdb.* script called during invocation of newfswin script.
- [n]curses calls X11 (or kernel) input function to get one input character, or the translated esc-sequence.
- If the key is Escape character, curses waits specified amount of time (usually 1 sec, see FS_SET("escdelay") for it setting), if another character(s) follows. If so, it looks in own translation table (aka terminfo "database", corresponding to current [FS]TERM and [FS]TERMINFO setting), to find matching special key.
- If curses knows the esc sequence, it returns the corresponding special character to Inkey(). If not, FlagShip checks the table specified by this FS_SET("speckey") function, and if the sequence is found, returns back the defined Inkey value.
- Otherwise when the delay time is not expired yet, curses waits for the next (translated) character of the sequence, to repeat above translation. If the delay time is expired, Inkey() returns the Esc value (27) and fills keyboard-ahead-buffer with the rest, if any.

Example:

```
? "'" + FS_SET("speckeys") + "'"          // '' = default
? FS_SET("speckeys", "FSspeckeys.def")    // .T.
? "'" + FS_SET("speckeys") + "'"          // '.../FSspeckeys.def'
```

Classification:

programming, input/output

Compatibility:

Available in FlagShip 6 (VFS) only.

Related:

Inkey(), READ, <FlagShip_dir>/terminfo/FSspeckeys.def

FS_SET ("terminal")

Syntax 1:

```
retC = FS_SET ("terminal")
```

Syntax 2:

```
retC = FS_SET ("terminal", @varC2, @varC3 )
```

Syntax 3:

```
retC = FS_SET ("terminal", @varC2, @varC3, @varC4,
           @varC5)
```

Purpose:

This function is used to determine

- the current terminal
- the current output character mapping, see FS_SET("outmap")
- the current output mapped terminal, see FS_SET("outmap")
- the current input character mapping, see FS_SET("inmap")
- the current input mapped terminal, see FS_SET("inmap")

Options:

<**varC2**> is a declared and initialized character variable (called by reference) to store the current file name with the character output mapping, default is <FlagShip_dir>/terminfo/FSchrmap.def. See section SYS and FS_SET("mapping"). The full name, including the path is returned, e.g. "/usr/local/FlagShip8/terminfo/FSchrmap.def" An empty returned value means the file was not found.

<**varC3**> is a declared and initialized character variable (called by reference), specified together with <varC2>. This is used to report the current terminal entry within the mapping file. See section SYS and FS_SET("mapping"). An empty returned value means the terminal entry <retC> was not found within the <varC2> file.

<**varC4**> is a declared and initialized character variable (called by reference) to store the current file name with the character input mapping, default is <FlagShip_dir>/terminfo/FSkeymap.def. See section SYS and FS_SET("inmap"). The full name, including the path is returned, e.g. "/usr/local/FlagShip8/terminfo/FSkeymap.def" An empty returned value means the file was not found.

<**varC5**> is a declared and initialized character variable (called by reference), specified together with <varC4>. This is used to report the current terminal entry within the mapping file. See section SYS and FS_SET("inmap"). An empty returned value means the terminal entry <retC> was not found within the <varC4> file.

Returns:

<**retC**> is a character value, representing the character string with the current setting of the environment variable TERM.

Description:

With FS_SET("term"), you can check the currently assigned terminal and optionally the used output and input mapping. The function is accepted in GUI mode too, but returns meaningful data in Terminal i/o mode only. In GUI mode, another translation tables are used, see FS_SET("guikeys").

Example:

```
local cTerm
local cFileOut := "", cTermOut := "", cFileIn := "", cTermIn := ""

cTerm := FS_SET ("term", @cFileOut, @cTermOut, @cFileIn, @cTermIn )
if AppIoMode() == "T"  // apply for terminal i/o mode only
   if empty(cTerm)
      alert("Environment TERM or FSTERM is not set")
      quit
   endif
   if empty(cFileOut)
      alert("Output mapping file (FSchrmap.def) not found")
   elseif empty(cTermOut)
      alert("Terminal '" + cTerm + "' not available in;" + ;
            cFileOut + ";incorrect screen output is possible")
   endif
   if empty(cFileIn)
      alert("Input/keyboard mapping file (FSkeymap.def) not found")
   elseif empty(cTermIn)
      alert("Terminal '" + cTerm + "' not available in;" + ;
            cFileIn + ";incorrect keyboard input is possible")
   endif
endif
```

Example:

see example in FS_SET ("outmap") and FS_SET ("inmap")

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

FS_SET("outmap"), FS_SET("inmap"), GETE(), environment variable TERM in section FSC and SYS, UNIX: man term, man terminfo

FS_SET ("translext")

Syntax:

`retC = FS_SET ("translext", expC2 [, expC3])`

Purpose:

Translates hard-coded file extensions during the create, open or file search access.

Arguments:

<**expC2**> is the hard coded extension to be replaced, e.g. "NTX". Do not enter the period.

Options:

<**expC3**> is the new file extension, which will replace the <expC2>, e.g. "idx". To delete the previous setting of <expC2>, enter null- string "". If the argument is not given, the previous setting for the same <expC2> is returned if required, NIL otherwise.

Returns:

<**retC**> is on success the new (or the current) extension substitution <expC3>, NIL otherwise.

Description:

The file extension is a part of the full file name. On DOS, the full file name consists of three parts: the main part of the file name (one to eight characters), a period (.) and an optional file extension (one to three characters). On UNIX, no special parts of the file name are used, the file name can include any number of periods. For more information refer to LNG.3.1.

To maintain compatibility with DOS programs, FlagShip supports the DOS naming convention by default on access to databases, indices and text files.

Many programs written for the DOS system have hard-coded file extensions, like FILE("ABC.NTX"). To convert all such occurrences to the recommended FILE("ABC" + INDEXEXT()) in a large application requires much valuable programmers' time. Therefore, FlagShip can translate them during the file access time automatically.

Additionally, use of the lower/upper file and path translation using FS_SET("lower") and FS_SET ("pathlower") is recommended, which also influences the translated extension.

For portability, usage of INDEXEXT() is recommended instead of hard coded extensions.

Up to five different translation rules may be given.

Example:

```
FS_SET ("transl", "ntx", "idx")    // xxx.NTX --> xxx.idx
FS_SET ("transl", "PRN", "Spool")  // xxx.PRN --> xxx.spool
FS_SET ("lower", .T.)
IF FILE("AbC.NtX")
   ? "index file abc found as " + TRUEPATH("aBc.nTx")
```

```
ENDIF
IF FILE("EfG" + INDEXEXT())
   ? "file efg found as " + TRUEPATH("eFg" + INDEXEXT())
ENDIF
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

INDEXEXT(), FS_SET("lower"), FS_SET("pathlow"), TRUEPATH()

FS_SET ("typeahead")

Syntax:

`retL = FS_SET ("typeahead" [,expL2])`

Purpose:

Determines or sets the look-ahead capability of the current terminal in use.

Options:

<**expL2**> is a logical expression. If the value is TRUE value enables the look ahead capability, if it is FALSE, it disables it. If <expL2> is not specified, the current status is returned. The default setting on program begin is FALSE.

Returns:

<**retN**> is a logical value, representing the current look ahead status. TRUE signals the currently enabled look ahead terminal status.

Description:

To interface your terminal, FlagShip uses the UNIX library "curses". This supports the required terminal settings and optimizes the current screen output.

Because the terminals are often connected using a slow serial interface, only the characters changed will be transmitted. Side effects may occur, if any key is entered during the terminal output: the output is terminated and the program seems to "hang", because the current message you are waiting for will not be displayed.

FlagShip therefore works by default without the "look type ahead" option, so curses will transmit all characters. If you are working on very slow connection (such as modem), you may enable the "look ahead" and transmit only changed characters.

Example:

```
? FS_SET ("type")                          // .F.
IF modem_is_online
   ok = FS_SET ("type", .T.)               // .T.
   ? "Don't touch keyboard during the screen output !"
ENDIF
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

section SYS, UNIX: man curses

FS_SET ("zerobyte")

Syntax:

`retL = FS_SET ("zerobyte" [,expL2])`

Purpose:

Enables or disables the support of embedded zero bytes CHR(0) in string expression and operations.

Options:

<**expL2**> is a logical expression. TRUE value enables the support of embedded zero bytes, FALSE disables it. If <expL2> is not specified, the current status is returned. The default setting at program begin is FALSE.

Returns:

<**retL**> is a logical value, representing the current zero byte support. TRUE signals the support is enabled.

Description:

In the C language, binary zero bytes, represented by CHR(0), terminate the character string by default. For special purposes, when an embedded zero byte is required (e.g. to send special escape sequences to the printer), FlagShip supports the zero byte on request. Refer to LNG.2.6.5 for more information.

The **automatic** support of embedded zero byte is built in, and therefore not affected by FS_SET("zero"):

- all assignment operators `:= = += -=`
- all comparison operators `= == != <> # $ > < >= <=`
- following string handling commands and functions: ?, ??, ALLTRIM(), ASC(), AT(), BIN2I(), BIN2L(), BIN2W(), CTOD(), DESCEND(), EMPTY(), FREAD(), I2BIN(), L2BIN(), LEFT(), LEN(), LOWER(), LTRIM(), OUTSTD(), PADx(), QOUT(), QQOUT(), RAT(), REPLICATE(), RIGHT(), RTRIM(), STRPEEK(), STUFF(), SUBSTR(), TRIM(), UPPER()

The following standard functions **optionally** support the embedded zero byte according to FS_SET("zero") setting: FREADSTR(), FREADTEXT(), STRLEN(), STRPOKE(), STRTRAN(), STRZERO(), TRANSFORM(), _parclen(), _retclen(), _storclen()

Other commands and standard functions do **not** support the embedded zero byte, i.e.: @..SAY, @..GET, DEVOUT(), DEVOUTPICT(), OUTERR(), STOD() - except for printer output where supported automatically.

Example:

```
str1 := "abc" ; str2 := "efg"
str3 := str1 + CHR(0) + str2
str4 := STRTRAN(str3, chr(0), "\0")       // note: not replaced
? str1, str2, str3, "="                   // abc efg abc?efg =
for ii := 1 to len(str3)
   ?? " " + ltrim(asc(substr(str3,ii,1))) // 97 98 99 0 101 102 103
next
? LEN(str1), LEN(str3), "=", str3         // 3  7 = abc?efg
? STRLEN(str1), STRLEN(str3), " devout(): "
devout(str3)                              // 3  3 devout(): abc
?? "  strtran():", str4                  // strtran(): abc?efg
@ row()+1,0 SAY "@..Say   : " + str3
@ row(),col()+1 SAY str4                // @..say: abc abc
? "?, Qout():",str3, str4               // Qout(): abc?efg abc?efg

? '--- with FS_SET("zero",.T.) ---'
SET ZEROBYTEOUT TO "#"
FS_SET ("zero", .T.)
str4 := STRTRAN(str3, chr(0), "\0")       // note: replaced
? LEN(str1), LEN(str3), "=", str3         // 3  7 = abc#efg
? STRLEN(str1), STRLEN(str3), " devout(): "
devout(str3)                              // 3  7 devout(): abc
?? "  strtran():", str4                  // strtran(): abc\0efg
@ row()+1,0 SAY "@..Say   : " + str3
@ row(),col()+1 SAY str4                // @..say: abc abc\0efg
? "?, Qout():",str3, str4               // Qout(): abc#efg abc\0efg
WAIT
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

see above operators, commands and functions

FWRITE ()

Syntax:

`retN = FWRITE (expN1, expC2, [expN3], [expL4])`

Purpose:

Writes the contents of a character variable to a binary file.

Arguments:

<**expN1**> is the file handle previously returned by FOPEN() or FCREATE().

<**expC2**> is the character variable to write to the specified file.

Options:

<**expN3**> is the number of bytes to write to the file, starting from the current file pointer position. If this argument is not specified, the entire contents of the character variable are written to the file. If the <expN3> value is greater then the declared variable length, only the length of the variable contents is written. If the value is less than one, nothing is written.

<**expL4**> is optional logical value. If true (.T.), FWRITE() will retry the write on failure. The default is .F. = no retry.

Returns:

<**retN**> is a numeric value, representing the number of bytes currently written to the file. If the return value is zero or less than <expN3>, then either the disk is full, the variable contents have been truncated, or an error has occurred. The error FERROR() function will return the last corresponding error code or zero if FWRITE() succeeds. Exception: when the passed <expN3> is <= 0, or the length of <expC2> is 0, zero is returned and FERROR() is set to zero. When <expN1> is not numeric or is < 0, or when <expC2> is not a string, -1 is returned and FERROR() remain unchanged.

Description:

FWRITE() is a low-level file function that writes data to an open binary file from a character string buffer. You can either write all or a portion of the variable contents.

FWRITE() writes the embedded zero bytes in <expC2> into the file, independently of the current FS_SET("zerobyte") setting. However, manipulating such a variable requires the zero-byte setting. If the binary zero support is inactive, only the FREAD() data can be written back. All other string manipulations return the shortened string part only.

Warning: FWRITE() allows a low-level access to the file system, similar to the C function write(). Therefore it should be used with care and only when you are familiar with the OS and file system.

Example:

```
LOCAL buff := SPACE(20)
handle = FCREATE("test.txt")
? FWRITE(handle, "John gets in")           // 12

FS_SET ("zerobyte", .T.)
STRPUSH (buff, 10, CHR(0))
? FWRITE(handle, buff)                     // 20
? LEN (buff)                               // 20

FS_SET ("zerobyte", .F.)
? FWRITE(handle, buff)                     // 20
? LEN (buff)                               // 10
? FSEEK (handle, 0, 1)                     // 53
? FWRITE(handle, buff, 500)                // 20
? FSEEK (handle, 0, 1)                     // 73
FCLOSE  (handle)
```

Classification:

system, file access

Related:

FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FREADTXT(), FSEEK(), FS_SET("zero")

GETACTIVE ()

Syntax:

`retO = GETACTIVE ()`

Purpose:

Determines the active GET object.

Returns:

<**retO**> is the currently active GET object within the current READ. If no READ is active, NIL is returned.

Description:

GETACTIVE() provides access to the active GET object during a READ. The current active GET object is the one with input focus at the time GETACTIVE() is called.

Example:

```
SET KEY -2 TO test
@ 1,0 GET var1 VALID test()
@ 2,0 GET var2 WHEN  test()
@ 3,0 GET var3
READ
FUNCTION test (p1, p2, p3)
LOCAL obj := GETACTIVE()
IF obj != NIL
   @ obj:ROW +10 SAY obj:NAME
   ?? " buff=",    obj:BUFFER
   ?? " changed=", obj:CHANGED
ENDIF
RETURN .T.
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Source:

The source code is available in the file <FlagShip_dir>/system/ getsys.prg

Related:

@..GET, READ, GET objects

GETALIGN ()

Syntax:

`NIL = GetAlign ([expA1])`

Purpose:

Align all @..GET columns in a passed or default GetList{} array according to the @..SAY..GET.. caption

Arguments:

<**expA1**> is an array containing Get objects. If not given, the current "GetList" variable is used.

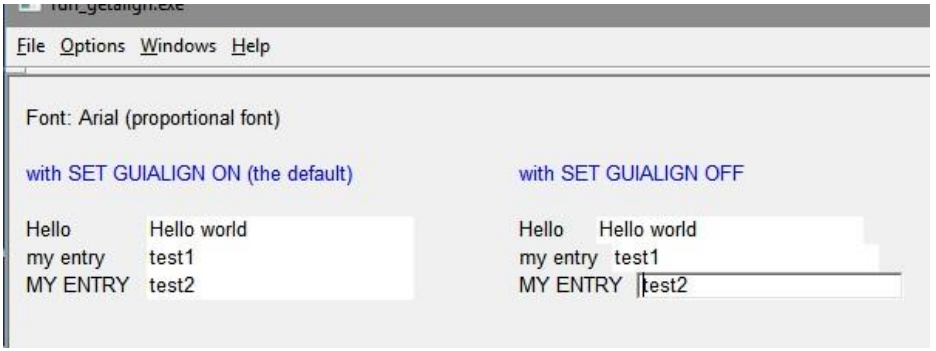
Description:

In GUI with proportional character set, the width of a text may differ even for the same string length. This would produce unpleasant formatting of the attached READ field. For example,

```
@ 10,1 say "Hello   " GET var1
@ 12,1 say "my entry" GET var2
@ 13,1 say "MY ENTRY" GET var3
READ
```

assumes to display the entry fields in column 10. It will be so displayed in Terminal i/o or in GUI mode with fixed font type (like Courier), but not in GUI with proportional font (like Arial, Helvetica or Times). To avoid heavy rework and calculating of the real width via Strlen2col(), this function aligns all GET fields to the same relative column, corresponding to the screen design.

This function is called automatically from ReadModal() == READ when SET GUIALIGN is set ON (the default). Apply for GUI mode only.



You also may invoke the GetAlign() function manually, outside of READ, without considering of the current SET GUIALIGN setting.

Classification: programming

Compatibility: New in FS5

Source: Source code is available in <FlagShip_dir>/system/getsys.prg

Related: SET GUIALIGN, @..GET, READ, GET objects

GETAPPLYKEY ()

Syntax:

NIL = GETAPPLYKEY (expO1, expN2)

Purpose:

Applies a key to a GET object from within a GET reader.

Arguments:

<**expO1**> is the active GET object.

<**expN2**> is the INKEY() value to apply to the GET object.

Returns:

The function always returns NIL.

Description:

GETAPPLYKE() is a GET system function that applies an INKEY() value to the active GET object. It simulates the key press by the user, including the cursor, function or alphanumeric keys. If the given key is assigned by SET KEY or SET FUNCTION, the key- procedure is executed in the usual way.

The FOCUS of the <expO1> object must be active (TRUE). Refer to the section OBJ.GET (GET:SETFOCUS) for more information.

Example:

Any F2 key press in the field "var2" stores A,B,C... in the entry field:

```
SET KEY -2 TO filldata
@ 1,1 GET var1
@ 2,1 GET var2
READ

PROCEDURE filldata (p1, p2, p3)
LOCAL getobj := GETACTIVE()
STATIC key := ASC("A")
IF getobj # NIL
   IF getobj:NAME == "VAR2" .and. getobj:HASFOCUS
      GETAPPLYKEY (getobj, key++)
   ENDIF
ENDIF
RETURN
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Source:

The source code is available in the file <FlagShip_dir>/system/ getsys.prg

Related:

@..GET, READ, GET objects, KEYBOARD

GETDOSETKEY ()

Syntax:

`NIL = GETDOSETKEY (expO1)`

Purpose:

Processes SET KEY during GET editing.

Arguments:

<**expO1**> is a reference to the current GET object.

Returns:

The function always returns NIL.

Description:

GETDOSETKEY() is a GET system function that executes a SET KEY code block, preserving the context of the GET object passed. The procedure name and line number are passed to the SET KEY procedure based on the most recent call to READMODAL().

Example:

see getsys.prg

Classification:

programming

Compatibility:

Available in FS and C5 only.

Source:

The source code is available in the file <FlagShip_dir>/system/ getsys.prg

Related:

@..GET, READ, GET objects, SET KEY, READMODAL()

GETENV ()

Syntax:

```
retC = GETENV (expC)
```

Purpose:

Returns the contents of an exported environmental variable.

Arguments:

<**expC**> is the name of the UNIX (or Windows) environment variable. Note that UNIX is case sensitive and that most exported variables are given in uppercase. In Windows, the environment is set by SET command, or by Start->Control Panel->System->Advanced->Environment Variables

Returns:

<**retC**> is the contents of the specified UNIX (or Windows) environmental shell variable. If the environment variable does not exists, (or in Linux/Unix: was not exported), a null string "" is returned.

Description:

GETE() or GETENV() can be used to pass environmental information from Unix/Windows into the current running executable. Typically, the current terminal, user, the environment, DOS-drive substitution etc. are retrieved. Other program configuration information, like the path names specifying the data or index directories, can be used by any user-defined (in Unix/Linux: exported) shell variable.

FlagShip for Linux/Unix requires that certain default environment variables be set:

PATH	Standard UNIX or Windows path (required)
TZ	Time zone conversion (required)
TERM	Name of the current terminal (required)
TERMINFO	Directory including the compiled terminfo description, see section FSC, SYS (optional)
LINES, **COLUMNS**	Overrides the terminfo lines# and/or cols# setting du÷ ring the curses initialization at start-up or using the FS_SET("setenv") function (optional setting)
FSOUTPUT	Directory for the standard printer output (optional)
x_FSDRIVE	Substitution of a DOS drive (x:) to a UNIX directory (optional)
SCRMAP	Directory containing the language dependent sorting tables, see FS_SET("load") function (optional)

FlagShip for MS-Windows does not need environment variables, except PATH for searching of the exe file.

The program can check the proper setting of these variables using the GETENV() function. Note also the FlagShip support of the current terminal and path translations using the FS_SET() functions.

The current environment variables may be changed using SetEnv() or FS_SET("setenv").

Example:

Determines the terminal type, checks the substitution of DOS drive C: and picks up the color settings from a file:

```
term_type = GETE("TERM")                    // case sensitive
* term_type := FS_SET ("terminal")          // equivalent
IF EMPTY(term_type)
   ? "Please set your terminal first"
   QUIT
ENDIF
IF SUBSTR(term_type,1,2) != "FS"
   ? "Your terminal is " + term_type
   ? "If you get difficulties with the screen output, " + ;
     "set your terminal to FS" + term_type
   ? "using TERM=FS" + term_type + "; EXPORT TERM"
   WAIT
ENDIF
IF EMPTY( GETENV("C_FSDRIVE"))
   ? "Set the directory for C: substitution first !"
   QUIT
ENDIF
IF FILE(term_type + ".mem")
   RESTORE FROM (term_type + ".mem") ADDITIVE
ELSE
   ? "saved screens for the terminal", ;
      term_type, "not available"
ENDIF
```

Classification:

programming

Compatibility:

Note the case sensitivity of environment variable names.

Related:

SetEnv(), FS_SET("setenv"), GetEnvArr(), section SYS and FSC

GETENVARR ()

Syntax:

```
retA = GetEnvArr ()
```

Purpose:

Returns an array of all environment variables.

Returns:

<**retA**> is 2-dimensional array containing

retA[n,1] = name of the environment variable

retA[n,2] = content of the environment variable

Description:

The functionality is similar to GetEnv() but you will retrieve all set environment variables at once.

Compatibility:

New in FS5

Related:

GETENV(), ASCAN(), FS_SET("setenv")

GETFUNCTION ()

Syntax:

```
retC = GetFunction (expN1)
```

Purpose:

Returns a string set by SET FUNCTION <expN1> TO ... or an empty string ""

Arguments:

<**expN1**> negative or 28: corresponds to Inkey() value F1 to alt-F12, i.e. K_F1 to K_ALT_F12 values in inkey.fh

<**expN1**> positive except 28: function key 1..48 same as specified in SET FUNCTION <expN> TO <string>. Note that GetFunction(28) will return assignment for F1 key, so to get assignment for Crtl-F8 key (28 in SET FUNCTION), use GetFunction (K_CTRL_F8) instead.

Returns:

<**retC**> is the returned string or ""

Compatibility:

New in FS5

Related:

SET FUNCTION, InkeyTrap()

GETPOSTVALID ()

Syntax:

```
retL = GETPOSTVALID (expO1)
```

Purpose:

Executes the post-validating expression or function.

Arguments:

<**expO1**> refers to the current GET object.

Returns:

<**retL**> is a logical value indicating whether the current GET object has been validated successfully.

Description:

GETPOSTVAL() is a GET system function that validates a GET object after editing, including evaluating get:POSTBLOCK (the VALID clause) if present.

The function is similar to the action of the VALID clause in @..GET command.

Example:

see getsys.prg

Classification:

programming

Compatibility:

Available in FS and C5 only.

Source:

The source code is available in the file <FlagShip_dir>/system/ getsys.prg

Related:

@..GET, READ, GET objects

GETPREVALID ()

Syntax:

```
retL = GETPREVALID (expO1)
```

Purpose:

Executes the pre-validating expression or function.

Arguments:

<**expO1**> refers to the current GET object.

Returns:

<**retL**> is a logical value indicating whether the current GET object has been validated successfully.

Description:

GETPREVALI() is a GET system function that validates a GET object for editing, including an evaluating get:PREBLOCK (the WHEN clause), if present.

get:EXITSTATE is also set to reflect the outcome of the pre-validation:

Code	getexit.fh	Description
0	GE_NOEXIT	pre-validation success, can be edited
8	GE_WHEN	pre-validation failure
7	GE_ESCAPE	CLEAR GETS was issued

The function is similar to the action of the WHEN clause in @..GET command.

Example:

see getsys.prg

Classification:

programming

Compatibility:

Available in FS and C5 only.

Source:

The source code is available in the file <FlagShip_dir>/system/ getsys.prg

Related:

@..GET, READ, GET objects

GETREADER ()

Syntax:

`retN = GETREADER (expO1, [expA2], [expL3])`

Purpose:

Executes standard READ behavior for one GET object.

Arguments:

<**expO1**> refers to a GET object.

<**expA2**> is GetList array containing GET objects. If not specified, the PUBLIC or PRIVATE GetList variable is used.

<**expL3**> is equivalent to CYCLE clause of READ. If not specified, .F. is the default value.

Returns:

This function returns 0 on standard processing of GET field in READ, or value 1...len(GetList) when another GET field was selected by mouse click. Other than numeric return values are interpreted as 0.

Description:

GETREADER() is normally a sub-function of the standard READ command or its READMODAL() equivalent to process editing for one GET object (field).

If the get:READER instance variable contains a code block, READMODAL() will evaluate that block instead of calling the GETREADER() function.

Example:

see getsys.prg

Classification:

programming

Compatibility:

Available in FS and C5 only, return values are new in VFS.

Source:

The source code is available in the file <FlagShip_dir>/system/ getsys.prg

Related:

@..GET, READ, GET objects

GUIDRAWLINE ()

Syntax:

NIL = GuiDrawLine ([expN1], [expN2], expN3, expN4, [expN5], [expC6], [expL7])

Purpose:

Draws a line in GUI mode (ignored in Terminal i/o) of specified width.

Arguments:

<expN1...expN4> are the start and end coordinates of the drawn line in the order of row, column, row, column. When <expN1> and <expN2> are NIL, the line drawing is continued from the current position to the <expN3>,expN4> position. If <expL7> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

<expN5> is the width of the line in pixels.

<expC6> is optional color specification. Only the foreground color is considered.

<expL7> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

Description:

The functionality is equivalent to the command

```
@ <expN1>,<expN2> [GUI] DRAW [TO] <expN3>,<expN4> [WIDTH <wpix>]
                                  [PIXEL|NOPIXEL] [COLOR <color>]
```

Compatibility:

New in FS5

Related:

@ ... DRAW ...

HARDCR ()

Syntax:

`retC = HARDCR (expC)`

Purpose:

Replaces all soft-return characters used in MEMOEDIT() with a hard return.

Arguments:

<**expC**> is a character string or memo field, which should be converted.

Returns:

<**retC**> is the converted character string.

Description:

Replaces all soft carriage returns CHR(141)+chr(10) within a string with hard carriage returns CHR(13)+chr(10), for the purpose of displaying strings edited with MEMOEDIT(). Soft carriage returns are inserted in the string by MEMOEDIT() when lines wrap.

If you don't need to archive soft carriage returns in the string, remove them by cText := strtran(cText, chr(141,10), ""), or set the global switch described in MemoEdit()

Example:

```
USE address
? HARDCR (notes)
x = MEMOEDIT (" ", 5,1, 20,79, .T.)    // edit the data
? HARDCR (x)                           // display w/o Soft-CR

x := strtran(cText, chr(141,10), "")   // remove all Soft-CR
x := strtran(cText, chr(13,10), "")    // remove all Hard-CR
? x
```

Classification:

programming

Related:

MEMOEDIT(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT(), STRTRAN(), STRPOKE()

HEADER ()

Syntax:

`retN = HEADER ()`

Purpose:

Retrieves the size of the header of a database file.

Returns:

<**retN**> is a numeric value, representing the size of the current database file's header in bytes. If no database is open in the current working area, zero is returned.

Description:

HEADER() can be used to determine the size of the database open in the current working area, along with RECSIZE() and RECCOUNT() or LASTREC(). Using DISKSPACE(), the available disk space can be checked before copying a database.

To execute the function in a working area different from the current one, use `alias->(HEADER())`.

Example:

```
USE article
file_size = HEADER() + RECCOUNT() * RECSIZE() +1
IF (file_size + 512) > DISKSPACE()
   ? "sorry, cannot copy, not enough disk space"
ELSE
   COPY FILE article.dbf TO article.sav
ENDIF
```

Classification:

database

Related:

DISKSPACE(), RECCOUNT(), RECSIZE(), oRdd:Header

HELP ()

Syntax:

NIL = HELP (expC1, expN2, expC3, expN4)

Purpose:

Display user defined help when the F1 key was depressed. The HELP procedure / function is a special case of UDF invoked by SET KEY.

Arguments:

<**expC1**> is a string representing the called procedure/function, similar to ProcName(1). Note, that some procedures and all code blocks, available by ProcName() or ProcStack(), may be filtered out, so <expC1> may not be always equivalent to ProcName(1). The filter is specified globally in `_aGlobSetting [GSET_C_SETKEYFILT]`, see *<FlagShip_dir>/system/initio.prg*, and can of course be re-defined according to your needs.

<**expN2**> is a numeric value representing the line number of the called procedure/function, similar to ProcLine(1), or 0 when the line number is not available (compiler switch -nl).

<**expC3**> is a string representing the variable name, when called from GET/READ or MENU TO, or an empty string otherwise. Corresponds to ReadVar() value.

<**expN4**> is a numeric value representing the pressed key, similar to LastKey(), or 0 or NIL if not available.

Description:

The default HELP procedure/function is triggered automatically by already assigned SET KEY K_F1 TO HELP. This allows you to override it by your own HELP procedure or function, displaying context sensitive help of your application, accessed by the default F1 key. You of course may also use any other name, when assigning it to the function key by SET KEY command.

The default HELP do nothing, except displaying standard help in MemoEdit().

Example:

```
FUNCTION HELP (cProcName, nLine, cVarName, nKey)  // user help
local cText, cStack := ProcStack()
DO CASE
CASE cProcName == "MYPROC1" .and. cVarName == "NAME"
    InfoBox("Help for editing variable NAME in procedure MyProc1")
CASE cVarName == "COUNTRY"
    InfoBox("Help for editing variable COUNTRY in any procedure")
CASE cProcName == "MYEDIT"
    InfoBox("Help for editing within procedure MyEdit")
CASE ProcName(2) == "ANYPROC"
    InfoBox("Help for editor called from procedure AnyProc")
CASE "_TMEMOEDIT" $ cStack .or. "_GMEMOEDIT" $ cStack .or. ;
     "MEMOEDITHA" $ cStack
    /* this is the default call from standard HELP, where
     * the _MemoEdHelp() is available in FlagShip library and
```

```
     * in source of <FlagShip_dir>/system/memoedithand.prg
     */
    _MemoEdHelp(cProcName, nLine, cVarName, nKey)
OTHERWISE
    cText := "sorry, no help available for this entry;in " + ;
       cProcName + if(nLine > 0, "/" + ltrim(nLine), "")
    if !empty(cVarName)
       cText += ";for entry name " + cVarName
    endif
    if FS_SET("devel")
       cText += "; ;Call-Stack: " + cStack
    endif
    InfoBox(cText)
ENDCASE
return
```

Compatibility:

Compatible to Clipper, which supports 3 arguments only and does not provide default MemoEdit help, nor ProcStack().

Related:

SET KEY, ON KEY, ProcName(), ProcLine(), ProcStack(), LastKey()

HEX2NUM ()

Syntax:

`retN = Hex2Num (expC1)`

Purpose:

Convert a string containing hexadecimal value to numeric integer.

Arguments:

<expC1> is a string containing the hex representation, e.g. "A5C0FF05" or "0x5f07De". Only characters 0..9,A..F,a..f are allowed in the string (up to 8) plus an optional prefix "0x", "0X" or "#".

Description:

Hex2num() return a numeric value corresponding to the hex string. See Num2Hex() for reverse conversion.

Example:

```
? Hex2num("4d2")                    // 1234
? Hex2num("0004d2")                 // 1234
? Hex2num("0x004D2")                // 1234
? Hex2num("#4D2")                   // 1234
```

Compatibility:

New in FS5

Related:

Num2Hex(), FS2:Hex2Str(), FS2:StrToHex()

I2BIN ()

Syntax:

`retC = I2BIN (expN)`

Purpose:

Transforms a numeric expression or variable to a 16-bit binary integer (Intel convention).

Arguments:

<**expN**> is a positive numeric value to be transformed. Decimal digits are truncated (if any). The valid range is -32768 to 32767.

Returns:

<**expC**> is a two-byte character string containing a 16-bit binary integer in Intel convention (high/low byte). On error, null-string "" is returned.

Description:

I2BIN () is used when you want to FWRITE () a short integer to a binary file in Intel format, the least significant byte first.

The inverse function of I2BIN() is BIN2I(). To convert long integers, use L2BIN().

Example:

```
#define NEWLINE CHR(10)
act_year  = YEAR (date())            && 1993
act_month = MONTH(date())            && 05
act_day   = DAY  (date())            && 15

handle = FCREATE("mytext.txt")
IF FERROR() = 0
   FWRITE (handle, "File created on:")
   FWRITE (handle, I2BIN(act_year), 1)
   FWRITE (handle, I2BIN(act_month), 1)
   FWRITE (handle, I2BIN(act_day), 1)
   FWRITE (handle, NEWLINE + "any text..." + NEWLINE)
   FCLOSE (handle)
ENDIF
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

BIN2I(), BIN2W(), BIN2L(), L2BIN(), FWRITE()

IF () | IIF ()

Syntax:

ret = IF (expL1, exp2, exp3)

or:

ret = IIF (expL1, exp2, exp3)

Purpose:

Returns one of the expressions given, dependant on the status of the logical condition.

Arguments:

<**expL1**> is the logical expression to be evaluated.

<**exp2**> is the returned expression of any type, when the <expL1> is TRUE.

<**exp3**> is the returned expression of any type, when the <expL1> is FALSE. This argument need not be the same data type as <exp2>.

Returns:

<**ret**> is the value of <exp2> or <exp3>, depending on the evaluation of the condition <expL1>.

Description:

IF () can be used to evaluate a condition within an expression or to convert a logical expression to another data type. Also, IF () can be used instead of the IF...ELSE...ENDIF structure, in cases where the clauses are short, to make the code more compact.

It can also be used as a part of an INDEX expression: if a field is empty, another field or expression can be used.

IF() or IIF() is the only function in FlagShip where the parameters are not evaluated first. Rather they are inferred from the context, depending on the result of <expL>.

Example 1:

```
x = IF (EMPTY(name), 2, 3)   |     IF empty(name)
                             |        x = 2
                             |     ELSE
                             |        x = 3
                             |     ENDIF
```

Example 2:

```
FUNCTION myfn (par1)                          && opt.param.
par1 = IF (VALTYPE(par1) != "N", 1, par1)  && set defaults
RETURN (par1 +1)
```

Classification:

programming

Related:

IF, DO CASE

INDEXCHECK ()

Syntax:

`retN = INDEXCHECK ([expN1])`

Purpose:

Checks whether the database is consistent with its associated indices.

Options:

<**expN1**> specifies the ordinal position of the index in the list of indices currently open in the current working area. Zero specifies the current controlling index. If the argument is not specified, zero is assumed.

Returns:

<**retN**> is a numeric value indicating the integrity status of the index file:

retN	Description
-1	No database or index file open
0	The integrity is correct
1	The integrity may be corrupted
2	The integrity is corrupted

Description:

In FlagShip, a unique index integrity checking is available for the standard .idx files. For more information, refer to LNG.4.5.

The INDEXCHECK() function is valuable to check the index integrity after opening a index file, and restoring the violated integrity by re-indexing the index file.

A correct database index always matches the number of key expressions stored with the database records. The only exception is a index created with the UNIQUE or FOR... clause, which usually contains fewer records than the database.

The violation of the index integrity will occur when appending new records or PACKing the database without assigning the index file by USE...INDEX or SET INDEX ON. The index integrity can be violated when replacing the contents of database fields without having activated an index file.

When INDEXCHECK() returns 1, the database update / modification count does not match the modification count of the index file. It indicates that the index file was not assigned (opened) during database modification. The index need not be corrupted, if the modified database field is not included in the index key. When in development mode, the first database movement will produce a warning message. Upon closing the index file, the modification count will be updated to confirm the count of the database.

When INDEXCHECK() returns 2, the key count in the index file does not match the record count of the database. This appears only if the database is indexed without the UNIQUE and FOR... clause. The only allowed database operations thereafter are INDEX ON, REINDEX, PACK and ZAP, which restores index integrity. Moving the

database pointer or accessing the records by any other commands will inevitably result in an unrecoverable run-time error.

To preserve index integrity, always ensure all related index files are assigned to the open database, at least during database modification. FlagShip automatically updates all assigned indices on every record modification, even if the natural index order is selected using SET ORDER TO 0.

IndexCheck() is tested automatically on any database movement (like SKIP, GOTO, GO TOP, GO BOTTOM, SEEK, LOCATE) and at CLOSE with open index/indices, and reports RTE (run-time-error) if one or more indices are corrupted. Indices created with FOR or UNIQUE condition are not tested. You may avoid this RTE by testing IndexCheck() after assigning indices - but before first database movement, and REINDEX if the test fails.

To execute the function in a different working area from the current one, use `alias->(INDEXCHECK())`.

Multiuser:

Integrity violation may also occur at run-time, if the current application has opened the database with indices, while at the same time another application modifies the same database without indices having been assigned beforehand.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Note: when the database and indices are accessed remotely via NFS or SAMBA, this may cause IndexCheck() to report failure when SET NFS is OFF. In such a case, invoke SET NFS ON, see further details there.

Example 1:

```
if getenv("ON_SERVER")
   SET NFS ON
endif
USE mydbf NEW SHARED
IF NETERR()
   ? "cannot open database"
   QUIT
ENDIF

SET INDEX TO index1, index2
IF NETERR()
   ? "cannot open index files"
   QUIT
ENDIF

IF INDEXCHECK(1) > 0 .or. INDEXCHECK(2) > 0
   USE mydbf EXCLUSIVE
   INDEX ON UPPER(name) TO index1
   INDEX ON idnumber    TO index2
   USE mydbf SHARED
   SET INDEX TO index1, index2
ENDIF
```

Example 2:

```
 /* check for and display cause of IndexCheck() error
  * (or run-time-error RTE 331) and fix corrupted indices
  */
 set font "courier",10
 USE mydbf INDEX myidx1, myidx2   // your .dbf and .idx
 errCnt := myIndexCheck(.F.)      // or (.T.) for unconditional
check

 FUNCTION myIndexCheck(lMustCheck)
 /*
  * this function is available in source in
  * <FlagShip_dir>/examples/checkindex.prg
  */
 ....
 return iError
```

Classification:

database

Compatibility:

Available in FlagShip only.

Related:

INDEX ON, REINDEX, USE, SET INDEX, oRdd:OrderInfo(DBOI_INDEXCHECK), oRdd:IndexCheck(), <FlagShip_dir>/examples/checkindex.prg

INDEXCOUNT ()

Syntax:

`retN = INDEXCOUNT ()`

Purpose:

Determines the number of indices being used in the current working area.

Returns:

<**retN**> is the number of currently used index files in the current working area. If no database or indices are open, zero is returned.

Description:

INDEXCOUNT() is a database function that determines the number of associated index files in the current working area.

To execute the function in a different working area from the current one, use `alias->(INDEXCOUNT())`.

Example:

```
USE customer ALIAS cust NEW
USE address NEW
SET INDEX TO adr1, adr2
? INDEXCOUNT()                          // 2
? cust->(INDEXCOUNT())                  // 0
```

Classification:

database

Compatibility:

Available in FS only.

Related:

SET INDEX TO, DBSETINDEX(), USE...INDEX, oRdd:IndexCount

INDEXDBF ()

Syntax:

retC = INDEXDBF ([expN1])

Purpose:

Determines the name of the associated database, which was used to create a index file.

Options:

<**expN1**> specifies the ordinal position of the index in the list of indices currently open in the current working area. Zero specifies the current controlling index. If no argument is given, zero is assumed.

Returns:

<**retC**> is the name of the database file which was active during the indexing using the INDEX ON command or the DBCREATEINDEX() function. The returned file name (up to 19 characters) includes the extension but not the path.

Description:

INDEXDBF() is a database function that reads the data from the index header of associated index file in the current working area. The header data is stored by the INDEX ON command.

To execute the function in a different working area than the current one, use `alias->(INDEXDBF())`.

Example:

```
USE address NEW EXCLUSIVE
INDEX ON UPPER(name) TO adr1
USE
//
USE mydbf NEW
SET INDEX TO adr1, adr2
? INDEXCOUNT()                              // 2
? INDEXDBF(1)                               // address.dbf
IF .not. ( INDEXDBF() $ DBF() )
   ? "wrong index file used !"
   QUIT
ENDIF
```

Classification:

database

Compatibility:

Available in FS only.

Related:

INDEX ON, SET INDEX TO, DBSETINDEX(), INDEXNAMES(), USE

INDEXEXT ()

Syntax:

`retC = INDEXEXT ()`

Purpose:

Returns the default index extension based on the database driver currently used.

Returns:

<**retC**> is in FlagShip always ".idx" to indicate the default "DBFIDX" database driver, unless another RDD is loaded.

Description:

Using INDEXEXT() instead of the hard coded index extension allows you to write fully portable programs for DOS, Windows and UNIX.

In a well-designed application, it is usual to check the ability of index files to create the indices automatically. To determine the existence of the index file, use FILE() function, see example.

Note: FlagShip supports the conversion of "hard coded" extensions to other ones using the FS_SET ("translext") function.

Example:

```
USE article
IF .NOT. FILE("adrname" + INDEXEXT())
   INDEX ON UPPER(name) TO adrname
ENDIF

FS_SET ("transl", "ntx", "idx")
IF FILE ("address.ntx")
   ? "index for address database is available"
ENDIF
```

Classification:

database

Compatibility:

FlagShip returns ".idx" with the default "DbfIdx" driver. In Clipper, ".NTX" is normally returned instead.

Related:

INDEXORD(), INDEXKEY(), FILE(), FS_SET(), oRdd:Info(), oRdd:OrderInfo()

INDEXKEY ()

Syntax:

`retC = INDEXKEY (expN)`

Purpose:

Returns the key expression of a specified index.

Arguments:

<**expN**> specifies the ordinal position of the index in the list of indices currently open in the current working area starting at one. Zero specifies the current controlling index.

Returns:

<**retC**> is the stored key expression string in the index header. If the <expN> argument is out of range, a null string "" is returned.

Description:

INDEXKEY() is a database function that determines the key expression of a specified index in the current working area and returns it as a character string. The returned expression can be evaluated as a macro (see example).

To execute the function in a different working area from the current one, use `alias->(INDEXKEY(n))`.

INDEXKEY() reads the key expression from the header of current or specified index file. With `&(INDEXKEY())` or `&(INDEXKEY(n))`, you will get the key value, determined from current database record. The real index key value, stored in the index file, is available via `DbObject():OrderInfo(DBOI_KEYVAL)`, and it should be equivalent to `&(INDEXKEY())`, otherwise the index is corrupted, and run-time-error RTE 331 is reported at access by SEEK, GOTO n + SKIP etc.

Example:

```
USE article INDEX name, idnum
? INDEXKEY(0)                           && "UPPER(Name)"
? &(INDEXKEY(0))                        && "SMITH    "
SET ORDER TO 2
? INDEXKEY(0)                           && "Id_Numb"
? INDEXKEY(1)                           && "UPPER(Name)"
? INDEXKEY(2)                           && "Id_Numb"
? INDEXKEY(3)                           && ""
? &(INDEXKEY(0))                        && 254

#include "rddsys.fh"
? DbObject():OrderInfo(DBOI_KEYVAL)     && 254
```

Classification:

database

Related:

SET INDEX, SET ORDER, USE, INDEXORD(), INDEXCHECK(), oRdd:IndexKey, oRdd:OrderInfo

INDEXNAMES ()

Syntax:

`retA = INDEXNAMES ()`

Purpose:

Determines the file names of all currently assigned indices in the current working area.

Returns:

<**retA**> is a one-dimensional array. Each character element contains the full index name, including path and extension. If no database or index is open, an empty array { } is returned.

Description:

INDEXNAMES() is a database function that determines the file names of all associated indices in the current working area. The name is returned as an array element in the order of the index assignments by SET INDEX TO.

To execute the function in a working area other than the current one, use `alias->(INDEXNAMES())`.

Example:

```
SET PATH TO /usr/data;../peter
USE adres NEW SHARED
SET INDEX TO adr1
USE mydbf NEW INDEX aa1, bb2

? adres->(INDEXNAMES())[1]          && /users/adr1.idx
xx = adres->(INDEXNAMES()); ? xx[1] && /users/adr1.idx
AEVAL (INDEXNAMES(), {|x| QOUT(x)}) && /usr/data/aa1.idx
                                    && /users/peter/bb2.idx
```

Classification:

database

Compatibility:

Available in FS only.

Related:

SET INDEX TO, DBSETINDEX(), oRdd:OrderInfo()

INDEXORD ()

Syntax:

`retN = INDEXORD ()`

Purpose:

Determines the ordinal number of the controlling index in the list of indices currently open in the selected working area.

Returns:

<**retN**> is the numeric value representing the position of the controlling index in the list of open indices in the current working area. A zero value indicates that the current database file is treated in the natural order.

Description:

INDEXORD() is a database function that determines the position of the controlling index in the list of index files opened by the last USE...INDEX or SET INDEX TO in the current working area.

This function allows a procedure to save the current index number, use a different open index and later re-use the original index.

To execute the function in a different working area from the current one, use `alias->(INDEXORD())`.

Example:

```
USE article INDEX name, idnumb
old_ord = INDEXORD()                    && store current order
? INDEXORD()                            &&   1
SET ORDER TO 2
? INDEXORD()                            &&   2
LIST name, address
SET ORDER TO (old_ord)                  && restore old order
```

Classification:

database

Related:

SET INDEX, SET ORDER, USE, INDEXKEY(), oRdd:IndexOrd()

INFOBOX ()

Syntax:

`retN = InfoBox (expC1, [expC2], [expN3])`

Purpose:

Display an infobox dialog, similar to Alert()

Arguments:

<**expC1**> is a string containing the displayed text, the lines are separated by semicolon ";" or "\n". See additional details below and in Alert() description.

<**expC2**> is an optional header text.

<**expN3**> is optional time-out in seconds. Zero or NIL specify to wait until user press key or mouse selection. When the time-out expires without user action, the box is closed and <retN> is set to 0.

Description:

The InfoBox() function is available for your convenience. It uses, and is equivalent to `InfoBox{NIL,cHeader,cMessage}:Show()`.

Alignment and Tuning:

The default alignment is left justified in GUI and centered in Terminal i/o mode. You may override these defaults by assigning

```
  _aGlobSetting[GSET_G_N_ALERT_ALIGN] := numValue  // default = 0
```

where <numValue> is

0: default alignment,
1: left justify,
2: center,
3: right justify.

In addition to, every single line can be individually aligned by three special characters at the line begin i.e. after the ";" or "\n" line separator:

```
    "<<!" align left,
    ">>!" align right,
    "><!" or "<>!" to center this line,
```

e.g.

```
   InfoBox("><!centered;<<!left;>>!right;default alignment")
```

Of course, these special marker are filtered out from the text. See example in <FlagShip_dir>/system/memoedithand.prg

If the line is too long, it is automatically wrapped. In GUI mode, you may specify the maximal text width and height (in pixel) by assigning

```
  _aGlobSetting[GSET_G_N_ALERT_WIDTH ] := nPix   // default = 0
  _aGlobSetting[GSET_G_N_ALERT_HEIGHT] := nPix   // default = 0
or <nPix> = 0:  adjust/fit box to current window size,
or <nPix> = -1: adjust/fit box to desktop size.
```

In GUI mode, the default font oApplic:FontWindow is used, except you specify your own font object by assigning

```
  _aGlobSetting[GSET_G_O_ALERT_FONT] := oFont|NIL  // def = NIL
```

In Terminal i/o mode, the displayed box and wrapping is adjusted automatically according to MaxCol() and MaxRow().

In Terminal i/o mode, the color is specified in global variable
`_aGlobSetting[GSET_T_C_INFBOXCOLOR] := "W+/BG,BG+/GR" // default`
and can be re-assigned according to your needs. In Terminal i/o mode, the border is displayed per default in plain ASCII characters (+---+) specified as B_PLAIN in box.fh. You may freely change this behavior, i.e. set the display to single or double semi-graphic border characters, by

```
_aGlobSetting[GSET_T_C_INFOBOX] := B_DOUBLE     -or-
_aGlobSetting[GSET_T_C_INFOBOX] := B_SINGLE
```

In GUI mode, the box is displayed as pop-up dialog and can be moved by mouse. In Terminal i/o, the screen area beneath the box is saved, the box displayed, and after user input the original screen area is restored. You may use pop-up dialog box in Terminal i/o mode instead, when assigning
`_aGlobSetting[GSET_L_BOXES_POPUP] := .T.    // default = .F.`
and move the pop-up box by cursor/special keys assigned to
`_aGlobSetting[GSET_A_BOXES_KEYS] := {{keyValue,orientation},..}}`
where <keyValue> is the Inkey() value and <orientation> is one char from "L,R,U,D,T,B" (Left, Right, Up, Down, TopLeft,BottomRight). Do not use K_LEFT, K_UP etc. for <keyValue>, since these keys are used for the button-selection, use Ctrl- or Alt-keys instead. Assigning an empty array to this global setting will disable InfoBox() movement. Upon needs, you may re-define the vertical and horizontal movement stepping by `_aGlobSetting[GSET_N_BOXES_ROWS|_COLS]`, see defaults in initio.prg

Example:

```
_aGlobSetting[GSET_T_C_INFBOXCOLOR] := "W+/BG,BG+/GR"  // default
_aGlobSetting[GSET_T_C_INFOBOX   ]  := B_SINGLE

InfoBox("Hello World", "Greetings")
```

Output:

Compatibility: New in VFS. The expN3 is supported since VFS7

Source:

The source code is available in <FlagShip_dir>/system/boxfunct.prg, and default settings in <FlagShip_dir>/system/initio.prg

Related:

Alert(), ErrBox(), WarnBox(), TextBox(), OBJ.InfoBox{}, ErrorBox{}, MessageBox{}, WarnBox{}, TextBox{}

INKEY ()

Syntax:

`retN = INKEY ([expN1], [expN2], [expL3], [expL4])`

Purpose:

General input function. Reads a character from the keyboard buffer, optionally waiting for key to be pressed. In GUI mode, Inkey() may report also mouse movement, press/release mouse buttons and mouse wheel action.

Options:

<**expN1**> specifies the number of seconds INKEY() will wait for a character to be typed in. The lowest valid wait value is 0.1 seconds. If zero is specified, INKEY() waits until a character has been entered.

If the argument is not specified, INKEY() does not wait for a key press.

<**expN2**> is an input mask, filtering only specified events from the buffer. The INKEY_* constants specified in include/inkey.fh are:

Mask <expN2>	Value	Description
INKEY_MOVE	1	Mouse movements
INKEY_LDOWN	2	Mouse left button down
INKEY_LUP	4	Mouse left button up
INKEY_RDOWN	8	Mouse right button down
INKEY_RUP	16	Mouse right button up
INKEY_MDOWN	32	Mouse mid button down
INKEY_MUP	64	Mouse mid button up
INKEY_BUTTONDOWN	42	Left|mid|right but down
INKEY_BUTTONUP	84	Left|mid|right but up
INKEY_BUTTON	126	all butt except move
INKEY_KEYBOARD	128	default: Keyboard event
INKEY_WHEEL	256	FS5:Mouse wheel movement
INKEY_MOUSE	383	any mouse event
INKEY_ALL	511	any mouse or keyboard
INKEY_ALL_BUT_MOVE	510	mouse + keybord w/o move
INKEY_CONTROL	512	report Ctrl-,Shift- etc

where the default is a value returned from Set(_SET_EVENTMASK). The required values can be added to each other e.g.

```
Inkey(0, INKEY_BUTTONDOWN + INKEY_KEYBOARD)
```

<**expL3**> is a logical value, .T. specifies that the input should be echoed. Default is .F., no echo

<**expL4**> is a logical value considered in GUI mode only and specifying the translation mode. Passing .F. will use SET CHARSET ISO, .T. will use SET CHARSET ASCII, and NIL or undefined will use current setting.

Returns:

<**retN**> is a numeric value in the range -47 to 1035, representing the ASCII value of the key or mouse fetched from the ahead buffer. If the buffer is empty, zero is returned.

Description:

INKEY() is a keyboard function that extracts the next key pending in the keyboard type-ahead buffer and returns a value representing that key. The last nine retrieved keyboard values are also saved internally and can be accessed using LASTKEY().

The INKEY() function with <expN> argument can be used to pause program execution, to check a termination key pressed in LIST, REPORT or other commands using the WHILE clause etc. If you wish to automatically abort waiting Inkey(n) after some time, use KeySec() or KeyTime() functions from FS2 Toolbox.

Procedures which have been assigned to keys using SET KEY...TO will not be evaluated within INKEY(), the key value is returned instead. If necessary, use DO <procname> to call these procedures.

INKEY() is similar to the NEXTKEY() function, but the NEXTKEY() does not remove the key from the keyboard buffer and does not store the value in LASTKEY().

For a list of INKEY() codes and "inkey.fh" constants, refer to the Appendix APX.

INKEY() is the central function for managing user input. It is used also by associated commands or functions like INKEYTRAP(), ACCEPT, INPUT, WAIT, READ etc. The keyboard input is in Terminal and GUI mode asynchronous, which means the user key press (or string passed by KEYBOARD command) is stored in the keyboard or event buffer and can be retrieved by the application later via Inkey() and associated commands or functions. The same is valid for mouse movement or button press in GUI mode, whereby many of the mouse (and keyboard) actions may be handled by the GUI subsystem automatically. In Basic i/o mode, the input is handled synchronously by the standard i/o system at the time of invoking Inkey(). See additional details in chapter Compatibility below, and in section LNG.5.4.4

For unconditional waiting w/o CPU load and w/o key trap, e.g. in background or CGI/Web applications, as well as for simple waiting or pause, use Sleep() or SleepMs() functions instead of Inkey(n).

Mouse:

Mouse movements, buttons and wheel are reported in GUI mode only.

Mouse movements are reported only when you enable the INKEY_MOVE or INKEY_MOUSE mask by SET EVENTMASK and/or by <expN2> of Inkey(), and are disabled by default with the INKEY_ALL_BUT_MOVE mask.

Mouse buttons and mouse wheel are enabled by default. The left, mid and right mouse buttons are reported in pressed and released status. Mouse buttons and mouse wheel status can be modified by simultaneous pressing the Shift, Control or Alt keyboard key.

The current mouse coordinate on the screen is reported by MRow() and MCol() functions.

Unicode:

In GUI mode, FlagShip supports also Unicode (UTF-8 and UTF-16). If the default font is set by `oApplic:Font:CharSet(FONT_UNICODE)` or SET GUICHARSET FONT_UNICODE and SET MULTIBYTE is ON, Inkey() process Unicode and ignores automatic conversions via SET GUITRANSL etc. See also example in *<FlagShip_dir>/examples/unicode.prg*, *memoedithand.prg, getsys.prg* and general Unicode description in section LNG.5.4.5

Performance:

You should **avoid** polling, i.e. the construct

```
while .T.             // an endless loop
  key := Inkey()      // get/check key without waiting (polling)
  if key != 0         // if any key is pressed/available,
     exit             // exit the endless loop
  endif
enddo
```

but use

```
key := INKEY(0)       // wait (sleep) until key press
```

instead. This is because the waiting INKEY(n) does not affect CPU load, whilst the first construct (non-waiting mode) heavy loads CPU through repeating the empty loop.

Tuning:

In GUI, Inkey() reacts per default on key press. You may change the behavior to consider a key at the time of releasing, by assigning

```
_aGlobSetting[GSET_G_N_KEYEVENT] := 2       // default = 1
```

The system-own mouse wheel driver often reports multiple up/down movements at single wheel action, depending on the mouse resolution. Inkey() will filter the subsequently reported movements by setting

```
_aGlobSetting[GSET_N_WHEEL_STEPS] := 4      // default
```

so here only the first of each 4 reported wheel up/down movements returns K_MOUSEWHEELUP or K_MOUSEWHEELDN, the subsequent three movements are returned as K_MOUSEWHEEL. You may check it by e.g. the *<FlagShip_dir>/examples/keyboard.prg* program. You may disable reporting mouse wheel by Inkey() at all by

```
_aGlobSetting[GSET_L_ACCEPT_WHEEL] := .F.   // default = .T.
```

or by setting the corresponding filter mask.

Example:

ESC terminates the listing, any other key pauses the output:

```
USE authors
LIST lastname, firstname WHILE myPause()
USE
RETURN

FUNCTION myPause
LOCAL key := INKEY ()
IF key = 0
   RETURN .T.
ELSEIF key = K_ESC                      // 27
   ? "*** aborted by user ***"
   RETURN .F.
ENDIF
? "press any key..."; INKEY(0)          // or simply: WAIT
RETURN .T.
```

Classification:

system request, keyboard and mouse input driver

Compatibility:

In Terminal i/o mode, the INKEY() functionality is only properly supported, if either fgsInitCurses() or fgsloctl2() is initialized (which is the default with activated Curses, see the <*FlagShip_dir>/ system/cursinit.prg file*). If both are disabled, the driver is in raw, waiting mode awaiting a key press and <return>.

In GUI mode, INKEY() functionality is always properly initialized at startup of the application, mouse is supported.

In Basic mode, INKEY(n) is simulated by standard C functions ioctl() and read(stdin) in Unix/Linux and _kbhit() + _getch() in Windows. If SET INPUT is OFF, input is ignored and only the _SET_NOINPUTCAHR value is returned.

The 2nd and 3rd parameters are new in FS5, same as mouse events

Unicode is supported in VFS7 and later.

Include:

The #include-able key manifests and constants are available in "inkey.fh", located in <FlagShip_dir>/include directory.

Related:

InkeyStatus(), InkeyTrap(), Inkey2str(), KEYBOARD, LASTKEY(), NEXTKEY(), SET KEY, SET TYPEAHEAD, inkey.fh, SLEEP(), SLEEPMS(), SYS.2.7

INKEYSTATUS ()

Syntax 1:

```
retL = InkeyStatus (expN1, [expL2] )
```

Syntax 2:

```
retN = InkeyStatus ( [expL2] )
```

Purpose:

Returns status of toggled and special keys

Arguments:

<**expN1**> is the requested status flag, as defined in inkey.fh

status <expN1>	Description	Type	at start	set()
STATUS_NUMLOCK	NumLock status	toggle	on/off	-
STATUS_CAPSLOCK	CapsLock status	toggle	on/off	-
STATUS_SCROLL	ScrollLock status	toggle	on/off	-
STATUS_INSERT	Insert status	toggle	off	yes
STATUS_SHIFT	Shift key status	on/off	off	-
STATUS_CTRL	CTRL key pressed	on/off	off	-
STATUS_ALT	ALT key pressed	on/off	off	-
STATUS_ALTGR	CTRL + ALT pressed	on/off	off	-
STATUS_MENU	Menu key status	on/off	off	-
STATUS_LWINDOW	Left window key	on/off	off	-
STATUS_RWINDOW	Right window key	on/off	off	-
STATUS_META	Meta key	on/off	off	-
STATUS_ANYKEY	all/any of above keys			
STATUS_LEFTDOWN	Mouse: Left button	on/off	off	-
STATUS_MIDDOWN	Mouse: Mid button	on/off	off	-
STATUS_RIGHTDOWN	Mouse: Right button	on/off	off	-
STATUS_WHEEL	Mouse: Wheel	on/off	off	-
STATUS_MOUSESHIFT	Mouse: Shift + button	on/off	off	-
STATUS_MOUSEALT	Mouse: Alt + button	on/off	off	-
STATUS_MOUSECTRL	Mouse: Ctrl + button	on/off	off	-
STATUS_MOUSEDOUBLE	Mouse: Double click	on/off	off	-
STATUS_ANYMOUSE	all/any of above mouse			

The status flags may be binary or-ed or added to each other.

Option:

<**expL2**> .T. value triggers to re-read LED status keys

Returns:

<**retL**> is .T. if the requested flag (or all given flags) is/are set
<**retN**> are all current flags or-ed (or added) to each other

Description:

With InkeyStatus() you may determine press of special keyboard keys or mouse button status, reported at the time of last invocation of Inkey(), InkeyTrap(), READ, WAIT etc. Some of these flags are available also by evaluating the returned Inkey() value, some are not available otherwise.

The LED lighted toggle keys (NumLock, CapsLock, ScrollLock) are determined at program start and are toggled on/off at key or mouse button press - but not when Inkey() returns 0, since the detection may be slightly time expensive (approx 1ms in X11). If you wish to determine these LED flags also without waiting for a key, invoke InkeyStatus(.T.) or InkeyStatus(STATUS_*, .T.)

All other switches reports the status of last Inkey() key press within current application. The flags are set by corresponding key press and reset by other keys within Inkey() or InkeyTrap() or other input functions. Neither KEYBOARD nor LastKey() or NextKey() affect the status.

Changing of the STATUS_INSERT flag by key press outside of GET/READ and MemoEdit() is allowed only when SET(_SET_INSERT_INKEY) is .T., the default is .F. You may set/clear the Insert mode in program by using SET(_SET_INSERT) or ReadInsert() functions.

The mouse button flags are set in GUI mode only. Similar to the status of keyboard flags, the flag is set when the left or mid or right mouse button is pressed and remain set until another mouse button was pressed. When also the Shift or Ctrl or Alt key was pressed, the corresponding modifier flag is set as well.

Implementation Note:

Determining the LED status keys in **Terminal i/o mode in Unix/Linux** depends on the used environment (X11 or plain console). To avoid linking the X11 library (which is sometimes not installed on servers or on basic systems), the default behavior of InkeyStatus() available in the FlagShip library considers plain console only, i.e. it returns .F. in application compiled and/or executed by -io=t switch in X11 environment (like KDE or Gnome console) - even if the LED lights; this is because these keys needs to be determined via X11 in such a case.

To enable this feature also for non-GUI applications running in X11 environment, simply compile the <FlagShip_dir>/system/inkeystatapi.c and add the object to your application, see details in header of the source file.

The LED status keys/lights are available per default in the FlagShip library (w/o recompiling) for application running in GUI mode, or for Terminal i/o applications running on plain textual console w/o X11, or in FlagShip for MS-Windows.

Example 1:

```
? "any key : "
key := inkey(0)   // wait for key
? "key pressed =", ltrim(key), "=", inkey2str(key)

? "Insert status is " + if(InkeyStatus(STATUS_INSERT), "ON","OFF")
SET(_SET_INSERT, .T.)  // or ReadInsert(.T.)
? "Insert status is " + if(InkeyStatus(STATUS_INSERT), "ON","OFF")
```

```
if InkeyStatus(STATUS_CAPSLOCK, .T.)
   ? "CapsLock is ON"
endif

if InkeyStatus(STATUS_CTRL + STATUS_ALT)
   ? "Ctrl and Alt key (or AltGr) pressed"
endif
if BinAnd(InkeyStatus(), STATUS_CTRL + STATUS_ALT) > 0
   ? "Either Ctrl or Alt or both or AltGr pressed"
endif
```

Example 2:

```
key := 0
while key != K_ESC
   ? "Press any key (or ESC) : "
   key := inkey(2)
   ?? ltrim(key), "=", inkey2str(key)
   ? "- Status:", GetInkeyFlags()
enddo
wait

FUNCTION GetInkeyFlags()
local cTxt := ""
if InkeyStatus(.T.) > 0
  cTxt += if(InkeyStatus(STATUS_INSERT),     "Insert ",    "")
  cTxt += if(InkeyStatus(STATUS_CAPSLOCK),   "CapsLock ",  "")
  cTxt += if(InkeyStatus(STATUS_NUMLOCK),    "NumLock ",   "")
  cTxt += if(InkeyStatus(STATUS_SCROLL),     "ScrollLock ","")
  cTxt += if(InkeyStatus(STATUS_SHIFT),      "Shift ",     "")
  cTxt += if(InkeyStatus(STATUS_CTRL),       "Ctrl ",      "")
  cTxt += if(InkeyStatus(STATUS_ALT),        "Alt ",       "")
  cTxt += if(InkeyStatus(STATUS_LWINDOW),    "LeftWin ",   "")
  cTxt += if(InkeyStatus(STATUS_RWINDOW),    "RightWin ",  "")
  cTxt += if(InkeyStatus(STATUS_MENU),       "Menu ",      "")
  cTxt += if(InkeyStatus(STATUS_META),       "Meta ",      "")
endif
if BinAnd(InkeyStatus(), STATUS_ANYMOUSE) > 0
  cTxt += if(InkeyStatus(STATUS_MOUSEDOUBLE),"Double-",    "")
  cTxt += if(InkeyStatus(STATUS_LEFTDOWN),   "MouseLeft",  "")
  cTxt += if(InkeyStatus(STATUS_MIDDOWN),    "MouseMid",   "")
  cTxt += if(InkeyStatus(STATUS_RIGHTDOWN),  "MouseRight", "")
  cTxt += if(InkeyStatus(STATUS_WHEEL),      "MouseWheel", "")
  cTxt += if(InkeyStatus(STATUS_MOUSESHIFT), "+Shift",     "")
  cTxt += if(InkeyStatus(STATUS_MOUSECTRL),  "+Ctrl",      "")
  cTxt += if(InkeyStatus(STATUS_MOUSEALT),   "+Alt",       "")
endif
return trim(cTxt)
```

Compatibility:

New in FS6

Related:

Inkey(), InkeyTrap(), Inkey2str(), SET(_SET_INSERT), inkey.fh, inkeystatapi.c

INKEYTRAP ()

Syntax:

```
retN = InkeyTrap ([expN1], [expN2], [expL3],
                  [expL4], [expL5])
```

Purpose:

Reads a character from the keyboard buffer, optionally waiting for key to be pressed. Consider the SET KEY, SET FILTER and ON KEY trapping.

Options:

<**expN1**> specifies the number of seconds INKEY() will wait for a character to be typed in. The lowest valid wait value is 0.1 seconds. If zero is specified, InkeyTrap() waits until a character has been entered. If the argument is not specified, InkeyTrap() does not wait for a key press.

<**expN2**> is an input mask, filtering only specified events from the buffer. See INKEY_* constants in include/inkey.fh, default is a value returned from SET(_SET_EVENTMASK)

<**expL3**> is a logical value, .T. specifies that the input should be echoed. Default is .F., no echo

<**expL4**> is optional logical value or NIL, specifying how to continue after processing of keys assigned by SET KEY or ON KEY:

- when true (.T.), InkeyTrap() executes the SET KEY or ON KEY function or codeblock and waits for the next key press, when <expN1> is present. This is similar to RETRY statement in FoxPro. An UDF assigned by ON ANY KEY is processed by the same way, but InkeyTrap() terminates thereafter returning the pressed key, to avoid an infinite loop.
- when false (.F.), InkeyTrap() executes the SET KEY, ON KEY or ON ANY KEY function or codeblock and terminates, returning the pressed key.
- Otherwise, when a key assigned via SET KEY is pressed, the UDF is executed and InkeyTrap() waits for the next key input when <expN1> was specified. When a key assigned by ON KEY or ON ANY KEY is pressed, the assigned UDF or code block is executed and InkeyTrap() terminates, returning the pressed key. This is similar to the behavior of WAIT command.

<**expL5**> is optional logical value or NIL, specifying how to handle a key assigned by SET FUNCTION. If .T. or NIL, it fills the keyboard buffer and returns its first character. Note that the same key assigned by SET KEY, ON KEY or ON ANY KEY has higher preference than the SET FUNCTION key.

Returns:

<**retN**> is a numeric value in the range -47 to 1035, representing the ASCII value of the key or mouse fetched from the ahead buffer. If the buffer is empty, zero is returned.

Description:

InkeyTrap() is equivalent to Inkey() but it automatically executes SET KEY, SET FUNCTION or ON KEY actions, not processed by Inkey().

Compatibility:

New in FS5

Related:

Inkey(), Inkey2str(), inkey.fh

INKEY2READ ()

Syntax:

```
retL = Inkey2Read (expC1)
```

Purpose:

Provides own Inkey2str() text values

Arguments:

<**expC1**> is the file name containing the Inkey2str() text

Returns:

<**retL**> is .T. on success, .F. on failure

Description:

With Inkey2Read(), you may provide your own text for Inkey2str()

Compatibility:

New in FS5

Related:

Inkey(), InkeyTrap(), Inkey2str(), inkey.fh

INKEY2STR ()

Syntax:

```
retC = Inkey2str (expN1)
```

Purpose:

Translates inkey value to human readable string

Arguments:

<**expN1**> is the INKEY() return value to be converted.

Returns:

<**retC**> is a character string explaining the passed inkey number.

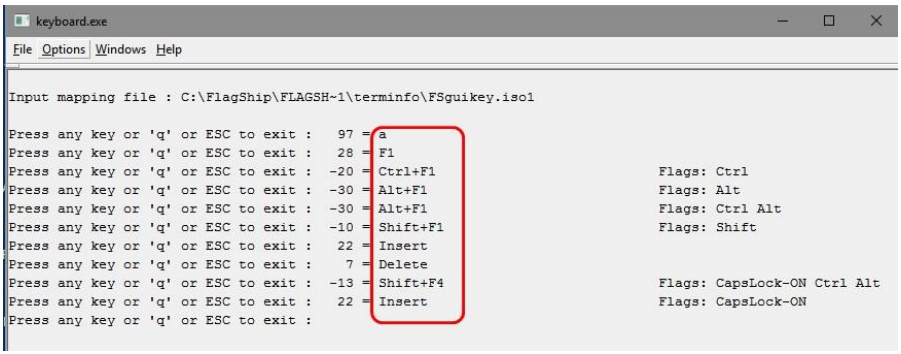
Description:

Inkey2str() tries to translate the passed number to human readable text. If no equivalence was found, empty string "" is returned. You may provide your own text by Inkey2read()

Example:

see *<FlagShip_dir>/examples/keyboard.prg* for complete example

Output:



Compatibility:

New in FS5

Related:

Inkey(), InkeyTrap(), Inkey2read(), inkey.fh

INSTDCHAR ()

Syntax:

retC = InStdChar ([expN1], [expL2])

Purpose:

Wait for one char (w/o RETURN) from stdin, same as non-canonic read(0) in C

Arguments:

<**expN1**> waiting period in seconds. If NIL, non-waiting read

<**expL2**> output echo if .T., default is .F. w/o echo

Returns:

<**retC**> is a single character entered or "" if none.

Description:

InStdChar() read directly from the device and is designed mainly for basic mode (-io=b) and Web/CGI applications.

Compatibility:

New in FS5

Related:

InStdString(), Inkey(), InkeyTrap(), Fread()

INSTDSTRING ()

Syntax:

`retC = InStdString ([expN1])`

Purpose:

Wait for string + RETURN from stdin, same as fgets(stdin) in C canonic with LF & echo.

Arguments:

<**expN1**> max number of characters to read. If not given, the whole string is returned.

Returns:

<**retC**> is the entered string.

Description:

InStdString() read directly from the device and is designed mainly for basic mode (-io=b) and Web/CGI applications. See also FreadStdIn() for alternative input

Compatibility:

New in FS5

Related:

InStdChar(), FreadStdIn(), Inkey(), InkeyTrap()

INT ()

Syntax:

`retN = INT (expN)`

Purpose:

Chops decimal digits off the result of a numeric expression, thus converting it to an integer value.

Arguments:

<**expN**> is the numeric expression to convert.

Returns:

<**retN**> is a numeric value, representing the integer part of the argument.

Description:

INT () is used when a number containing decimals has to be used as an integer by truncating (not rounding) all digits to the right of the decimal point. To round a decimal number up and down, use ROUND() instead.

When the <expN> exceeds the integer range of +-2*10**9, this Int() function automatically switches to double and returns numeric value with truncated decimals (precision >= +-9*10**14). Alternatively, you may use Round(num,0) to round instead of truncate the decimals.

Note that all mathematical operations are performed using floating point arithmetic (and coprocessor, if available). Due to internal HW storage and rounding of (double) floating numbers, you may get different results when truncating the integer part of a complex expression from e.g. simple numeric constants. See also section LNG.2.9.

Example:

Centers the displayed message on the screen:

```
center (10, "my text"
center (11, "this is a long text")

FUNCTION center (line, text)
LOCAL col := INT((MAXCOL() - LEN(text)) / 2)
@ line, col SAY text
RETURN NIL
```

Classification:

programming

Related:

ROUND(), STR(), ASC(), INT2NUM(), NUM2INT(), FS2:Ceiling(), FS2:Floor(), FS2:Sign(), FS2:IntPos(), FS2:IntNeg()

INT2NUM ()
NUM2INT ()

Syntax 1:

```
retN = INT2NUM (expN1|expI1)
```

Syntax 2:

```
retI = NUM2INT (expN1)
```

Purpose:

Converts a typed INTVAR variable to NUMERIC type (syntax 1) or a NUMERIC type to typed INTVAR variable (syntax 2).

Arguments:

<**expN1**> is a typed NUMERIC or a usual untyped numeric variable, containing a floating point value.

<**expI1**> is a typed INTVAR variable, containing a (long) integer (whole number) value.

Returns:

<**retN**> is the value of <exp1>, converted to a usual NUMERIC (floating) type.

<**retI**> is the value of <expN1>, rounded to a (long) integer value (the result is of INTVAR type).

Description:

INT2NUM() directly converts an INTVAR variable to the NUMERIC type. It is equivalent to an assignment varNUM := varINT

NUM2INT() directly converts and **rounds** a NUMERIC variable to the NUMERIC type. It is equivalent to calling ROUND(expN1,0), but avoids the double type conversion (since ROUND() returns a numeric type). An assignment varINT := varNUM is very similar, but does **not** perform the rounding.

Example:

```
Local iVar := 987654321, iTest AS INTVAR
Local nVar := 1234.6578, nTest AS NUMERIC

? iTest := nVar             // 1234
? iTest := NUM2INT(nVar)    // 1235
? nTest := iVar             // 987654320.999999999  (*)
? iTest := NUM2INT(iTest)   // 987654321
```

Classification: programming
Compatibility: available in FS only
Related:

LOCAL...AS, FS_SET("intvar"), FS2:Ceiling(), FS2:Floor(), FS2:Sign(), FS2:Exponent()

ISALPHA ()

Syntax:

retL = ISALPHA (expC1, [expL2])

Purpose:

Determines if the specified character string begins with an alphabetic character.

Arguments:

<**expC1**> is the character string to test.

Options:

<**expL2**> specifies, if the standard ASCII character set or the loadable language specification (e.g. via fs_set("load", 1, "FSsortab.jap") should be checked.

If specified TRUE, **and** a language table is loaded via fs_set("load"...) + fs_set("setlang"...), the function returns TRUE for all Alpha characters specified there, e.g. upper/lowercase A..Z, Ž, ™, š etc.

Otherwise, or if not given or specified FALSE, the function returns TRUE only if the first character of <expC**1**> is [A...Z] or [a...z].

Returns:

<**retL**> is TRUE if the first character of <expC> is alphabetic, or FALSE if it is not. An alphabetic character is a lowercase or an uppercase character from A to Z, or according to user defined character set, specified e.g. in FSsortab.xxx (see SYS.2.6).

Example:

```
? ISALPHA("A12")                          && .T.
? ISALPHA("1AB")                          && .F.
? ISALPHA("...")                          && .F.
```

Classification:

programming

Compatibility:

International languages are supported through FS_SET("load") and FS_SET("set") functions. This, and the second parameter is available in FlagShip only.

Related:

ISLOWER(), ISUPPER(), LOWER(), FS_SET(), UPPER(), FS_SET()

ISBEGSEQ ()

Syntax:

`retL = ISBEGSEQ ( )`

Purpose:

Determines if a safe BREAK to the next END SEQUENCE or RECOVER may be executed, i.e. if BEGIN SEQUENCE ... END is currently active.

Returns:

<**retL**> is TRUE if the BREAK statement can be executed and will jump to the next RECOVER or END SEQUENCE statement. FALSE signals, that the BEGIN SEQUENCE ... END is not active, and the BREAK statement will cause a run-time error.

Example:

```
openData ("first")
do while lastkey() != 27
   BEGIN SEQUENCE
      iKey := achoice (5,10,7,20,;
                       {"List", "Create", "Modify"})
      DO CASE
      CASE iKey == 1
         openData ("second")
         listData ()
      ENDCASE
      USE
   RECOVER
      ?? " (press any key or ESC)"
      inkey (0)
      @ 20,0
   END SEQUENCE
enddo

FUNCTION openData(inFile)
USE (inFile) NEW
if !used()
   @ 20,0 "Cannot open database " + inFile + ".dbf"
   if ISBEGSEQ()
      ?? " ... try later"
      BREAK
   else
      ?? " ... sorry"
      QUIT
   endif
endif
return NIL
```

Classification: programming
Compatibility: available in FS only
Related:

BEGIN SEQUENCE, BREAK, RECOVER, BREAK()

ISCOLOR ()

Syntax:

retL = ISCOLOR ()

or:

retL = ISCOLOUR ()

Purpose:

Determines if the program is using a color monitor (and the color terminfo description).

Returns:

<**retL**> is TRUE if a color monitor and the associated terminfo description is used, or FALSE if it is not.

Description:

Using ISCOLOR(), you can decide which color settings to use, those for a monochrome monitor, or those for a color monitor.

Example:

```
color1 = "BG/B,W/N"
color2 = "W/N,N+/W"
actcolor = IF (ISCOLOR(), color1, color2)
SET COLOR TO &actcolor                  // or: ...TO (actcolor)
```

Classification:

system, terminfo and environment vars

Compatibility:

On UNIX, the color capability will be determined from the terminfo entry "colors" and "pairs" of the current terminal (environment variable) TERM.

Related:

SET COLOR, SETCOLOR()

ISDBEXCL ()

Syntax:

`retL = ISDBEXCL ()`

Purpose:

Determines if the database is open in exclusive mode.

Returns:

<**retL**> is TRUE if the database in the current working area is open in exclusive mode, FALSE if it is not.

Description:

ISDBEXCL() is a database function that determines how the database in the current working area was opened. A TRUE return value signals that the USE...EXCLUSIVE or SET EXCLUSIVE ON (the default) command was used.

The function is useful to determine, if subsequent global database or index changes have been successful.

To execute the function in a different working area from the current one, use `alias->(ISDBEXCL())`.

Multiuser:

To open a database and its associated files in a multiuser and multitasking mode, use the SHARED clause of the USE command or set the global status SET EXCLUSIVE OFF. Once the database is open, the state cannot be changed until the database is closed. For more information refer to LNG.4.8.

Example:

```
LOCAL share := .F.                        // default: exclusive
PARAMETERS cmdpar                         // get command line
IF PCOUNT() > 0
   share := "/NET" $ UPPER(cmdpar)
ENDIF
IF share
   USE address NEW SHARED
ELSE
   ELSE address NEW EXCLUSIVE
ENDIF
IF NETERR()
   ? "cannot open address.dbf"
   QUIT
ENDIF
//
DO indexadr
PROCEDURE indexadr
SELECT address
IF !FILE("adr1" + INDEXEXT())             // index file exists ?
   IF ! ISDBEXCL()                        // no, create index
      WHILE ! FLOCK()                     // at least FLOCK()
      ENDDO()                             // required
   ENDIF
```

```
   INDEX ON UPPER(name) TO adr1
   IF ! ISDBEXCL()
      UNLOCK
   ENDIF
ENDIF
SET INDEX TO adr1
RETURN
```

Classification:

database

Compatibility:

Available in FS only.

Related:

ISDBFLOCK(), ISDBRLOCK(), USE, SET EXCLUSIVE, oRdd:Shared

ISDBFLOCK ()

Syntax:

`retL = ISDBFLOCK ()`

Purpose:

Determines if the entire database in the current working area is locked using FLOCK().

Returns:

<**retL**> is TRUE if the current database is already locked using file lock FLOCK(), FALSE if it is not.

Description:

ISDBFLOCK() is a database function that determines the locking state of the database in the current working area. A TRUE return value signals that the entire database is already locked by the current user and the subsequent REPLACE has been successful. In contrast to FLOCK(), ISDBFLOCK() will not lock the database itself.

To execute the function in a different working area than the current one, use `alias->(ISDBFLOCK())`.

Multiuser:

The file locking may be activated in shareable mode only. If the database has already been EXCLUSIVE-ly opened by the current user, ISDBFLOCK() returns TRUE.

Example:

```
USE address NEW SHARED
IF NETERR()
   ? "cannot open address.dbf"
   QUIT
ENDIF
IF ! ISDBFLOCK()                              // check status
   IF ! FLOCK()                               // try to lock
      ? "cannot lock a file, try later..."
      RETURN
   ENDIF
ENDIF
RECALL ALL
```

Classification:

database

Compatibility:

Available in FS only.

Related:

FLOCK(), ISDBRLOCK(), ISDBEXCL(), USE, SET EXCLUSIVE, oRdd:Info()

ISDBRLOCK ()

Syntax:

retL = ISDBRLOCK ([expN1])

Purpose:

Determines if the database record in the current working area is locked.

Options:

<**expN1**> specifies the record number, of which only the RLOCK() should be determined. If omitted, the general lock state of the current record is returned.

Returns:

<**retL**> is TRUE if the current database record is already locked by RLOCK(), FLOCK(), DBAPPEND() or APPEND BLANK, FALSE if it is not.

Description:

ISDBRLOCK() is a database function that determines the locking state of the database in the current working area. A TRUE return value signals that the database record has been already locked by the current user and the subsequent REPLACE has been successful. As opposed to RLOCK(), ISDBRLOCK() does not lock the database itself.

To execute the function in a different working area than the current one, use `alias->(ISDBRLOCK())`.

Multiuser:

The record locking may be activated in shareable mode only. If the database has been EXCLUSIVE-ly opened by the current user, ISDBRLOCK() returns TRUE.

Example:

```
USE address NEW SHARED
IF NETERR()
   ? "cannot open address.dbf"
   QUIT
ENDIF
IF ! ISDBRLOCK()                              // check status
   IF ! RLOCK()                               // try to lock
      ? "cannot lock the record, try later..."
      RETURN
   ENDIF
ENDIF
REPLACE name WITH "Smith"
```

Classification:

database

Compatibility:

Available in FS only.

Related:

RLOCK(), ISDBFLOCK(), ISDBEXCL(), USE, SET MULTILOCKS, oRdd:RlockList

ISDBMULTIPLE ()

Syntax:

```
retL = IsDbMultipleOpen ()
retL = IsDbMultip ()
```

Purpose:

Determines if the database in the current working area is multiple (concurrently) open within the same application.

Returns:

<**retL**> is TRUE if the current database is concurrently open in another work area within the same application, see also LNG.4.3.

Description:

IsDbMultip() is a database function that determines the concurrent use of the same database in different work areas at a time. To determine whether the current database is open by another user or process, use the UsersDbf() function instead.

Example:

```
USE address ALIAS adr1 NEW SHARED
? used(), select(), IsDbMultiple(), RecNo()  // .T. 1 .F. 1

USE address ALIAS adr2 NEW SHARED
GO BOTTOM
? used(), select(), IsDbMultiple(), RecNo()  // .T. 2 .T. 54321

SELECT adr1
? used(), select(), IsDbMultiple(), RecNo()  // .T. 1 .T. 1
USE
? used(), select(), IsDbMultiple(), RecNo()  // .F. 1 .F. 0

SELECT adr2
? used(), select(), IsDbMultiple(), RecNo()  // .T. 2 .F. 54321
```

Classification:

database

Compatibility:

Available in FS6 only.

Related:

USERSDBF(), DBFINFO()

ISDIGIT ()

Syntax:

retL = ISDIGIT (expC)

Purpose:

Determines if leftmost character in a character string is a digit.

Arguments:

<**expC**> is the character string to test.

Returns:

<**retL**> is TRUE if the first character of <expC> is a digit between zero and nine.

Example:

```
? ISDIGIT("a12")                          && .F.
? ISDIGIT("12a")                          && .T.
? ISDIGIT(".123")                         && .F.
```

Classification:

programming

Related:

ISALPHA(), ISUPPER(), ISLOWER(), LOWER(), UPPER()

ISFUNCTION ()

Syntax:

`retL = ISFUNCTION (expC1)`

Purpose:

Determines if a standard or user defined function is available (i.e. linked to the application).

Arguments:

<**expC1**> specifies the name of the required standard function or the name of a user defined procedure/ function.

Returns:

<**retL**> is TRUE if the specified procedure or function is available to the current application.

Description:

ISFUNCTION() determines, if the specified public procedure/function was linked to the current application. It is similar to the TYPE() function, but unlike these ISFUNCTION() does not evaluate (execute) the UDF. Since STATIC UDFs and UDPs are already resolved at compile-time, their names cannot be determined by ISFUNCTION() at run-time.

If a standard or user defined function or procedure is to be invoked by a macro evaluation only, and not called by name elsewhere in the program, use the EXTERNAL or REQUEST command to ensure that it will be linked. Otherwise, a run-time error may occur. Note, both a UDF used in an index key and the user defined procedures and functions for ACHOICE(), MEMOEDIT(), DBEDIT() are invoked by macro evaluation.

Example:

```
EXTERNAL myudf2                          // ensure to link it
IF ISFUNCTION("myudf1")                  // is other one linked ?
   name = "myudf1"                       // yes, use it
ELSE
   name = "myudf2"
ENDIF
DO &name WITH 1, "text"
```

Classification:

programming

Compatibility:

Available in FS only.

Related:

TYPE(), ISOBJPROPERTY(), ISOBJCLASS()

ISGUIMODE ()

Syntax:

`retL = IsGuiMode ( )`

Purpose:

Check if GUI mode is generally available

Returns:

<**retL**> is .T. if X11 (or MS-Windows) is available and running. It returns .F. when X11 was yet not started (startx).

Description:

This function determines the mode of X11 server or MS-Windows. To test whether the application is really running in GUI mode (which depends on the compiler and/or command-line -io=? switch), use AppIoMode() instead, where AppIoMode() returns "G" when the application was linked/started with -io=a or -io=g switch.

Example:

```
static aMode := {{"G","GUI"},{"T","Terminal"},{"B","Basic"}}
local  cMode := aMode[ascan(aMode, {|x| x[1] == AppIoMode()}), 2]
if IsGuiMode()
   ? "The application is able to run in GUI mode"
   if AppIoMode() == "G"
      ?? " and runs so"
   else
      ? "but is compiled/was invoked in " + cMode + " i/o mode"
   endif
else
   ? "X11 server is not active, using " + cMode + " i/o mode"
endif
```

Compatibility:

New in FS5

Related:

AppIoMode(), FSC -io=? switch

ISLOWER ()

Syntax:

retL = ISLOWER (expC)

Purpose:

Determines if the first character of the specified string is lowercase.

Arguments:

<**expC**> is the character string to test.

Returns:

<**retL**> is TRUE if the first character of <expC> is lowercase, or FALSE if it is not. If the first character is not alphabetic (A-Z, a-z), FALSE is returned.

Example:

```
? ISLOWER("a12")                    && .T.
? ISLOWER("A12")                    && .F.
```

Classification:

programming

Compatibility:

International languages are supported through FS_SET("load") and FS_SET("set") functions.

Related:

ISUPPER(), ISALPHA(), ISDIGIT(), LOWER(), UPPER(), FS_SET()

ISOBJCLASS ()

Syntax 1:

retA = ISOBJCLASS (expO1)

Syntax 2:

retL = ISOBJCLASS (expO1, expC2)

Purpose:

Determines the class name of an object variable, as well as names of its super classes, if any.

Arguments:

<**expO1**> is an already instantiated object variable.

<**expC2**> class name (or alias) to be tested. The case is not significant.

Returns:

<**retA**> is an array containing name(s) of the current and all inherited classes. The element retA[1] contains the name of SELF. If the class is inherited from other classes, the element retA[2] contains the name of the SUPER class. Elements following contain names of inherited classes above the SUPER class, if available. On error, i.e. if the <expO1> was not instantiated yet, an empty (zero length) array is returned.

<**retL**> is .T. if the object <expO1> is of class <expC2> or is inherited from class <expC2>, or the class of <expO1> is aliased by <expC2>. Otherwise .F. is returned.

Description:

ISOBJCLASS() in syntax 1 determines the class hierarchy at run- time. It returns the current SELF class name and all inheritances, if any.

ISOBJCLASS() in syntax 2 checks if the object is of class name <expC2> or is inherited from class <expC2>.

Example:

```
 CLASS myClass1 ALIAS test1
   EXPORT  var1

 CLASS myClass2 INHERIT myClass1 ALIAS test2
   EXPORT  var3

 oMyObj1 := myClass1 {}                // instantiate basic class
 aeval (IsObjClass(oMyObj1), {|x| qout(x)})   // MYCLASS1

 oMyObj2 := myClass2 {}                // instantiate derivated
class
 aClassName := IsObjClass(oMyObj2)
 aeval (aClassName, {|x| qout(x)})     // MYCLASS2  MYCLASS1

 @ 5,1 GET myVar                       // instant. in GETLIST[n]
 aClassName := IsObjClass(GetList[1])
 ? len(aClassName), aClassName[1], ;
       IsObjClass(GetList[1],"get")    // 1  _GGET  .T.
```

```
? IsObjClass(oMyObj1, "myclass1")     // .T. (self)
? IsObjClass(oMyObj1, "myclass2")     // .F.
? IsObjClass(oMyObj1, "test1")        // .T. (alias of self)
? IsObjClass(oMyObj1, "test2")        // .F.

? IsObjClass(oMyObj2, "myclass1")     // .T. (super, inherited)
? IsObjClass(oMyObj2, "myclass2")     // .T. (self)
? IsObjClass(oMyObj2, "test1")        // .F. (not alias of self)
? IsObjClass(oMyObj2, "test2")        // .T. (alias if self)
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

CLASS, ISOBJPROPERTY(), DBOBJECT(), ISFUNCTION()

ISOBJEQUIV ()

Syntax:

```
retL = IsObjEquiv (expO1, expO2)
```

Purpose:

Check if two objects are equivalent

Arguments:

<**expO1** and **expO2**> are the object variables to test.

Returns:

<**retL**> is .T. if both objects are equivalent, i.e. the object data are the same, otherwise F.

Description:

Two objects are equivalent, when assigning an object to another, i.e if <expO1> := <expO2> or <expO2> := <expO1>. When both the objects <expO1> and <expO2> are instantiated by themselves, the IsObjEquiv() will never return .T. although the object data may be currently the same - but they will differ at any change of the first or second object instances.

Compatibility:

New in FS5

Related:

AppIoMode()

ISOBJPROPERTY ()

Syntax 1:

```
retA = ISOBJPROPErty (expO1)
```

Syntax 2:

```
retL = ISOBJPROPErty (expO1, expN2, expC3, [expN4])
```

Purpose:

Syntax 1 determines all visible instances and available methods. Syntax 2 determines whether an exported instance, assign, access or method is specified within the object class.

Arguments:

<**expO1**> is an already instantiated object variable.

<**expN2**> specifies the type of required information:

1: is <expC3> an exported variable ?
2: is <expC3> an access method ?
3: is <expC3> an assign method ?
4: is <expC3> an usual method ?
5: is <expC3> a name of self or super class ?
6: is <expC3> a name of class alias ?

<**expC3**> specifies the name of the required instance or method.

Options:

<**expN4**> specifies the source of the property:

0: is the property generally available ?
1: is the property declared in the SELF class ?
2: is the property declared in the SUPER class ?

If the parameter is omitted, 0 is assumed.

Returns:

<**retA**> is a two-dimensional array of available properties. The first element of the sub-array contains the instance or method name (similar to <expC3>), the second the property type (similar to <expN2>). Note, that the array sorting sequence must not match the sequence of the class declaration.

<**retL**> is TRUE if the specified instance or method is available in the given object/class.

For classes inherited from DataServer or other predefined classes, TRUE may also signal that the method is available, but this may point only to "NoMethod()", "NoiVarGet()" or "NoiVarPut()", see also sect. LNG.2.11.3. In such a case, check the real availability with expN4=1.

Description:

ISOBJPROPER() determines at run-time, if the specified export instance or assign, access or usual method was declared within the class. It determines the currently linked SELF: or SUPER: property of the given object.

The simplified invocation according to syntax 1 returns an array containing information about the SELF class.

Example 1:

```
CLASS myGet INHERIT get
METHOD left CLASS myGet
   return SUPER:left()
METHOD other CLASS myGet
   return NIL
local oGet := myGet{ }
? IsProperty (oGet, 4, "left")          // visible: .T.
? IsProperty (oGet, 4, "left", 1)       // self   : .T.
? IsProperty (oGet, 4, "left", 2)       // super  : .T.

? IsProperty (oGet, 4, "right")         // visible: .T.
? IsProperty (oGet, 4, "right", 1)      // self   : .F.
? IsProperty (oGet, 4, "right", 2)      // super  : .T.
? IsProperty (oGet, 4, "other", 0)      // visible: .T.
? IsProperty (oGet, 4, "other", 1)      // self   : .T.
? IsProperty (oGet, 4, "other", 2)      // super  : .F.
```

Example 2:

```
CLASS myClass
  EXPORT  var1
  PROTECT var2

METHOD meth1 CLASS myClass
  return "something"
ACCESS var2  CLASS myClass
  return var2

ASSIGN var2(value) CLASS myClass
  return var2 := value
Function MyFunct()
LOCAL oMyObj                             // AS myClass
oMyObj := myClass {}                     // instantiate
**     := myClassNew ()                  // - alternative

? ISOBJPROPER (oMyObj, 1, "var1")        // .T.
? ISOBJPROPER (oMyObj, 1, "var2")        // .F.
? ISOBJPROPER (oMyObj, 2, "var2")        // .T.
? ISOBJPROPER (oMyObj, 2, "var2", 2)     // .F.
? ISOBJPROPER (oMyObj, 3, "var2")        // .F.
? ISOBJPROPER (oMyObj, 4, "meth1")       // .T.
? ISOBJPROPER (oMyObj, 4, "init")        // .F.

aeval (ISOBJCLASS (oMyObj), {|x| qout("**", x, "**")} )
aeval (ISOBJPROPER(oMyObj), {|x| qout(x[1], x[2])} )

   //    ** MYCLASS **
   //    METH1   4
```

```
   //    VAR1    1
   //    VAR2    2
   //    VAR2    3
return NIL
```

Classification:

programming

Compatibility:

Available in FS only. It is a superset of VO functions IsInstanceOf() and IsMethod()

Related:

CLASS, METHOD, ISOBJCLASS(), ISFUNCTION()

ISPRINTER ()

Syntax:

```
retL = ISPRINTER ()
```

Purpose:

Ensures compatibility with other dialects. FlagShip stores all printer output to the printer file or internal buffer for PrintGui(), so, logically, the printer is always "ON LINE".

Returns:

<**retL**> is always TRUE.

Classification:

programming

Compatibility:

In other xBase dialects like Clipper, ISPRINTER() determines (using the IBM BIOS call) if the LPT1 port is available and ready.

Related:

SET DEVICE, SET PRINT

ISUPPER ()

Syntax:

`retL = ISUPPER (expC)`

Purpose:

Determines if the first character of the specified string is uppercase.

Arguments:

<**expC**> is the character string to test.

Returns:

<**retL**> is TRUE if the first character of <expC> is an uppercase letter, or FALSE if it is not. If the first character is not alphabetic (A-Z, a-z), FALSE is returned.

Example:

```
? ISUPPER("a12")                    && .F.
? ISUPPER("A12")                    && .T.
? ISUPPER("1A2")                    && .F.
```

Classification:

programming

Compatibility:

International languages are supported through FS_SET("load") and FS_SET("set") functions.

Related:

ISLOWER(), ISALPHA(), ISDIGIT(), LOWER(), UPPER(), FS_SET()

L2BIN ()

Syntax:

`retC = L2BIN (expN)`

Purpose:

Transforms a numeric expression to a character string consisting of four bytes, formatted as a binary long integer. The least significant byte comes first.

Arguments:

<**expN**> is a positive numeric value to be transformed. Decimal digits are truncated (if any). The valid range is -2,147,483,647 to 2,147,483,646.

Returns:

<**retC**> is a four-byte character string containing a 32-bit binary integer in Intel convention (least significant byte comes first). On error, null-string "" is returned.

Description:

L2BIN() is used to FWRITE() a long integer (32 bits) in a Intel format to a binary file. This function is like I2BIN() which formats a short (16 bit) binary value. The inverse function of L2BIN() is BIN2L().

Example 1:

```
handle = FCREATE("data.bin")
FWRITE (handle, L2BIN(123456), 4)
FCLOSE (handle)
```

Example 2:

```
outl2bin (10)                       //  10   0   0   0
outl2bin (1)                        //   1   0   0   0
outl2bin (12345)                    //  57  48   0   0
outl2bin (-1)                       // 255 255 255 255

FUNCTION outl2bin (x)
LOCAL out, ii
out := L2BIN (x)
? x,"="
FOR ii := 1 to LEN(out)
   ?? ASC(SUBSTR(out,ii,1))
NEXT
RETURN NIL
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

BIN2I(), BIN2L(), BIN2W(), I2BIN(), FREAD(), FREADSTR()

LASTKEY ()

Syntax:

retN = LASTKEY ([expNL1], [expN2], [expNL3])

Syntax 2:

retN = LASTKEY (.T.)

Purpose:

Finds out which key was previously fetched from the keyboard buffer or clears the lastkey buffer.

Arguments:

<**expN1**> is the optional depth of the LASTKEY stack (0..9). If no argument is given, default 0 is assumed.

<**expL1**> if the parameter is specified .T. according to syntax 2, the whole lastkey buffer is cleared. Hence, LASTKEY(.T.) is a shortcut for a loop

```
FOR i := 0 to SET(_SET_LASTKEYBUFFSIZE)-1 ; LASTKEY(i,,.T.) ; NEXT
```

<**expN2**> is an input mask, filtering only specified events from the buffer. See INKEY_* constants in include/inkey.fh, default value is INKEY_ALL

<**expNL3**> is an optional logical or numeric value. If set .T., the <expN1> buffer position (or the default LastKey() value if <expN1> is not given) is cleared, and the next LASTKEY([expN1]) invocation returns 0 - unless other key was stored there by Inkey() etc. If the <expNL3> is numeric, this Inkey() value is stored in the buffer, and is returned by the next LASTKEY([expN1]) invocation.

Returns:

<**retN**> is a numeric value in the range -47 to 255, representing the ASCII value of the last (or previous) key fetched. LASTKEY(0) and LASTKEY() returns the last key pressed, LASTKEY(1) one key before it, and so on, up to LASTKEY(9). When the lastkey buffer is cleared, LASTKEY() returns a value available on the stack before it clearing.

Description:

LASTKEY() is a keyboard function that reports the INKEY() value of a key fetched from the keyboard buffer by the INKEY() function, or a wait state such as ACCEPT, INPUT, READ, WAIT, ACHOICE(), DBEDIT(), or MEMOEDIT().

LASTKEY(n) is useful for trapping previous user actions or to report errors.

The lastkey() buffer size can be increased by Set(_SET_LASTKEYBUFFSIZE, nSize) where the default is 10, and cleared using the 3rd LastKey() parameter.

See appendix table for INKEY() return codes, which are identical with the LASTKEY() codes.

Example 1:

```
memo = MEMOEDIT (memofld, 0,0,24,79, .T.)
IF LASTKEY() <> 27
   REPLACE memofld WITH memo
ENDIF
```

Example 2:

```
INKEY()                          && fetch last key press
IF FlagShip                      && call by macro to
   FOR i = 0 TO 9                && suppress Clipper errors
      x = "LASTKEY(" + LTRIM(STR(i)) + ")"
      ? "Key pressed (depth " + LTRIM(STR(i)) + ") : "
      ?? &x
   NEXT
ELSE
   ? LASTKEY()
ENDIF
```

Example 3:

```
#ifdef FlagShip                   // alternative
   FOR i = 0 TO 9
      ? "Key pressed (depth " + LTRIM(STR(i)) + ") : "
      ?? LASTKEY(i)
   NEXT
#else
   ? LASTKEY()
#endif
```

Example 4:

```
key := Inkey(0)                   // wait for key
? "Last key =", ltrim(LastKey())
CLEAR TYPEAHEAD                   // clear typeahead buffer
last := LastKey(,,.T.)            // clear current LastKey()
? "Last key =", ltrim(LastKey())
```

Classification:

programming

Compatibility:

Clipper does not support the optional argument and the LASTKEY() stack. For Clipper'87 compatible programs, use the macro call in example 2 above. For C5 compatibility, the #ifdef alternative in example 3 is more comfortable.

The 2nd and 3rd parameter is new in FS5, same as mouse events

Include:

The #include-able key manifests and constants are available in "inkey.fh", located in <FlagShip_dir>/include directory.

Related:

KEYBOARD, INKEY(), NEXTKEY()

LASTREC ()

Syntax:

retN = LASTREC ()

or:

retN = RECCOUNT ()

Purpose:

Retrieves the number of physical records in the current database file.

Returns:

<**retN**> is a numeric value, representing the number of records which are physically present in the current database file. SET FILTER and SET DELETED ON have no effect on the return value of LASTREC() and RECCOUNT().

If there is no database in use in the current working area, zero is returned.

Description:

LASTREC() is a database function that determines the number of physical records in the current database.

To execute the function in a different working area from the current one, use `alias->(LASTREC())`.

Example:

```
USE article
? LASTREC()                              && 100
SET FILTER TO Theme <> "Computers"
COUNT TO Others
? Others, LASTREC()                      && 77 100
```

Classification:

database

Compatibility:

LASTREC() is identical to RECCOUNT() in other xBase dialects.

Related:

SET FILTER, SET DELETED, EOF(), oRdd:LastRec, oRdd:RecCount

LEFT ()

Syntax:

retC = LEFT (expC1, expN2)

Purpose:

Extracts the specified number of characters from the left of the specified character string.

Arguments:

<**expC1**> is the character string to extract the characters from.

<**expN2**> is the number of characters to extract from the left side.

Returns:

<**retC**> is a character string which consists of <expN2> leftmost characters of <expC1>. If <expN2> is less than 1, a null string is returned. If <expN2> is greater than the length of <expC1>, the entire string is returned.

Description:

LEFT() is equivalent to SUBSTR(expC1, 1, expN2). The inverse operation of LEFT() is RIGHT().

Example:

```
LOCAL name := "Smith, Paul W."
? LEFT ("a12b34", 2)                    && a1
? LEFT (name, AT(",", name) -1)         && Smith
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

AT(), LTRIM(), RAT(), RIGHT(), RTRIM(), STUFF(), SUBSTR(), TRIM(), FS_SET(), FS2:ReplLeft(), FS2:RemLeft(), FS2:RemAll(), FS2:PosRepl()

LEN ()

Syntax:

`retN = LEN (expC | expA)`

Purpose:

Reports the length of a character string, the number of elements of an array or the length of a character database field.

Arguments:

<**expC**> is the character expression for which the length should be returned.

<**expA**> is the array for which number of elements should be returned.

Returns:

<**retN**> is a numeric value representing the length of a character expression, or the number of elements of the specified array. For a null-string "" or an empty array, zero is returned.

Description:

With a character string <expC>, each byte counts as one, including an embedded null byte CHR(0). The similar function STRLEN() reports the string length up to, or including zero bytes, in dependence of the FS_SET("zero", .T.) setting.

For an array, LEN() returns the number of elements. If the array is multidimensional, sub-arrays count as one element; LEN() simply returns the length of the first dimension. To determine the number of elements in another dimension, use LEN() on the (dimension -1), see LNG.2.6.4.

Example:

```
LOCAL   test1:= {1, {2,3}, {4,5,6,7}}}, str1 := "abcd"
DECLARE test2[99], test3[10,20]
test2[55] := "aaa"
? LEN(test1), LEN(test2)                    &&  3  99
? LEN(test1[2]), LEN(test1[3])              &&  2   4
? LEN(test3), LEN(test3[1])                 && 10  20
? LEN(str1),  LEN(test2[55])                &&  4   3
? LEN(""), LEN("Peter")                     &&  0   5
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

STRLEN(), FIELDLEN(), FS_SET()

LISTBOX ()

Syntax:

retO = ListBox ([expN1], [expN2], [expN3], [expN4], [expL5], [expL6])

Purpose:

Instantiate ListBox or ComboBox class, equivalent to ListBox{...}

Arguments:

<**expN1 ... expN4**> are top, left, bottom, right coordinates in that order. If not given 0,0 is assumed for top,left and maxrow/maxcol() is assumed for bottom,right coordinates. If <expL6> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

<**expL5**> if .T., **ComboBox** class will be created, otherwise ListBox is created (the default)

<**expL6**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

Returns:

<**retO**> is instantiated object variable of the Listbox or Combobox. To assign other properties, like the array with items, see OBJ.Listbox

Description:

This ListBox() function is available mainly for your convenience and for compatibility to Clipper. It is equivalent to instantiating of the Listbox{...} or ComboBox{...} class.

The ListBox class is used in GUI based application automatically for e.g. Achoice()

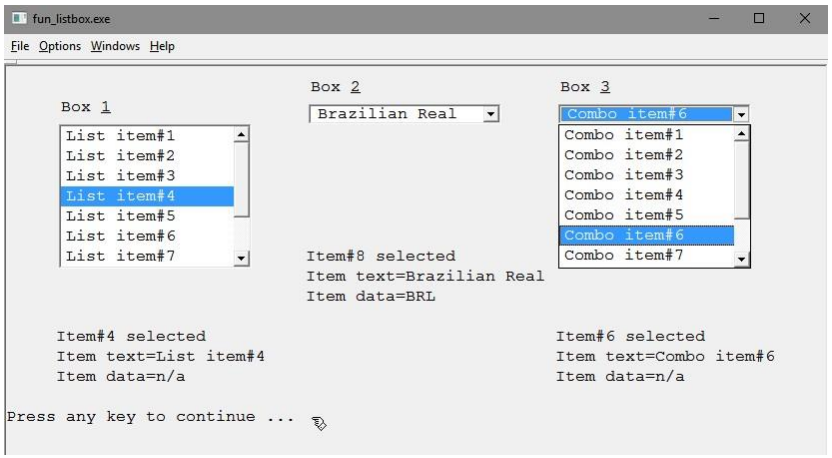
An alternative to ListBox() function is the `@..GET..LISTBOX../READ` command.

Example:

```
LOCAL oList1, oList2, oList3, ii
 set font "Courier",12
 oApplic:Resize(25,80,,.T.)

 oList1 := ListBox(2,5,  10,25)           // Listbox
 oList2 := ListBox(2,30, 10,50, .T.)      // Combobox
 oList3 := ListBox(2,55, 10,75, .T.)      // Combobox
 for ii := 1 to 10
    oList1:AddItem("List item#" + ltrim(ii))  // fill Listbox 1
    oList3:AddItem("Combo item#" + ltrim(ii)) // fill Combobox 3
 next
 oList2:FillUsing( { {"Euro","EUR"}, {"U.S. Dollars","USD"}, ;
                 {"Canad. Dollars","CDN"}, {"UK Pounds","GBP"}, ;
                 {"Japanese Yen","YEN"}, {"Swedish Krona","SEK" }, ;
                 {"Swiss Franc","CHF"}, {"Brazilian Real","BRL"} } )

 oList1:Caprow :=1.5; oList1:Capcol :=5.5; oList1:Caption :="Box &1"
 oList2:Caprow :=0.5; oList2:Capcol :=30.5;oList2:Caption :="Box &2"
 oList3:Caprow :=0.5; oList3:Capcol :=55.5;oList3:Caption :="Box &3"
```

```
oList1:Sblock := {|obj,pos,text,data| ;
     myDisplay(13, 5, obj,pos,text,data)}
oList2:Sblock := {|obj,pos,text,data| ;
     myDisplay( 9,30, obj,pos,text,data)}
oList3:Sblock := {|obj,pos,text,data|
     myDisplay(13,55, obj,pos,text,data)}

oList1:Display()    // display box (and process it in GUI mode)
oList2:Display()
oList3:Display()

if AppIoMode() == "T" // optional: display&process in Terminal i/o
  oList1:Exec()
  oList2:Exec()
  oList3:Exec()
endif

setpos(16,0)
wait

Function myDisplay(row,col, obj,pos,text,data)
@ row,col clear to row+2,col+25
@ row,col say "Item#" + ltrim(pos) + " selected"
@ row+1,col say "Item text=" + ltrim(text)
@ row+2,col say "Item data=" + if(data != NIL, data, "n/a" )
return .T.
```

Output:



Compatibility:

New in FS5, available in CL53

Related:

ComboBox{}, ListBox{} classes, Achoice()

LOCK ()

Syntax:

```
retL = LOCK ()
```

or:

```
retL = RLOCK ()
```

Purpose:

Locks the current record in the selected working area to allow write access in multiuser mode.

See more:

RLOCK () function.

LOG ()

Syntax:

`retN = LOG (expN)`

Purpose:

Calculates the natural logarithm of a numeric value.

Arguments:

<**expN**> is the numeric expression greater than zero to be calculated.

Returns:

<**retN**> is the resulting logarithm. In case of an illegal argument, zero is returned.

Description:

LOG() returns the natural logarithm as a numeric value. The natural logarithm has a base of e which is approximately 2.7183. To calculate the decimal logarithm, use LOG(expN) / LOG(10).

The inverse operation of LOG () is EXP ().

Example:

```
? LOG (2.71)                              && 1.00
? EXP (LOG(23))                           && 23.00
```

Classification:

programming

Related:

SET DECIMALS, EXP()

LOWER ()

Syntax:

retC = LOWER (expC)

Purpose:

Converts all alphabetic characters of a string to lowercase.

Arguments:

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

Returns:

<**retC**> is a character string with all uppercase alphabetic characters converted to lowercase according to their ASCII values. If the FS_SET ("loadlang") language table is used, another conversion may be specified.

Description:

LOWER() is generally used to format character strings for display purposes or to normalize strings for case-independent comparison or indexing.

For proper Lower() conversion of extended character set > chr(127), the FSsortab.def - or other table assigned by FS_SET("Setl"/"Load") is used.

The inverse operation of LOWER() is UPPER().

Example:

```
? LOWER("ABCDE")                          && abcde
? LOWER("1266 Maple St.")                 && 1266 maple st.
? UPPER("1266 Maple St.")                 && 1266 MAPLE ST.
FS_SET ("loadlang", 1, "FSsortab.ger")
FS_SET ("setlang",  1)
? LOWER("MÜLLER")                         && müller
```

Classification:

programming

Compatibility:

The user defined lower and upper translation using tables loaded by FS_SET("load") and FS_SET("setlang") is available in FlagShip only. FS supports embedded zero bytes by default.

Related:

ISLOWER(), UPPER(), ISUPPER(), ISALPHA(), ISDIGIT(), FS_SET()

LTRIM ()

Syntax:

retC = LTRIM (expC|expN)

Purpose:

Removes all leading spaces from a string.

Arguments:

<**expC**> is the character string to be trimmed.

<**expN**> is a numeric value which is converted to string first and then trimmed.

Returns:

<**retC**> is a copy of the <expC> string with all the leading blanks removed. In case of a null string "" or all spaces, LTRIM() returns null string "".

Description:

LTRIM() can be used to remove leading blanks when numbers are converted to strings by means of STR (), and generally for formatting strings with leading blanks.

LTRIM() is related to RTRIM() which removes trailing spaces, and ALLTRIM() which removes both leading and trailing spaces.

The inverse operation of LTRIM() is PADL().

Example:

```
LOCAL yrs := 23
? STR(yrs)+ " years old."          && "      23 years old."
? LTRIM(STR(yrs)) + " years old."  && "23 years old."
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default. FS5 accept numeric parameter as well.

Related:

ALLTRIM(), STR(), SUBSTR(), TRIM(), RTRIM(), PADL(), PADR(), PADC(), FS_SET(), FS2:RemLeft()

LUPDATE ()

Syntax:

`retD = LUPDATE ()`

Purpose:

Returns the last modification date of the current database file.

Returns:

<**retD**> is a date value, reflecting the date of the last update. If there is no database file in use, an empty date is returned.

Description:

LUPDATE() is a database function that determines when the database in the current working area was last modified. This date is updated whenever the number of records is changed, the database is closed or when COMMIT or DBCOMMIT() is called.

To execute the function in a different working area from the current one, use `alias->(LUPDATE())`.

With the default SET DB3COMPAT OFF, the modification date from 1/1/1900 to 12/31/2155 is supported. With SET DB3COMPAT ON, FlagShip's LUPDATE() adds 1900 when the year saved in .dbf is greater than 50, and 2000 otherwise.

Example:

```
SET CENTURY ON
USE article
? LUPDATE(), DATE()                    && 06/25/1998  02/03/2011
DELETE
PACK
? LUPDATE(), DATE()                    && 02/03/2011  02/03/2011
USE
USE article
? LUPDATE(), DATE()                    && 02/03/2011  02/03/2011
```

Classification:

database

Compatibility:

SET DB3COMPAT is new in FS7. Note that FlagShip saves the header on any .dbf update, Clipper at closing the database. dBaseIII+ and Clipper displays dates for 20th century (1900..1999) only.

Related:

CLOSE, SELECT, USE, oRdd:Lupdate, SET DB3COMPAT

MACROEVAL ()

Syntax:

`ret = MacroEval (expC1)`

Purpose:

Evaluate macro, similar to &(expC1) but works on any variable type as well as on fields

Arguments:

<**expC1**> is the macro string to be evaluated.

Returns:

<**ret**> is the result of macro evaluation.

Description:

This MacroEval() function is similar to run-time macro evaluation by the assignment ret := &(expC1) but it accept also fields or any character expression as parameter.

Example:

```
cMacro := "date()"
? &(cMacro)                  // 01/31/2002
? MacroEval(cMacro)          // 01/31/2002
```

Classification:

programming

Compatibility:

New in FS5

Related:

Macro evaluation in LNG.2.10, MacroSubst()

MACROSUBST ()

Syntax:

`retC = MacroSubst (expC1)`

Purpose:

Evaluate/substitute macro, similar to "...&macro ..." in text, but works on variables, expression and on fields, not only on string constant as with the usual Text-Substitution (see LNG.2.10)

Arguments:

<**expC1**> is a string containing optionally macro(s) to be evaluated.

Returns:

<**retC**> is the result of macro evaluation. On error (e.g. with non character parameter), NIL is returned.

Description:

This function is similar to run-time macro substitution by the assignment cRet := "any text `&cMacroVar` any text" but it accept also fields, or any character expression as parameter.

As opposite to the assignment where the macro must be visible at the time of constant assignment, the MacroSubst() evaluates the whole parameter string. You may see it as a "late" substitution as opposite to the assignment where the substitution is initiated at compile time.

The same rules as with Text-Substitution apply to be evaluated:

- The macro operator & is immediately followed by character(s) without blanks, and
- The characters behind the macro operator (up to space or .) are assumed to be a name of (auto)PRIVATE or PUBLIC variable and if such is available and of type character, it is macro- evaluated/substituted. This means, the corresponding part of <expC1> is replaced by the content of the macro variable.
- The substitution may be nested, i.e. the macro variable can contain another macro. The nesting depth is limited to 100 to avoid unintentional endless recursion.

If the macro variable specified within the <expC1> is not available or is not of character type, no substitution is taken and <expC1> is returned "as is".

Example:

```
PRIVATE cMacro := "Hello", cMacro2
LOCAL cText
cText := "&cMacro world"      // substituted at constant assignment
? cText                       // "Hello world"
? MacroSubst(cText)           // "Hello world"

cText := "&cMac" + "ro world" // no substitution in these constants
? cText                       // "&cMacro world"
```

```
? MacroSubst(cText)           // "Hello world"
? MacroSubst("&" + "cMac" + "ro world")  // "Hello world"

cMacro2 := "Hi and &" + "cMacro"         // nested macro
? MacroSubst("&" + "cMacro2 world")      // "Hi and Hello world"
```

Classification:

programming

Compatibility:

New in FS5

Related:

Macro evaluation and Text substitution in LNG.2.10, MacroEval()

MAX ()

Syntax:

```
retN = MAX (expN1, expN2, [... expNn])
retD = MAX (expD1, expD2, [... expDn])
```

Purpose:

Returns the larger of two or more numeric or date values.

Arguments:

<**expN1**> ... <**expNn**> are the numeric expressions to be compared.

<**expD1**> ... <**expDn**> are the date expressions to be compared.

Returns:

<**retN**> or <**retD**> is the largest of the two or more arguments.

Description:

MAX() can be used when the result of an expression has to be kept greater than a minimum value.

The inverse operation of MAX() is MIN().

Example:

```
? MAX (100, 88)                        && 100
? MAX (88, 100)                        && 100
? MAX (-88, -100)                      && -88
? DATE(), MAX (DATE(), DATE()+40)      && 09/05/93   10/15/93
```

Classification:

programming

Compatibility:

The support of more than two arguments is new in FS5

Related:

MIN()

MAXCOL ()
MAX_COL ()

Syntax:

```
retN = MAXCOL ([expLN1], [expN2])
retN = MAX_COL ( )
```

Purpose:

Determines the actual size of the output screen or the GUI window or of the FS2 sub-window.

Arguments:

<**expLN1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col. You may specify <expLN1> also numeric as UNIT_ROWCOL, UNIT_PIXEL, UNIT_MM, UNIT_CM or UNIT_INCH. If not given, the current SET PIXEL or SET COORD UNIT is used, the default is UNIT_ROWCOL (screen columns).

<**expN2**> is new MaxCol setting in either cols or pixel, depending on <expLN1>. If only one parameter is supplied and is numeric > 6, it is accepted as new MaxCol setting too. Note that MaxCol() is updated automatically in GUI mode, when you display any character via @..SAY or ?, ??, Qout() or Qqout() etc. at a line exceeding the current MaxCol() value.

Returns:

<**retN**> is the rightmost available screen column (0.. ..retN), according to the terminfo entry for the current terminal TERM in Terminal i/o or from the current window size in GUI. If <expLN1> is not given or is UNIT_ROWCOL and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font). The coordinate return value is rounded to integer by default, but the behavior may be changed, see tuning below.

Description:

MAXCOL() and MAX_COL() are equivalent functions that determine the maximum visible column of the screen and allows you to write full terminal independent programs. They differ in MDI applications or when using FS2 windowing: MAX_COL() always report the size of main screen, i.e. of window ID# 0, whilst MAXCOL() the size of currently selected sub-window.

Terminal i/o: Changing or setting the environment variable COLUMNS will override the terminfo entry cols#. This is automatically detected at program start-up. While the application is running, the curses and the MAXCOL() setting may be reinitialized using the FS_SET("setenv","COLUMNS","80",.T.) function, which will set MAXCOL() to 79. If the terminal cannot display the number of columns set, garbage may be output.

GUI i/o: the visible window size may by changed/resized by a mouse, except you specify fixed size by oApplic:DefSize(). The used drawing area (pane) is not affected by the window resizing, the whole user window is scrollable by the horizontal and vertical bars. The MaxCol() function return the current size of the user area, not of

the visible window size. Use oApplic:DefSize() method if you want to determine size of the application window. For resizing the application window, see oApplic:Resize()

Basic i/o: MaxCol() return constant 79 simulating 80 cols display

If you are using sub-windows via standard MDIopen() or the Wopen() from FS2 Toolbox, MaxRow() and MaxCol() reports the size of the current sub-window. You may also use WmaxRow() and WmaxCol() from the FS2 package which allows you to specify the window ID# without changing the selection.

Tuning:

The return value is usually rounded to integer. You may disable this behavior by setting

```
_aGlobSetting[GSET_L_ROUNDMAXROWCOL] := .F. // default = .T.
```

Example:

```
CLEAR
@ 0,0 to MAXROW(),MAXCOL()              && box full screen

IF MAXCOL() < 79
   ? "Cannot execute this application with your terminal "
   ?? FS_SET ("term")
   ? "because 80 columns needed, available are " + ;
      ltrim(str(MAXCOL() +1)) + " only."
   ? "please check other settings for TERM and start again"
   QUIT
ENDIF
```

Classification:

programming, system, terminfo, environment vars

Compatibility:

MAXCOL() is available in C5. MAX_COL() is supported for backward compatibility to FS3. See also terminfo settings and the environment variable TERM and COLUMNS, if such is set. The GUI mode and oApplic object is new in FS5

Related:

MAXROW(), COL(), ROW(), FS_SET(), SYS.2, FSC.3.3

MAXROW ()
MAX_ROW ()

Syntax:

```
retN = MAXROW ([expLN1], [expN2])
retN = MAX_ROW ( )
```

Purpose:

Determines the actual size of the output screen or the GUI window or of the FS2 sub-window.

Arguments:

<**expLN1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col. You may specify <expLN1> also numeric as UNIT_ROWCOL, UNIT_PIXEL, UNIT_MM, UNIT_CM or UNIT_INCH. If not given, the current SET PIXEL or SET COORD UNIT is used, the default is UNIT_ROWCOL (screen rows).

<**expN2**> is new MaxRow setting in either rows or pixel, depending on <expLN1>. If only one parameter is supplied and is numeric > 6, it is accepted as new MaxRow setting too. Note that MaxRow() is updated automatically in GUI mode, when you display any character via @..SAY or ? or Qout() etc. at a line exceeding the current MaxRow() value. If <expLN1> is not given or is UNIT_ROWCOL and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

Returns:

<**retN**> is the last available screen row (0..retN), according to the terminfo entry for the current terminal TERM in Terminal i/o or from the current window size in GUI. If <expLN1> is not given or is UNIT_ROWCOL and SET PIXEL or SET COORD is not set, GUI coordinates are calculated from current SET FONT (or oApplic:Font). The coordinate return value is rounded to integer by default, but this may be changed, see tuning below.

Description:

MAXROW() and MAX_ROW() are equivalent functions that determine the maximum visible row of the screen and enables you to write full terminal independent programs. They differ in MDI applications or when using FS2 windowing: MAX_ROW() always report the size of main screen, i.e. of window ID# 0, whilst MAXROW() the size of currently selected sub-window.

Terminal i/o: Changing or setting the environment variable LINES will override the terminfo entry lines#. This is automatically detected at program start-up. While the application is running, the curses and the MAXROW() setting may be reinitialized using the `FS_SET("setenv", "LINES", "43", .T.)` function, which will also set MAXROW() to 42. If the current terminal cannot display the number of lines set, garbage may be output.

GUI i/o: the visible window size may by changed/resized by a mouse, except you specify fixed size by oApplic:DefSize(). The used drawing area (pane) is not affected by the window resizing, the whole user window is scrollable by the horizontal and vertical bars. The MaxRow() function return the current size of the user area, not of the visible window size. Use oApplic:DefSize() method if you want to determine size of the application window. For resizing the application window, see oApplic:Resize() To determine the currently visible rows (instead of max. available rows), use RowVisible() instead of MaxRow().

Basic i/o: MaxRow() return constant 24 simulating 25 line display

If you are using sub-windows via standard MDIopen() or the Wopen() from FS2 Toolbox, MaxRow() and MaxCol() reports the size of the current sub-window. You may also use WmaxRow() and WmaxCol() from the FS2 package which allows you to specify the window ID# without changing the selection and which corresponds to the Clipper's CT3 MaxRow() and MaxCol().

Tuning:

The return value is usually rounded to integer. You may disable this behavior by setting

```
_aGlobSetting[GSET_L_ROUNDMAXROWCOL] := .F. // default = .T.
```

Example:

```
CLEAR
@ 0,0 to MAXROW(),MAXCOL() && box full screen
@ MAXROW(),0 say "this is the last line"
```

Classification:

programming, system, terminfo, environment vars

Compatibility:

MAXROW() is available in Clip5. MAX_ROW() is supported for backward compatibility to FS3. See also terminfo settings and the environment variable TERM and LINES, if such is set and RowVisible() for GUI.

The GUI mode and oApplic object is new in FS5

Note, some UNIX terminals support 24 instead of 25 lines only. It is wise for portable programs to use e.g. @ MAXROW(),0 SAY "msg" instead of @ 24,0 SAY "msg" in order to use the last available screen line.

The GUI mode and oApplic object is new in FS5

Related:

MAXCOL(), COL(), ROW(), RowVisible(), FS_SET(), SYS.2, FSC.3.3

MCOL ()

Syntax:

```
retN = Mcol ([expL1], [expL2])
```

Purpose:

Determine the mouse cursor's user-screen column position

Arguments:

<**expL1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col,otherwise the current SET PIXEL status is used.

<**expL2**> apply for row/col coordinates only. If .F., the return value as integer (default), .T. returns decimal fraction.

Returns:

<**retN**> is the current mouse cursor column position. If <expL1> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

Description:

The mouse support is available in GUI mode only. In the Terminal and Basic i/o mode, 0 is returned.

Compatibility:

New in FS5, available in CL53

Related:

Mpresent(), Mrow(), MsetCursor(), MsetPos(), Mhide(), Mshow(), MsetCursor(), MsaveState(), MrestState(), MleftDown(), MrightDown(), Mstate(), Col(), MaxCol()

MDBLCK ()

Syntax:

`retN = MdbLck ([expN1])`

Purpose:

Determine or set the double-click speed threshold of the mouse

Arguments:

<expN1> is the maximum allowable amount of time between mouse key presses for a double-click to be detected. This is measured in milliseconds.

Returns:

<**retN**> is the current double-click speed threshold

Description:

This function is used to set or retrieve the current mouse's double-click speed threshold. It is useful when the mouse's double-click sensitivity needs to be adjusted.

Compatibility:

New in FS5, available in CL53

Related:

Mrow(), Mcol(), MSetCursor()

MDICLOSE ()

Syntax:

`retN = MDIclose ( [expN1] )`

Purpose:

Close the current or specified MDI sub-window open by MDIopen() or by Wopen() in GUI mode.

Arguments:

<**expN1**> is the window ID# for which this function apply. If not specified, the currently selected sub-window is closed, except it is the main screen with ID#0.

Returns:

<**retN**> is the ID# number of the currently selected sub-window, or -1 if the call fails.

Description:

MDIclose() closes the currently selected MDI sub-window and select a sub-window with the highest ID#. MDIclose(expN1) closes the specified sub-window and does not change the selection.

The ID# (handle) of the closed sub-window may be re-used by the next Wopen() call. Using <expN1> = 0 or invoking Wclose() without parameter from the main screen is ignored.

MDIclose() is fully equivalent to the FS2 function Wclose(expN1) call, but does not require FS2 license. To use sub-windows in SDI mode or in Terminal i/o, refer to Wopen() and Wclose() functions from FS2 Toolbox.

Note: the close icon [X] displayed in the sub-window title frame is created by the window manager, not by FlagShip self. An user click on this button (in sub-window) is simply ignored, as opposite to the same button of the whole application which is handled by the InitIoQuit() function, see details in the initiomenu.prg source. To close the sub-window programmatically, use this MdiClose().

Example:

```
w1 := MDIopen(10,10, 20,50)
w2 := MDIopen(20,20, 25,60, "Second sub-window")
w3 := MDIopen( 5,39, 15,60, "Third sub-window")

w := MDIselect(w1)
w := MDIclose()       // closes w1, w3 is selected thereafter
w := MDIclose(w2)     // closes w2, w3 remain selected
w := MDIclose()       // closes w3, main screen ID#0 is selected
```

***Classification*:** programming, windowing

Compatibility: new in FS5, compatible to Wclose() of FS2-Toolbox

Related:

MDIopen(), MDIselect(), FS2:Wopen(), FS2:Wclose(), FS2:WaClose()

MDIOPEN ()

Syntax:

```
retN = MDIopen ( expN1, expN2, expN3, expN4,
                 [expC5], [expL6] )
```

Purpose:

Open a new MDI sub-window. Applicable for GUI mode only for MDI applications (e.g. compiled with the -mdi switch).

Arguments:

<**expN1**>...<**expN4**> are the top, left, bottom, right coordinates of the new sub-window, either in row/col or in pixel. Accepted values are between 0..MaxRow() and 0..MaxCol()

Options:

<**expC5**> is an optional string, specifying the sub-window caption. If not empty(), Wopen() calls Wcaption(<expC6>). If not specified, the sub-window caption in GUI mode is "Window #nn" where nn is the window ID# returned by Wopen(). In Terminal i/o mode, when <expC6> is not specified or is an empty string, no action is taken.

<**expL6**> is an optional pixel specification. If .T., the coordinates are in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used. Apply for GUI mode only, ignored otherwise.

Returns:

<**retN**> is the ID# number of the new and selected sub-window, or -1 if the call fails. You will need to store the return value to a variable, to be able to access/select it later by MDIselect() or to close the sub-window via MDIclose().

Warnings and errors are reported in Developer's mode, i.e. when FS_SET("devel",.T.) is set. Otherwise check the <retN> for value <= 0.

Description:

MDIOPEN() creates and opens a new MDI sub-window. If successful, it returns a number (handle) for this window and select it. If the coordinates are invalid, the application does not run in MDI mode, or the call fails for other reason, MDIOPEN() returns -1 and the window that was active at the time of the function call (usually the main screen) is still active. There is no limit of used sub- windows in FlagShip.

A MDI sub-window is a separate widget (control) in GUI. From programmer's view, it differs from the main screen only in size. In MDI mode (i.e. when compiled with the -mdi switch), the first sub-window is created automatically and becomes ID# 0. After you open new sub-window, all screen output is redirected to the window and this sub-window receives input focus.

The sub-window behaves same as the main screen. All the standard settings, including SET PIXEL ON|OFF apply also for the selected sub-window, same as the row/column status like Col(), Row(), MaxCol() and MaxRow().

MDIOPEN() is fully equivalent to a call of FS2 function

```
id := WOPEN (expN1, expN2, expN3, expN4, .F., expC5, 0, expL6)
```

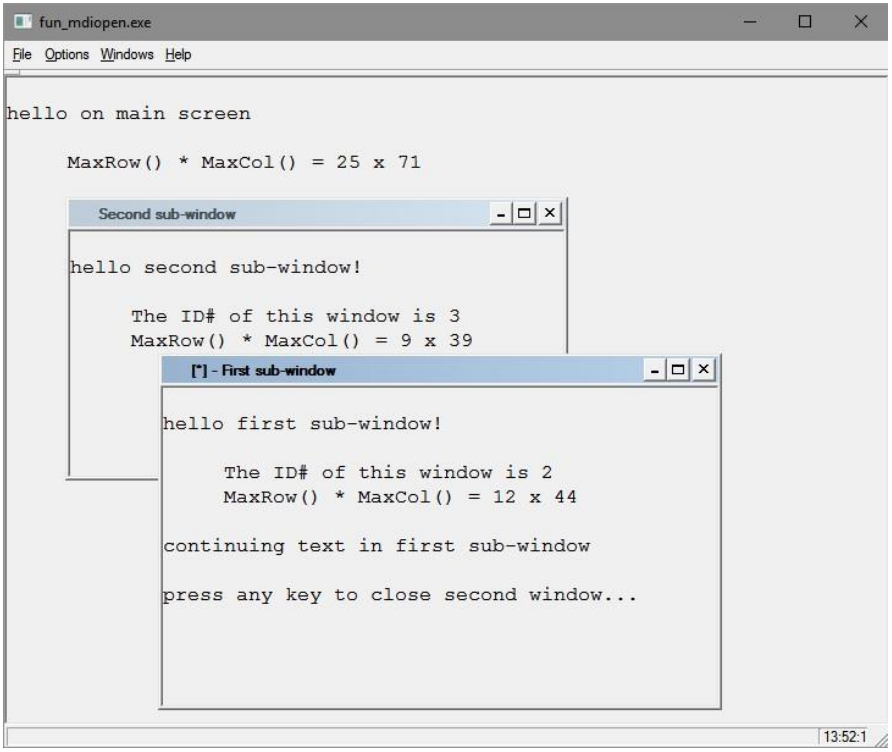
but MDIOPEN() does not require FS2 license. In fact, MDIOPEN() is a subset of WOPEN(). To use sub-windows in SDI mode or in Terminal i/o, refer to WOPEN() function from FS2 Toolbox. You also may use all other W*() windowing functions from FS2, like Wmove(), WmoveUser(), Wclose(), WaClose(), Wselect() and so on with the returned ID# from MDIOPEN () in GUI mode.

You can retrieve the active window ID# by calling MDISELECT() with no parameters. The active window is marked by **[*]** sign in the header and with "checked" mark in the Menu->Windows->Sub-Windows list. You may select and view an inactive window (read-only) by Menu->Windows->Sub-Windows (user modifiable), see also initiomenu.prg for details.

Note: the close icon [X] displayed in the sub-window title frame is created by the window manager, not by FlagShip self. An user click on this button (in sub-window) is simply ignored, as opposite to the same button of the whole application which is handled by the InitIoQuit() function, see details in the initiomenu.prg source. To close the sub-window programmatically, use MDICLOSE().

Example:

```
set font "courier",12
oApplic:resize(25,70,,.T.)

win1 := MDIopen(10,10, 23,55, "First sub-window")
if win1 <= 0
   alert("Not compiled by the -mdi switch, so no MDI sub-windows")
else
   ? "hello first sub-window!"
   @ 3,5 SAY "The ID# of this window is " + ltrim(win1)
   @ 4,5 SAY "MaxRow() * MaxCol() = " + ltrim(MaxRow()) + " x " + ;
             ltrim(MaxCol())

   win2 := MDIopen(5,5, 15,45, "Second sub-window")
   ? "hello second sub-window!"
   @ 3,5 SAY "The ID# of this window is " + ltrim(win2)
   @ 4,5 SAY "MaxRow() * MaxCol() = " + ltrim(MaxRow()) + " x " + ;
             ltrim(MaxCol())

   MDIselect(0)                          // select main window
   ? "hello on main screen"
   @ 3,5 SAY "MaxRow() * MaxCol() = " + ltrim(MaxRow()) + " x " + ;
             ltrim(MaxCol())
   MDIselect(win1)                       // select first sub-window
   @ 6,0 say "continuing text in first sub-window"
   ?
   wait "press any key to close second window..."
   MDIclose(win2)                        // close second sub-window
   wait "any key to close this sub-window..."
   MDIclose()                            // close current sub-window
endif
wait
```



Classification: programming, windowing

Compatibility: new in FS5, compatible to Wopen() of FS2-Toolbox

Related: MDIselect(), MDIclose(), FS2:Wopen(), .../examples/mdi.prg

MDISELECT ()

Syntax:

```
retN = MDIselect ( [expN1], [expL2] )
```

Purpose:

Activates one of the open MDI sub-windows or the main screen.

Options:

<**expN1**> is the window ID# that is selected. The ID# is returned from MDIopen() or Wopen(). Specifying 0 (zero) selects the main screen. When <expN1> is not specified, MDIselect() only returns the ID# of the currently selected sub-window.

<**expL2**> is an optional logical value, specifying if the newly selected sub-window should receive input focus. The default is .T.

Returns:

<**retN**> is the ID# number of the currently selected sub-window, or -1 if the call fails.

Description:

MDIselect() allows you to take an already open but inactive MDI sub-window and reactivate it. Window ID# 0 corresponds to the main window. By omitting the first parameter you can determine the ID# (handle) of the window currently selected.

MDIselect() is fully equivalent to FS2 function Wselect() call, but MDIselect() does not require FS2 license. To use sub-windows in SDI mode or in Terminal i/o, refer to Wopen() and Wclose() functions from FS2 Toolbox.

Example:

See MDIopen()

Classification:

programming, windowing

Compatibility:

New in FS5, compatible to Wselect() of FS2-Toolbox.

Related:

MDIopen(), MDIclose(), FS2:Wopen(), FS2:Wclose(), FS2:Wselect()

MEMOCODE ()
MEMOENCODE ()

Syntax:

`retC = MemoCode ( expC1 , [expL2] )`

or

`retC = MemoEncode ( expC1, [expL2] )`

Purpose:

Encodes the passed string so that the result does not contain CHR(0) nor CHR(26).

Arguments:

<**expC1**> is the character string that is processed. This string may contain any byte sequence, also embedded zero bytes CHR(0) and CHR(26).

Option:

<**expL2**> is optional logical expression which controls whether extended translation may be used. Value of .F. prohibits it, i.e. when <expC1> contains all available 255 bytes no translation is taken and a Developer Warning (if enabled) will arise. Otherwise, if <expL2> is .T. or NIL or not specified, extended translation is taken in such a case, resulting in larger string <retC>.

Returns:

<**retC**> is the encoded string. It is always 8 bytes longer than the length of <exC1>, except when extended translation is required, see <expL2>. The result does not contain CHR(0) nor CHR(26) and can be safely stored in database memo fields. On error, "" is returned.

Description:

MemoEncode() encodes the passed <expC1> string so that the <retC> does not contain zero byte nor CHR(26) and can hence be safely stored in memo fields of a database. To read the memo field into variable and by decoding it to clear text, use MemoDecode().

An already coded string by MemoEncode() cannot be passed to MemoEncode() anew, a run-time error is displayed in such a case. You should avoid any string manipulation like concatenation, substr() etc. on the resulting <retC> string, otherwise unpredictable results may occur by decoding via MemoDecode().

When CHR(0) and/or CHR(26) is found in <expC1>, MemoEncode() replaces these character by an unique character not available otherwise in the text, e.g. ^X and ^Y, or CHR(1) and CHR(2) etc. Only when the <expC1> already includes all available character values (1..255) and the extended translation via <expL2> is disabled, the coding fails and run-time error is displayed. The resulting string <retC> is prefixed by an unique byte sequence which includes also the replacement characters and is hence 8 bytes longer than <expC1> (or even longer when extended translation is required).

Example 1:

```
// incorrect results
cMemoText := "Abcd" + chr(26) + "efg" + chr(0) + "ijk"
APPEND BLANK
REPLACE FIELD->Memo WITH cMemoText
...
cStr := FIELD->Memo
? len(cStr)             // 4 since CHR(26) terminates it in .dbt
? cStr                  // "Abcd"
? len(FIELD->Memo)      // 4

// incorrect results
cMemoText := "Abcdefg" + chr(0) + "ijk"
REPLACE FIELD->Memo WITH cMemoText
...
cStr := FIELD->Memo
? len(cStr)             // 7 since CHR(0) terminates it in .dbt
? cStr                  // "Abcdefg"
? len(FIELD->Memo)      // 7

// correct results
cMemoText := "Abcd" + chr(26) + "efg" + chr(0) + "ijk"
REPLACE FIELD->Memo WITH MemoEncode(cMemoText)
...
cStr := MemoDecode(FIELD->Memo)
? len(cStr)             // 12 = ok
? cStr                  // "Abcd\032efg\000ijk"
? len(FIELD->Memo)      // 20
```

Example 2:

If you need to encode memo data in Clipper to use cross-platform .dbt files coded by this way, add this simplified code somewhere in your Clipper or multi-platform source:

```
#ifndef FlagShip
 FUNCTION MemoEncode(str)
   static cMagic := chr(1) + chr(239) + chr(254)  // don't change
   local ii, cRepl1, cRepl2, cRet
   if len(str) >= 8 .and. left(str,3) == cMagic .and. ;
         substr(str,6,3) == cMagic
      alert("MemoEncode(): string already coded by MemoEncode()")
      return ""
   endif
   cRepl1 := cRepl2 := chr(0)
   for ii := 1 to 255
      if ii != 26 .and. !(chr(ii) $ str)
         cRepl1 := chr(ii)
         exit
      endif
   next
   for ii := asc(cRepl1) +1 to 255
      if ii != 26 .and. !(chr(ii) $ str)
         cRepl2 := chr(ii)
         exit
      endif
   next
   if cRepl2 == chr(0)
```

```
      alert("MemoEncode() cannot convert,;all 253 char values used")
      return ""
    endif
    cRet := strtran(str,  chr(0),  cRepl1)
    cRet := strtran(cRet, chr(26), cRepl2)
    return cMagic + cRepl1 + cRepl2 + cMagic + cRet
  FUNCTION MemoCode(str)
    return MemoEncode(str)
#endif
```

Classification:

programming, database handling

Compatibility:

Available since FS5. See above example for Clipper.

Related:

MemoDecode(), Strtran(), FS2:CharPack(), FS2:CharRepl()

MEMODECODE ()

Syntax:

`retC = MemoDecode ( expC1 )`

Purpose:

Decodes the string previously coded by MemoEncode().

Arguments:

<**expC1**> is the character string that is processed. This is usually a memo field of a database.

Returns:

<**retC**> is the decoded string. It is always 8 bytes shorter than the length of <exC1>.

Description:

MemoDecode() decodes the string previously encoded by MemoEncode() or by MemoCode().

The returned value may contain embedded zero byte CHR(0) and CHR(26).

When the <expC1> string was not coded by MemoEncode(), run-time error is displayed.

Example 1:

see MemoEncode()

Example 2:

If you need to read FlagShip data coded by MemoEncode() in Clipper, and to use cross-platform .dbt files coded by this way, add this simplified code somewhere in your Clipper or multi-platform source:

```
#ifndef FlagShip
 FUNCTION MemoDecode(str)
   static cMagic := chr(1) + chr(239) + chr(254)  // don't change
   local cRepl1, cRepl2, cRet
   if len(str) < 8 .or. !(left(str,3) == cMagic) .or. ;
         !(substr(str,6,3) == cMagic)
      alert("MemoDecode(): string not created by MemoEncode()")
      return ""
   endif
   cRepl1 := substr(str,4,1)
   cRepl2 := substr(str,5,1)
   cRet := substr(str,9)
   cRet := strtran(cRet, cRepl1, chr(0))
   cRet := strtran(cRet, cRepl2, chr(26))
   return cRet
#endif
```

Classification:

programming, database .dbt handling

Compatibility:

Available since FS5. See above example for Clipper.

Related:

MemoCode(), MemoEncode(), Strtran(), FS2:CharUnpack(), FS2:CharRepl()

MEMOEDIT ()

Syntax:

```
retC = MEMOEDIT ([expC1], [expN2], [expN3],
           [expN4], [expN5], [expL6], [expC7],
           [expN8], [expN9], [expN10], [expN11],
           [expN12], [expN13],[expL14], [expO15],
           [expL16])
```

Purpose:

Displays and edits a text from a character string or a memo field using a window area or the screen.

Options:

<**expC1**> is the character string or memo field to display and edit.

<**expN2**...**expN5**> defines the top, left, bottom and right window coordinates. If they are all omitted, the entire screen is used. If <expL14> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

<**expL6**> determines whether a memo is edited (.T.) or simply displayed (.F.). The default is TRUE.

<**expC7**>|<**expL7**> is a string specifying the name of a user-defined function to control the editing process. The function name must be specified without parentheses or arguments. Refer to the discussion below for more information. If a logical FALSE is specified, MEMOEDIT() exits immediately after displaying the text; the state of <expL**6**> is ignored in such a case. Logical TRUE is equivalent to NIL.

<**expN8**> determines the line length, which defaults to <expN5> - <expN3>. If <expN8> is greater that the default, the window scrolls horizontally.

<**expN9**> determines the tabulator size to be expanded by spaces. The default value is 4. If tab characters CHR(9) are included in the <expC1> string or the TAB key is pressed, MEMOEDIT() expands them by spaces.

<**expN10**> is the initial memo line where the cursor is placed. The default is 1.

<**expN11**> is the initial memo column where the cursor is placed. The default is 0.

<**expN12**> is the initial row for placing the cursor relative to the window position. The default is 0.

<**expN13**> is the initial column for placing the cursor relative to the window position. The default is 0.

<**expL14**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

<**expO15**> is an optional Font object, used for the MemoEdit(). Applicable for GUI mode only, ignored otherwise.

<**expL16**> is a logical value specifying if the MemoEdit() widget should be deleted at exit from MemoEdit(). A .T. value clears the widget on exit, .F. (the default) remains the widget visible. Applicable for GUI mode only, ignored otherwise.

Returns:

<**retC**> is a character string. When terminated with Ctrl-W, MEMOEDIT() returns the modified string. The original string is returned and the changes are discarded if ESC is entered.

Description:

MEMOEDIT() is a general purpose text browsing and editing function that edits memo fields and long character strings. It supports a number of different modes and a user-defined function to allow key reconfiguration and other activities that enable the design of flexible user interfaces and programs.

MEMOEDIT() supports the following keys and actions, and their configuration in the UDF specified (Table 1):

Key		Action	Brow	config
Cursor up	ctrl-E	Move up one line	*	no
Cursor down	ctrl-X	Move down one line	*	no
Cursor left <-	ctrl-S	Move left one character	*	no
Cursor right ->	ctrl-D	Move right one char	*	no
ctrl-Cursor <-	ctrl-Z	Move left one word		no
ctrl-Cursor ->	ctrl-B	Move right one word		no
Home	ctrl-A	Beginning of current line	*	no
End	ctrl-F	End of current line	*	no
ctrl-Home	ctrl-]	Begin of current window	*	no
	ctrl-L	End of current window	*	no
PgUp	ctrl-R	Previous window	*	no
PgDn	ctrl-C	Next window	*	no
ctrl-PgUp	ctrl--	Beginning of text	*	no
ctrl-PgDn	ctrl-^	End of text	*	no
Insert	ctrl-V	Toggle insert on/off		yes
Delete	ctrl-G	Delete current char		no
Backspace	ctrl-H	Delete left character		no
	ctrl-Y	Delete current line		yes
	ctrl-T	Delete word right		yes
Tab	ctrl-I	Insert spaces to next tab		no
Enter	ctrl-M	Next line, new paragraph		no
	ctrl-U	Reformat paragraph		yes
ctrl-End	ctrl-W	Finish edit, save	*	yes
Esc	ctrl-[	Abort edit, don't save	*	yes
other key	CHR(-47..31)	Pass config key to UDF		yes
other key	CHR(32..255)	Insert char in text	*	(yes)
Left-Mouse-Butt-Down		Mark text for copy-and-paste	**	
Mid-Mouse-Button	Alt-C	Copy marked text to clipboard	**	
Shift+Mid-Mouse-But	Alt-V	Paste clipboard to curr.field	**	

* Browse (read-only) mode only supported keys

** GUI mode only. See description in "cut-and-paste" below.

Modes:

MEMOEDIT() supports two display modes depending on the value of <expL6>:

- In the **edit and update mode** (<expL6> is TRUE, the default), all the above listed navigation and editing keys are active and the user can change the contents of the text buffer. Some of the navigation keys are configurable within the user-defined function <expC7>, if such are specified. If ESC or ^W are pressed, MEMOEDIT() exits and the text buffer or the original text is returned. To avoid unintended loosing of text changes, MemoEdit() prompts you on ESC considering the state of Set(_SET_MEMOEDITWARN), see details in FUN:Set().
- In the **browse mode** (<expL6> is FALSE), all navigation keys are active and have the same functionality as in the update mode except that the default scroll state is on. Uparrow and Downarrow scroll the contents of the MEMOEDIT() window up or down one line. The user can only navigate about the text buffer but cannot edit or insert new text. If ESC or ^W are pressed, MEMOEDIT() exits, returning the original text.

Scrolling mode: when 35 is returned from the UDF, or when in browse mode, the default cursor up and down movement toggles to the movement of the whole text window relative to the current cursor position.

Text buffer:

The specified string or memo field <expC1> is first copied to an internal text buffer, which is edited or browsed by the user. Exiting MEMOEDIT() with ^W, the contents of the text buffer are copied to the return variable <retC>. Upon pressing ESC, the contents of <expC1> are copied to <retC>. This return value can then be assigned to a variable or memo field, or passed as an argument to another function.

Insert/overstrike:

When the user enters text in the edit mode, he can choose the insert or overstrike entry, toggling them by pressing the INSERT or ^V key. The application can set the mode using READINSERT() function or returning 22 from the UDF.

In insert mode, characters are entered into the text buffer at the current cursor position and the remainder of the text moves to the right. Insert mode is indicated in the status area. In overstrike mode, characters are entered at the current cursor position overwriting existing characters while the rest of the text buffer remains in its current position. The default mode is overstrike.

Tab characters: When the TAB key (09hex) is pressed, FlagShip's MEMOEDIT() inserts space characters up to the next tab position. The indent size of tabs is global for the entire memo editing; the default is four. MEMOEDIT() also converts tab characters to spaces in the entered string <expC1>.

Wrapping:

As the user enters text and the cursor reaches the edge of the MEMOEDIT() window, the current line wraps to the next line in the text buffer and a soft carriage return CHR(141)+CHR(10) is inserted into the text. If the <expN8> argument is specified, text wraps when the cursor position is the same as the argument value. To explicitly start a new line or paragraph, the user must press the ENTER or RETURN key, which generates a newline mark CHR(13)+CHR(10).

The **word wrapping** state can be changed by UDF returning 34. When word-wrap is on (the default), MEMOEDIT() inserts the soft carriage return at the closest word break (= space) to the window border or line length, whichever occurs first. When word-wrap is off, MEMOEDIT() scrolls text entry beyond the window definition until the end-of-line is reached.

Note that soft carriage returns may interfere with the result of display commands such as ? and REPORT FORM or processing with another word processor, like "vi". Using HARDCR(), MEMOTRAN() or STRTRAN() functions, the soft carriage return can be translated to the usual NEWLINE (or CR/LF) characters.

Paragraph reform: Pressing Ctrl-U or returning a 2 from the UDF reformats the memo text until a hard carriage (end-of-paragraph) or the end-of-memo is encountered. This happens regardless of whether word-wrap is on or off.

Cut and Paste:

In GUI mode, FlagShip supports the global X11 or Windows clipboard for exchanging/transfer keyboard data. You may copy and paste text via clipboard from/to other windows or applications on the screen, or from/to other/current GET field or MemoEdit() text.

To **copy** part of the MemoEdit text into clipboard, issue:

- mark the text by depressed left mouse button, then
- press the **Alt-C** or middle mouse button (both user modifiable)

To copy text from another application on screen to clipboard, use the corresponding key sequence of this application.

To **paste** clipboard at current text position, issue:

- press **Alt-V** or Shift + middle mouse button (both user modifiable)

When INSERT state is on, the pasted text from clipboard is inserted, otherwise the MemoEdit() text is partially overwritten by the text from clipboard (same as you would type it).

The copy and paste buttons or keys are user modifiable by assigning corresponding INKEY() value (see inkey.fh for manifests) to:

```
_aGlobSetting[GSET_G_N_MEMO_COPY1 ] := K_MBUTTONDOWN   // copy
_aGlobSetting[GSET_G_N_MEMO_COPY2 ] := K_ALT_C         // copy
_aGlobSetting[GSET_G_N_MEMO_PASTE1] := K_SH_MBUTTONDN  // paste
_aGlobSetting[GSET_G_N_MEMO_PASTE2] := K_ALT_V         // paste
```

When you assign NIL (which is the default setting in initio.prg), the corresponding `_aGlobSetting[GSET_G_N_GET_*]` value will be used (see the READ command) to behave same as in GET/READ. Assigning 0 (zero) will disable this setting and the default behavior is used.

Note that the common Ctrl-C and Ctrl-V keys are already assigned otherwise (PgDn and Insert), therefore Alt-C and Alt-V are pre- defined instead. You may need to assign other keys when these conflicts with Topbar menu or with your SET KEY redirection. In MS-Windows, you may probably prefer K_RBUTTONDOWN for paste; it is not set by default to avoid unintentional copying from the clipboard.

In Terminal i/o mode, similar functionality is provided (in Unix) via the "gpm" cut-and-paste console utility/daemon and FlagShip keyboard buffer by using it pre-defined keys and/or mouse buttons. To copy large strings, you probably may need to extend the buffer size by SET TYPEAHEAD.

Unicode:

In GUI mode, FlagShip supports also Unicode (UTF-8 and UTF-16). If the default font is set by `oApplic:Font:CharSet(FONT_UNICODE)` or SET GUICHARSET FONT_UNICODE, MemoEdit() displays also Unicode glyphs e.g. for far east languages. To edit text with Unicode glyphs, SET MULTIBYTE ON must be set as well. Predefined text needs to be stored in UTF-8 encoding (glyphs are usually one to four bytes ea chr(128..255)), or transformed from UTF-16 to UTF-8 by Utf16_Utf8(). To copy-and-paste external text in Unicode MemoEdit(), the text file must be encoded in UTF-8.

In Linux, you may need to set Unicode font, e.g. `SET FONT "mincho"` as well. See also example in <FlagShip_dir>/examples/unicode.prg and general Unicode description in section LNG.5.4.5

UDF usage:

The user defined function specified by <expC7> handles key exceptions and reconfigures special keys. The UDF is called at various times by MEMOEDIT(), generally in response to keys it does not recognize. In the user function, the program performs various actions, depending on the current MEMOEDIT() mode, then RETURNs a value telling MEMOEDIT() what to do next.

When the UDF argument is specified, MEMOEDIT() defines two classes of keys: non-configurable and key exceptions. When a non-configurable key is pressed, MEMOEDIT() executes it; otherwise a key exception is generated and UDF is called. When there are no keys left in the keyboard buffer for MEMOEDIT() to process, the user function is called once again with the state 0.

When MEMOEDIT() calls the UDF, it automatically passes three arguments:

- the "mode" of MEMOEDIT(), indicating the status depending on the last key pressed or the last action taken prior to executing the UDF,
- the current text buffer "line" (starting with 1), and
- the current text buffer "column" (starting with 0).

The following status values (modes) passed to the UDF are possible (Table 2):

Mode	memoedit.fh	Description
0	ME_IDLE	Idle, no more key pressed
1	ME_UNKEY	Unknown key, text unaltered
2	ME_UNKEYX	Unknown key, text altered
3	ME_INIT	Initialization mode

Immediately after MEMOEDIT() is invoked, the UDF is entered with mode 3. At this point, the UDF can set the required insert or wrap mode returning the relevant value. MEMOEDIT() then calls the UDF repeatedly with mode 3 until 0 is returned.

The idle state with mode 0 is reported once there is no pending key to process. It may generally be needed to update line and column number displays. Return codes from the UDF other than ME_DEFAULT (zero) are ignored.

MEMOEDIT() calls the user defined function whenever a key exception occurs. The exception keys are all the configurable keys (see Table 1) and other function keys. The UDF is entered with mode 1 or 2. Configurable keys are processed by returning 0 to execute the MEMOEDIT() default action. Returning a different value executes another key action, thereby redefining the key. The following table summarizes the possible return values from the UDF and their consequences (Table 3):

RETURN	memoedit.fh	Description
0	ME_DEFAULT	Perform default action
1..31	(K_xxx in inkey.fh)	Perform requested action corresponding to key value, see INKEY() values in appendix
32	ME_IGNORE	Ignore the current key
33	ME_DATA	Treat unknown key as data
34	ME_TOGGLEWRAP	Toggle word wrap mode
35	ME_TOGGLESCROLL	Toggle scroll mode
100	ME_WORDRIGHT	Perform word-right operation
101	ME_BOTTOMRIGHT	Perform bottom-right operat.
2001..2255	(2000+K_xxx defined in inkey.fh)	Insert character CHR(1..255) into the text

Tuning:

At start of MemoEdit(), all soft-carriage-return characters chr(141) + chr(10), used to mark word-wraps in previous editing session, are removed by default. You may replace them by hard-CR (e.g. to ensure same line wraps in different MemoEdit sizes) by assigning

```
_aGlobSetting[GSET_L_MEMOE_SOFTHARD] := .T.  // default = .F.
```

When pressing ESC key in non-read-only mode, a pop-up displays per default "Do you really want to abort the editing now ?". You may modify this behavior by invoking

```
Set(_SET_MEMOEDITWARN, nOn|lOn)
```

before calling MemoEdit(), where

nOn = 2 : always display warning and prompt (default)
1 : display warning and prompt only if text was changed

0 : don't display warning at all
lOn = .T. same as nOn = 2 (display warning and prompt)
= .F. same as nOn = 0 (don't display warnings)

When MemoEdit() ends, and the text was modified, all RETURNs in the text are transformed to CR+LF = chr(13)+chr(10). You may leave LF only by assigning

```
_aGlobSetting[GSET_L_MEMOE_HARDCRLF] := .F.  // default = .T.
```

Note that most printers will need CR+LF for new line.

Similarly, all word-wraps in changed text are marked by soft- carriage-return's chr(141)+chr(10) used e.g. in MemoLine(). You may remove all these soft-carriage-return characters from text by assigning

```
_aGlobSetting[GSET_T_L_MEMOE_SOFTCR] := .F.  // default = .T.
```

for Terminal i/o mode, and/or by

```
_aGlobSetting[GSET_G_L_MEMOE_SOFTCR] := .F.  // default = .T.
```

for application running in GUI mode. Alternatively, you may use HardCr() to convert them to CR+LF after finishing MemoEdit()

In Terminal i/o mode, you may display status marks for each visible text line by assigning

```
_aGlobSetting[GSET_T_A_MEMOE_SHOW] := array  // default = {}
```

where the <array> is either empty, or contain 4 to 5 elements:

array[1] is numeric column displacement to <expN5>
array[2..4] are strings displayed for std.line, splitted text, and hard-CR marked lines
array[5] is optional color specification, otherwise SetColor()
for example

```
_aGlobSetting[GSET_T_A_MEMOE_SHOW] := {1," ","+","<","G+"}
```

In GUI mode, MemoEdit shows ToolTip (by moving mouse over the box) saying "MemoEdit - a small text editor". You may assign anything else or disable it by assigning empty string "" to

```
_aGlobSetting[GSET_C_MEMOE_TOOLTIP] := "...any text..."
```

With SET MULTIBYTE ON, additional tunings are described in section CMD.SET MULTIBYTE

Example 1:

To display current memo field without editing:

```
USE my_dbf
SET CURSOR OFF
MEMOEDIT (my_memo, 6, 6, 14, 54, .F.)
SET CURSOR ON
```

Example 2:

To edit and save the memo field in database (.dbt):

```
USE my_dbf
@ 5,5 to 15, 55 double
@ 16,0 say "modify text end save by ^W or abort with ESC"
Set(_SET_MEMOEDITWARN, 1)    // warnings only on text change
REPLACE my_memo WITH MEMOEDIT (my_memo, 6, 6, 14, 54)
```

Example 3:

Edit Unix/Windows text file (see also Tuning section above):

```
LOCAL text
text := MEMOREAD ("mytext.txt ")
text := MEMOEDIT (text)                          // edit text
IF LASTKEY() == 23                               // saved by ^W
  text := STRTRAN (text,CHR(141)+CHR(10),"")  // remove wrap
  text := STRTRAN (text, CHR(13), "")         // UNIX convention
  MEMOWRIT ("mytext.txt ", text)
ENDIF
```

Example 4:

General purpose edit function, using an UDF:

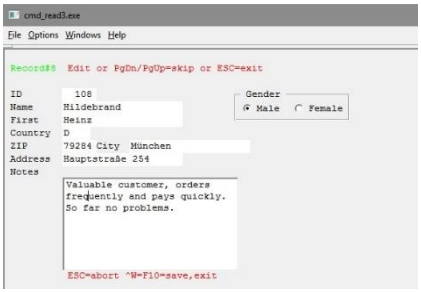
```
USE mydbf NEW SHARED
mytext = myedit (5,1, MAXROW(), MAXCOL()-1, memo, DBF())
WHILE !RLOCK(); END
REPLACE memo WITH mytext
UNLOCK

#include "memoedit.fh"
#include "inkey.fh"
#ifdef FlagShip
#  define K_REFORMAT 21
#else
#  define K_REFORMAT 2
#endif

FUNCTION myedit (ytop, xtop, ybot, xbot, text, title)
*****************************************************
LOCAL actscreen := SAVESCREEN (ytop, xtop, ybot, xbot)
PRIVATE msgcol := xbot-1, msgrow := ybot
@ ytop, xtop CLEAR TO ybot, xbot
@ ytop, xtop TO ybot, xbot DOUBLE
@ ytop, xtop +1 SAY "[" + title + "]"
@ ybot, xtop +1 SAY "ESC=abort ^W=F10=save,exit" +;
            " F2=word+ F3=bottom F4=wrap F5=scrl F6=key"
text := MEMOEDIT (text, ytop+1, xtop+1, ybot-1, xbot-1, ;
                  .T., "myedit_udf")
RESTSCREEN (ytop, xtop, ybot, xbot, actscreen)
RETURN text

FUNCTION myedit_udf (mode, line, column)
****************************************
LOCAL  key  := LASTKEY(), retval := ME_DEFAULT
LOCAL  getlist := {}, screen, num := 0
STATIC init := 0, wrap := .T., scrl := .F.
@ msgrow,msgcol-11 TO msgrow,msgcol DOUBLE

DO CASE
CASE mode == ME_IDLE
   @ msgrow, msgcol-7 SAY "waiting"
CASE mode == ME_UNKEY .or. mode == ME_UNKEYX
   DO CASE
   CASE key == K_ESC
*     IF .not. MemoScoreEsc()              // see scor_mem.prg
```

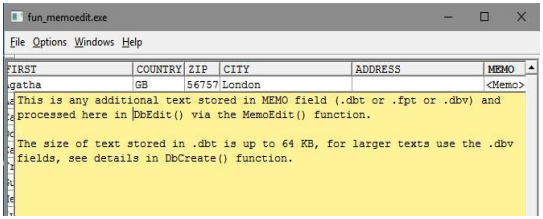
```
      IF ALERT("Abort edit ?", {"No", "Yes"}) == 1
         retval = ME_IGNORE
      ENDIF
   CASE key == K_F2
      retval := ME_WORDRIGHT
   CASE key == K_F3
      retval := ME_BOTTOMRIGHT
   CASE key == K_F4
      wrap := !wrap
      retval := ME_TOGGLEWRAP
      @ 0,51 say IF (wrap, " <wrap> ", "<nowrap>")
   CASE key == K_F5
      scrl := !scrl
      retval := ME_TOGGLESCROLL
      @ 0,44 say IF (scrl, "<scrl>", SPACE(6))
   CASE key == K_F6
      screen := SAVESCREEN (ybot, 0, ybot, 50)
      @ ybot, 0 CLEAR TO ybot,50
      @ ybot,0 SAY "Enter the ASCII value 1..255" ;
               GET num PICT "999" RANGE 0,255
      READ
      RESTSCREEN (ybot, 0, ybot, 50, screen)
      retval := IF (num > 0, 2000 + num, retval)
   CASE key == K_REFORMAT
      retval := 2
   CASE key == K_CTRL_W .or. key == K_F10
      retval := K_CTRL_W
   OTHERWISE
      @ msgrow,msgcol-11 SAY "CHR("+STR(key,3)+") ??"
   ENDCASE
CASE mode == ME_INIT
   IF init == 0
      init := 1
      wrap := .F.; retval := ME_TOGGLEWRAP      // word wrap off
      @ 0,51 say IF (wrap, " <wrap> ", "<nowrap>")
   ELSEIF init == 1
      init := 2
      scrl := .T.; retval := ME_TOGGLESCROLL    // scroll on
      @ 0,44 say IF (scrl, "<scrl>", SPACE(6))
   ELSEIF init == 2
      init := 3; retval := K_INS                // insert mode
   ENDIF
ENDCASE
RETURN retval
```

Example 5:

See CMD.READ example 3 for edit/skip simplified address database (with ASCII/PC8/OEM charset), using also RADIOGROUP and MEMOEDIT().

For browsing and editing, see FUN.DBEDIT() Example 1 where the MemoEdit() is used to view or edit MEMO fields:





Compatibility:

Unlike Clipper, the cursor movement keys in FlagShip always have the same corresponding control keys. There is a difference to Clipper in the handling of control character input, cf. table 1 above and INKEY() table in the appendix. To reformat the paragraph, both return codes 2 or 21 are supported. The handling of the TAB key pressed differs in VFS and C5, since the tab indent is variable in Unix.

The user modifiable MEMOEDIT() messages are available in the file <FlagShip_dir>/system/scor_mem.prg, and in memedithand.prg keyboard handler.

VFS supports soft carriage-return in the way C5 handles it. When the text is reformatted, the CHR(141) + CHR(10) is replaced by one space character or removed, see also Tuning above.

For paragraph reformatting, FlagShip supports both the UNIX newline CHR(10) and the DOS CR+LF. On output, CR+LF (CHR(13)+CHR(10)) is produced. To convert ASCII text to the UNIX convention, use additionally STRTRAN(string, CHR(13,10), CHR(10)) before storing it to the file, or the global tuning assignment.

Unicode is supported in GUI mode by VFS7 and later.

Classification:

programming (and database access, if memo field is used)

Include:

<FlagShip_dir>/include/memoedit.fh <FlagShip_dir>/include/inkey.fh

Related:

ACHOICE(), DBEDIT(), HARDCR(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT(), Set(_SET_MEMOEDITWARN), memedithand.prg

MEMOLINE ()

Syntax:

```
retC = MEMOLINE (expC1, [expN2], [expN3], [expN4],
                 [expL5])
```

Purpose:

Extracts a formatted line of text from a character string or a memo field.

Arguments:

<**expC1**> is the source string or memo field.

Options:

<**expN2**> is the number of characters per line. The default is 79, the valid range is 4 to 254.

<**expN3**> is the line number which should be extracted. The default is 1.

<**expN4**> is tabulator stop width. The default is 4.

<**expL5**> switches the word wrapping on/off. TRUE enables word wrapping, FALSE disables it. The default value is TRUE. If the wrap is on and the line length indicated breaks the line in the middle of a word, that word is not included as part of the return value <retC>.

If the wrapping <expL5> is on, and the line has fewer characters than the indicated length <expN2>, the return value is padded with blanks. If the wrapping is off, only the shorter value of the line length or the number of characters specified by <expN2> is re- turned; the next extracted line begins behind the next newline (LF or CR/LF).

Returns:

<**retC**> is a character string with a copy of the specified line. If the line number <expN3> is greater than the total number of lines in <expC1>, null string "" is returned.

Description:

MEMOLINE() is often used with MLCOUNT() to extract lines of text from character strings and memo fields based on the number of characters per line. Normally, the line length, tab width and wrapping correspond to the setting of MEMOEDIT().

MEMOLINE() also handles strings and memos formatted by MEMOEDIT() as plain ASCII text in the Unix and DOS/Windows convention.

Example:

```
To put  all the text on the same page, the required number of lines
is determined using MLCOUNT() before printing.

LOCAL width := 120, maxrow := 75, lines, count
FIELD note
USE address
SET PRINTER ON
lines = MLCOUNT (note, width)
FOR count = 1 TO lines
   ? MEMOLINE (note, width, count)
   IF PROW() >= maxrow
      EJECT
   ENDIF
NEXT
SET PRINTER OFF
```

Classification:

programming

Related:

HARDCR(), MEMOEDIT(), MEMOTRAN(), MEMOWRIT(), MLCOUNT(), MLPOS()

MEMOREAD ()

Syntax:

`retC = MEMOREAD (expC1, [expL2])`

Purpose:

Reads a text file from the disk into a character memory variable or a memo field.

Arguments:

<**expC**> is the name of the text file including the extension and optionally a path. If no path is specified, the file is searched in the current directory. If not found, the UNIX and Windows environment PATH will be examined. MEMOREAD() does not use the SET DEFAULT or SET PATH to search for the file.

<**expL2**> is optional logical value. If true (.T.), MEMOREAD() will retry the read on failure. The default is .F. = no retry.

Returns:

<**retC**> is a character string with a copy of the contents of the specified text file. In FlagShip, there is no size limit to the file read. If the file cannot be found, a null string "" is returned and FERROR() is set.

Description:

MEMOREAD() reads an ASCII file into memory where it can be manipulated as a character string (e.g. using MEMOEDIT()) or stored to a memo field. The user must have at least the "r" (read) access right to the file.

The inverse function of MEMOREAD() is MEMOWRIT(). To read the text file line-by-line, you may preferably use FreadTxt().

Multiuser:

To read strings or memo fields in a multiuser environment, use FLOCKF() together with FREAD() or FREADSTR(). Note that MEMOREAD() cannot be influenced by FLOCKF().

Example:

```
LOCAL note := MEMOREAD("mynote.txt")
IF FERROR() == 0                      // success?
   USE mydbf
   REPLACE memofld WITH MEMOEDIT(note)
   MEMOWRIT ("note.txt", memofld)
ENDIF
```

Classification:

system, file access

Related:

MEMOWRIT(), FERROR(), FREADTXT(), MEMOEDIT(), MEMOLINE(), MEMOTRAN(), MLCOUNT(), MLPOS(), HARDCR(), FLOCKF(), FS_SET()

MEMORY ()

Syntax:

```
retN = MEMORY (expN)
```

Syntax 2:

```
retN = MEMORY (1, expN2)
```

Purpose:

Determines total, used or free memory space, largest assignable string, swap/paging space, and buffering.

Arguments:

<**expN**> is a numeric value specifying the required action:

expN	Description/Action	Clipper	Windows	Linux
NIL:	same as expN = 0			
0:	Free Variable Space (heap)	retN	retN	retN
1:	Largest String (*see <expN2>)	64	retN	retN
2:	Available RUN Memory	retN	retN	retN
3:	Available Virtual Memory	retN	retN	retN
4:	Free Expanded Memory	retN	99999	99999
101:	Total physical memory	retN	retN	retN
102:	Segments in Fixed Memory/Heap	retN	1	1
103:	Free Swap Memory	retN	retN	retN
104:	Free Conventional Memory	retN	retN	retN
105:	Used Expanded Memory	0	0	0
106:	Total Swap Memory	0	-1	retN
107:	Buffers	0	-1	retN

<**expN2**> in Syntax 2 is the string size (in bytes, up to 2000000000) to be checked for sufficient space on heap.

Returns:

<**retN**> is a numeric value representing the requested status in KB (kilobytes), see above table. If the requested mode is not available, the above displayed default value is returned instead. On error, -1 is returned.

With **Syntax 2**, <retN> is either <expN2> +1 which signals success, i.e. that the requested string size of <expN2> bytes (note: not kilobytes as otherwise) can be assigned, or is zero when not enough free continuous heap space is available. In case of zero return, an subsequent assignment of string size <expN2> would raise RTE 301 = no memory available.

Description:

MEMORY() is a compatibility function to the xBase DOS derivate. As opposite to limited DOS memory, all 32 and 64bit operating systems (like Windows, Linux or commercial Unix) uses virtual memory = real RAM plus paging/swapping, which results in 2GB or more space.

All FlagShip variables are stored on dynamic heap. With Syntax 1, you may determine the total or available memory (heap) size, returned by system (or kernel). On Linux, these values are similar to external invocation of free, top or cat /proc/meminfo. On Windows, system call returns the corresponding value.

With MEMORY(1), the returned value reports total available space, which however can be fragmented. Therefore assignment of large string may fail even if the returned value is large enough. **Syntax 2** allows you to check unfragmented heap to fit requested string size.

Note, the FlagShip variable can contains strings up to 2 Gigabytes long, but only up to 64 Kilobytes can be stored in the database memo or character field.

Caution: since this function is expensive, use it with care and only if required. Heavy use within a loop like in the example below may slow down the execution speed.

Example:

```
 ? "Free memory at program start:", ltrim(memory()), "KB =", ;
   ltrim(memory() / 1024), "MB of total",ltrim(memory(3) /
1024),"MB"
 wait
 size := 1024 * 1024           // 1 MB
 str  := replicate("x", size)  // 1 MB string
 for ii := 1 to 30000          // try to allocate 30000 * 1MB = 30
GB
    idx := strzero(ii,5)
    var&idx := str             // ea 1 MB string in
var00001..var30000
    alloc := ii * size / 1024  // in KB
    avail := memory()          // in KB
    /*
     * You will detect that the available memory may temporarily
grow
     * when swapping/paging starts.  To check swapping, run 'top' in
     * separate Linux console, or Task Manager in Windows.
     */
    ? "Allocated:",  ltrim(alloc), "KB =", ltrim(alloc / 1024), ;
      "MB -- free:", ltrim(avail), "KB =", ltrim(avail / 1024), "MB"
    if memory(106) > 0
       ?? ", swap used=",ltrim((memory(106)-memory(103)) /
1024),"MB"
    elseif memory(103) > 0
       ?? ", avail.swap=",ltrim(memory(103) / 1024),"MB"
    endif
    /*
     * check for availability of 1MB continuous heap space to
     * avoid RTE = not enough memory.
     */
    if memory(1, size) < size   // note: in bytes, not in KB
       ? "-- not enough continuous heap space, exiting loop"
       exit
    endif
 next
 wait "done ..."
 quit  // at program end, all used memory is freed
```

Classification:

programming

Compatibility:

Syntax 2 is FlagShip extension. Functionality extended in VFS7, previous FlagShip versions returned 640KB with Memory(2), and 64KB otherwise. In Clipper, only values 0,1,2 are documented, Memory(1) returns max 64 KB, Memory(0) usually max 345 KB. FoxPro/VFP always returns 640 KB for Memory() or SYS(12), VFS7 returns correct value which us usually more than 1 GB (i.e. more than 1048576 KB) at the program begin.

Related:

LEN()

MEMOTRAN ()

Syntax:

`retC = MEMOTRAN (expC1, [expC2], [expC3])`

Purpose:

Replaces the line terminating characters LF, CR/LF and soft-CR/LF in a memo field or a string with user defined characters.

Arguments:

<**expC1**> is the source string or memo field.

Options:

<**expC2**> is the character or string which should replace the CR/LF (CHR(13)+CHR(10) e.g. in DOS file, memo field) or LF only (CHR(10) in a UNIX file) within <expC1>. If not specified, the default is semicolon ";".

<**expC3**> is the character or string, which should replace the soft-CR/LF CHR(141) + CHR(10) in <expC1>. If not specified, a space " " will be used as default.

Returns:

<**retC**> is the translated string.

Description:

MEMOTRAN() transforms the paragraph mark (CR/LF) or the word-wrap mark (soft-CR/LF) used in MEMOEDIT() to other, user defined characters. Also, the NEWLINE mark of DOS (CR/LF) or Unix (LF) ASCII files can be translated.

To translate the soft-CR/LF only, HARDCR() or STRTRAN() may be used instead.

Example:

```
LOCAL notetext, handle
USE address
notetext = MEMOTRAN (notefield, " ", chr(10))
handle = FCREATE ("notes.data")
IF handle = 0
   FWRITE (handle, notetext)
   FCLOSE (handle)
ENDIF
```

Classification:

programming

Compatibility:

The LF translation is supported by FlagShip only.

Related:

HARDCR(), STRTRAN(), MEMOEDIT(), MEMOLINE()

MEMOWRIT ()

Syntax:

retL = MEMOWRIT (expC1, expC2, [expL3])

Purpose:

Writes a character string to the specified text file.

Arguments:

<**expC1**> is the name of the text file including the extension and optionally a path, if required. If a path is not specified, the file is written to the current directory and the current SET DEFAULT does not apply for MEMOWRIT().

<**expC2**> is the character string, the contents of which are to be written to the specified file.

<**expL3**> is optional logical value. If true (.T.), MEMOWRIT() will retry the write on failure. The default is .F. = no retry.

Returns:

<**retL**> is logical TRUE if the operation was successful.

Description:

MEMOWRIT() writes a character string or memo field to an ASCII file. The user must have at least the "rw" access right to the file and the "wx" right to the directory. If the write operation was not successful, <retL> returns FALSE and FERROR() is set.

The inverse function of MEMOWRIT() is MEMOREAD().

Multiuser:

To write strings or memo fields in a multiuser environment, use FLOCKF() together with FWRITE(). Note that MEMOWRIT() cannot be influenced by FLOCKF().

Example:

```
helptext = MEMOREAD ("help.txt")
helptext = MEMOEDIT (helptext)
IF LASTKEY() # 27
   IF .not. MEMOWRIT ("help.txt", helptext)
      ? "The file help.txt cannot be saved, error", ;
        LTRIM(STR(FERROR())), "occurred."
      WAIT
   ENDIF
ENDIF
```

Classification:

system, file access

Related:

MEMOREAD(), FERROR(), MEMOEDIT(), MEMOLINE(), MEMOTRAN(), HARDCR(), MLCOUNT(), MLPOS(), FLOCKF(), FS_SET()

MEMVARBLOCK ()

Syntax:

retB = MEMVARBLOCK (expC)

Purpose:

Returns a set/retrieve code block for a given memory variable.

Arguments:

<**expC**> is the name of the PRIVATE or PUBLIC variable referred to by the set/retrieve block, specified as a character string. LOCAL, STATIC or typed variables cannot be used, since they cannot be determined using the macro evaluation of <expC>.

Returns:

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

Description:

The code block created by MEMVARBLOCK() has two operations depending on whether an argument is passed to the code block when it is evaluated. If evaluated with an argument, it assigns the value of the argument to the <expC> variable. Otherwise, the evaluated code block returns the value of <expC>.

Using the code block instead of a similar macro evaluation will increase process speed.

Example:

```
LOCAL  block1, block2
PRIVATE myvar := "any text"
block1 := {|par| IF (par==NIL, myvar, myvar := par)}
block2 := MEMVARBLOCK ("myvar")         // equivalent to block1

? EVAL (block2)                         // any text
? EVAL (block2, "new text")             // new text
? myvar                                 // new text
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

FIELDBLOCK(), FIELDWBLOCK()

MHIDE ()

Syntax:

`NIL = Mhide ( )`

Purpose:

Hide the mouse pointer/cursor

Returns:

Mhide() always returns NIL

Description:

Mhide() hides the mouse pointer. This function can be used together with Mshow() when updating the screen or signaling the user not to use mouse at this time.

Compatibility:

New in FS5, available in CL53

Related:

Mshow(), Mrow(), Mcol(), MSetCursor()

MIN ()

Syntax:

```
retN = MIN (expN1, expN2, [... expNn])
retD = MIN (expD1, expD2, [... expDn])
```

Purpose:

Returns the smaller of two numeric or date values.

Arguments:

<**expN1**> ... <**expNn**> are the numeric expressions to be compared.

<**expD1**> ... <**expDn**> are the date expressions to be compared.

Returns:

<**retN**> or <**retD**> is the smaller of the two or more arguments.

Description:

MIN() can be used when the result of an expression has to be kept below a maximum value.

The inverse operation of MIN() is MAX().

Example:

```
? MIN (100, 88)                      &&   88
? MIN (88, 100)                      &&   88
? MIN (-88, -100)                    && -100
? DATE(), MIN (DATE(), DATE()-40)    && 09/05/93   07/27/93
```

Classification:

programming

Compatibility:

The support of more than two arguments is new in FS5

Related:

MAX(), MinMax()

MINMAX ()

Syntax:

`retND = MinMax (expND1, expND2, expND3)`

Purpose:

Checks and if required set the return value to the lower or upper boundary, testing expND1 <= expND2 <= expND3

Arguments:

<**expND1**>	is numeric or date expression with the lower boundary
<**expND1**>	is numeric or date expression the checked value
<**expND3**>	is numeric or date expression with the upper boundary

Returns:

<**retND**> is the result of this comparison

Description:

This function is equivalent to Max(expND1, Min(expND2, expND3))

Compatibility:

New in FS5

Related:

Min(), Max(), Between()

MLCOUNT ()

Syntax:

`retN = MLCOUNT (expC1, [expN2], [expN3], [expL4])`

Purpose:

Counts the number of lines to be printed from a character string or a memo field.

Arguments:

<**expC1**> is the source string or memo field to count.

Options:

<**expN2**> is the number of characters per line. The valid range is 4 to 254. If not given, the default is 79.

<**expN3**> is the tabulator stop width, the default is 4.

<**expL4**> toggles word wrapping on and off. TRUE (the default) enables word wrap, FALSE disables it.

If the wrapping <expL4> is on, and the line width breaks a word, it will be word wrapped to the next line and the next line will begin with that word. If the wrapping is off, the line ends with the newline mark or the line width; the next line begins behind the newline (LF or CR/LF).

Returns:

<**retN**> is the number of lines in <expC1> depending on the line length, the tabulator stop width and the word wrapping.

Description:

MLCOUNT() is often used with MEMOLINE() to print character strings and memo fields based on the number of characters per line. MLCOUNT() returns the final line count used as the end criteria for the line extracting and printing. Normally, the line length, tab width and wrapping corresponds to the setting of MEMOEDIT().

MLCOUNT() also handles strings and memos formatted by MEMOEDIT(), as a plain ASCII text in the Unix and DOS/Windows convention.

Example 1:

```
lines = MLCOUNT (note, 80)
FOR i = 1 TO lines
   ? MEMOLINE (note, 80)
NEXT
```

Example 2:

see also example in MEMOLINE() and MLCTOPOS().

Classification: programming

Related:

MEMOLINE(), MLPOS(), MEMOTRAN(), HARDCR(), MEMOEDIT(), MEMOREAD(), MEMOWRIT()

MLCTOPOS ()

Syntax:

retN = MLCTOPOS (expC1, expN2, expN3, expN4, [expN5], [expL6])

Purpose:

Returns byte position of a formatted string based on line and column position.

Arguments:

<**expC1**> is the text string to scan.

<**expN2**> is the number of characters per line. The valid range is 4 to 254.

<**expN3**> is the line number counting from 1.

<**expN4**> is the column number counting from 0.

Options:

<**expN5**> is the tabulator stop width. The default is 4.

<**expL6**> is the word wrap flag. If not specified, the default is TRUE.

Returns:

<**retN**> is the byte position within <expC1> counting from 1 up. If the byte is out of range, 0 is returned.

Description:

MLCTOPOS() determines the byte position that corresponds to a particular line and column within the formatted text. Normally, the line length, tab width and wrapping correspond to the setting of MEMOEDIT(). The returned value may be used as a argument in SUBSTR() or other string functions. MLCTOPOS() for the column 0 is equivalent to MLPOS().

Example:

```
*               ....5....0....5....0....5....0
LOCAL text := "This is a long text containing several " + ;
              "lines in MEMOEDIT or MEMOLINE using a " +;
              "width of 25characters."

? MLCOUNT  (text, 20)                        // 5
? MLCTOPOS (text, 20, 3, 2)                  // 23
? SUBSTR (text, MLCTOPOS (text,25,2,0), 12)  // containing s
? SUBSTR (text, MLCTOPOS (text,25,2,2), 12)  // containing sev
? SUBSTR (text, MLPOS (text,25,3), 12)       // in MEMOEDIT
```

Classification: programming

Compatibility: available in FS and C5 on*ly*

Related:

MPOSTOLC(), MLPOS(), MLCOUNT(), MEMOLINE(), MEMOEDIT()

MLEFTDOWN ()

Syntax:

```
retL = MlefttDown ( )
```

Purpose:

Determine the status of the left mouse button

Returns:

<**retL**> is .T. if the mouse's left button is currently pressed, otherwise .F.

Description:

MlefttDown() determines the button press status of the left mouse button.

Note: you may determine the mouse click also from

```
key := Inkey(n, INKEY_ALL_BUT_MOVE)
```

where the <key> returns more information e.g. double click, up or down press of the left, middle or right button etc.

Compatibility:

New in FS5, available in CL53

Related:

Mpresent(), MrightDown(), Mrow(), Mcol(), MSetCursor(), Inkey()

MLPOS ()

Syntax:

retN = MLPOS (expC1, expN2, expN3, [expN4], [expL5]))

Purpose:

Determines the position of a line inside a string or a memo field.

Arguments:

<**expC1**> is the source string or memo field.

<**expN2**> is the number of characters per line. The valid range is 4 to 254.

<**expN3**> is the required line number counting from 1.

Options:

<**expN4**> is the tabulator stop width. The default is 4.

<**expL5**> is the word wrap flag. If not specified, the default is TRUE.

Returns:

<**retN**> is a number representing the byte position of the required line beginning. If the line does not exist in <expC1>, the total length of the source string will be returned.

Description:

MLPOS() determines the byte position that corresponds to a particular line within the formatted text. Normally, the line length, tab width and wrapping correspond to the setting of MEMOEDIT(). The returned value may be used as a argument in SUBSTR() or other string functions. MLPOS() is equivalent to MLCTOPOS() for the column 0.

Example:

```
*              ....5....0....5....0....5....0
LOCAL text := "This is a long text containing several " + ;
              "lines in MEMOEDIT or MEMOLINE using a " +;
              "width of 25characters."

? MLCOUNT  (text, 20)                          // 5
? MLPOS  (text, 20, 3)                         // 21
? SUBSTR (text, MLCTOPOS (text,25,2,0), 12)  // containing s
? SUBSTR (text, MLPOS (text,25,2), 12)       // containing s
```

Classification:

programming

Compatibility:

The arguments 4 and 5 are new in FS and C5.

Related:

MLCTOPOS(), MLCOUNT(), MEMOLINE(), MEMOEDIT()

MOD ()

Syntax:

`retN = MOD (expN1, expN2)`

Purpose:

Returns dBASE III+ modulus of two numbers.

Arguments:

<**expN1**> is the dividend of the division operation.

<**expN2**> is the divisor of the division operation.

Returns:

<**retN**> represents the remainder of <expN1> divided by <expN2>.

Description:

The result values of MOD() correspond to dBASE, but differs slightly from the default % modulus operator (and the equivalent mathematical operation):

expN1	expN2		%	MOD()
3	0	.....	Error	3
3	-2	.....	1	-1
-3	2	.....	-1	1
-3	0	.....	Error	-3
-1	3	.....	-1	2
-2	3	.....	-2	1
2	-3	.....	2	-1
1	-3	.....	1	-2

Example:

```
? 10  % 3,   MOD (10,  3)              // 1     1.00
?  0  % 3,   MOD ( 0,  5)              // 0     0.00
?  5  % 2,   MOD ( 5,  2)              // 1     1.00
? -5  % 2,   MOD (-5,  2)              // -1    1.00
? 5.5 % 2,   MOD (5.5, 2)              // 1.50  1.50
? 5.5 % 2.2, MOD (5.5, 2.2)            // 1.10  1.10
```

Classification:

programming

Related:

The % operator

MONTH ()

Syntax:

retN = MONTH ([expD])

Purpose:

Extracts the month of the year from a date value.

Arguments:

<expD> is the date value. If not specified, the current system date() is used.

Returns:

<**retN**> is a numeric value in the range from 1 to 12. If a null or invalid date is specified, zero is returned.

Description:

MONTH() is used where the numeric month value is needed as a part of some calculation. Other date extracting functions are DAY() and YEAR(). The related CMONTH() function returns the name of the month as a character string (in English or other human languages according to the FS_SET("load") setting).

Example:

```
? dd := DATE()                          // 08/15/93
? MONTH()                               // 8
? MONTH(DATE())                         // 8
? MONTH(DATE()) + 20                    // 9
? DAY(dd), MONTH(dd), YEAR(dd)          // 15  8  1993
? CMONTH(DATE() -30)                    // July
FS_SET ("load", 1, "FSsortab.ger")
FS_SET ("setl", 1)
? CMONTH(DATE() -30)                    // Juli
```

Classification:

programming

Compatibility:

Clipper requires the <expD> value, it does not use current date() when <expD1> parameter is empty.

Related:

CDOW(), CMONTH(), CTOD(), DATE(), DATEVALID(), DAY(), DOW(), DTOC(), DTOS(), YEAR(), FS2:AddMonth(), FS2:Cmonth2(), FS2:CtoMonth(), FS2:NtoCmonth(), FS2:IsLeap(), FS2:BoM(), FS2:LastDayOfM(), FS2:Week(), FS2:WoM(), FS2:Quarter()

MPOSTOLC ()

Syntax:

retA = MPOSTOLC (expC1, expN2, expN3, [expN4], [expL5])

Purpose:

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

Arguments:

<**expC1**> is the text string to scan.

<**expN2**> is the number of characters per line. The valid range is 4 to 254.

<**expN3**> is the byte position counting from 1.

Options:

<**expN4**> is the tabulator stop width. The default is 4.

<**expL5**> is the word wrap flag. If not specified, the default is TRUE.

Returns:

<**retA**> is a two-element array containing the line and the column values for the specified byte position <expN3>:

retA[1] contains the line number counting from 1, retA[2] is the column of the line, counting from 0.

Description:

MPOSTOLC() determines the formatted line and column corresponding to a particular byte position within a string <expC1>. Normally, the line length, tab width and wrapping correspond to the setting of MEMOEDIT(). The inverse function of MPOSTOLC() is MLCTOPOS().

Example:

```
*              ....5....0....5....0....5....0
LOCAL text := "This is a long text containing several " + ;
              "lines in MEMOEDIT or MEMOLINE"
? MLCTOPOS (text, 20, 3, 2)                    // 42
? SUBSTR (text, MLCTOPOS (text,25,2,1), 12)    // ontaining se
currpos := MPOSTOLC (text,20,22)
? currpos[1], currpos[2]                       // 2  1
```

Classification:

programming

Compatibility:

Available in FS and C5 only.

Related:

MLCTOPOS(), MLPOS(), MLCOUNT(), MEMOLINE(), MEMOEDIT()

MPRESENT ()

Syntax:

```
retL = Mpresent ( )
```

Purpose:

Determine if a mouse is present

Returns:

<**retL**> is .T. if mouse is present, otherwise .F.

Description:

This function is used to determine whether a mouse is available or not. Applicable in GUI mode only.

Compatibility:

New in FS5, available in CL53

Related:

Mcol(), Mrow(), MsetCursor(), MsetPos(), Mhide(), Mshow(), Mstate(), MsetCursor(), MsaveState(), MrestState(), MleftDown(), MrightDown()

MRESTSTATE ()

Syntax:

```
NIL = MrestState ( )
```

Purpose:

Re-establish the previous state of a mouse

Arguments:

<**expC1**> is the saved mouse cursor state from MsaveState().

Returns:

This function always returns NIL

Description:

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

Compatibility:

New in FS5, available in CL53

Related:

MsaveState(), Mpresent(), Mhide(), Mrow(), Mcol(), MSetCursor()

MRIGHTDOWN ()

Syntax:

```
retL = MrightDown ( )
```

Purpose:

Determine the status of the right mouse button

Returns:

<**retL**> is .T. if the mouse's right button is currently pressed, otherwise .F.

Description:

MlefttDown() determines the button press status of the left mouse button.

Note: you may determine the mouse click also from

```
key := Inkey(n, INKEY_ALL_BUT_MOVE)
```

where the <key> returns more information e.g. double click, up or down press of the left, middle or right button etc.

Compatibility:

New in FS5, available in CL53

Related:

Mpresent(), MleftDown(), Mrow(), Mcol(), MSetCursor(), Mstate(), Inkey()

MROW ()

Syntax:

```
retN = Mrow ([expL1], [expL2])
```

Purpose:

Determine the mouse cursor's user-screen row position

Arguments:

<**expL1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col,otherwise the current SET PIXEL status is used.

<**expL2**> apply for row/col coordinates only. If .F., the return value as integer (default), .T. returns decimal fraction.

Returns:

<**retN**> is the current mouse cursor row position. If <expL1> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

Description:

The mouse support is available in GUI mode only. In the Terminal and Basic i/o mode, 0 is returned.

Compatibility:

New in FS5, available in CL53

Related:

Mpresent(), Mcol(), MsetCursor(), MsetPos(), Mhide(), Mshow(), MsetCursor(), MsaveState(), MrestState(), MleftDown(), MrightDown(), Mstate(), Row(), MaxRow()

MSAVESTATE ()

Syntax:

```
retC = MsaveState ( )
```

Purpose:

Save the current state of a mouse

Returns:

<**retC**> is a character string that describes the mouse's current state. This string can be passed later to MresrState().

Description:

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

Compatibility:

New in FS5, available in CL53

Related:

Mpresent(), MrestState(), Mrow(), Mcol(), MSetCursor(), Mstate()

MSETCURSOR ()

Syntax:

```
retL = MsetCursor ([expNL1])
```

Purpose:

Determine or set mouse visibility and/or shape.

Arguments:

<**expL1**> if .T., the mouse visibility is enabled (when mouse is present). If specified .F., the mouse cursor is hidden.

<**expN1**> is the required mouse shape. The CURSOR_* constants are defined in mouse.fh

mouse.fh constant	value	Description
CURSOR_ARROW	-1	standard arrow cursor
CURSOR_UPARROW	-12	upwards arrow
CURSOR_CROSS	-8	crosshair
CURSOR_WAIT	-9	hourglass/watch
CURSOR_IBEAM	-11	i-beam (I)
CURSOR_SIZE_VER	-2	vertical resize
CURSOR_SIZE_HOR	-3	horizontal resize
CURSOR_SIZE_RDIAG	-5	diagonal resize (/)
CURSOR_SIZE_LDIAG	-4	diagonal resize (\)
CURSOR_SIZE_ALL	-13	all directions resize
CURSOR_INVISIBLE	-17	blank/invisible cursor
CURSOR_SPLITVER	-14	vertical splitting
CURSOR_SPLITHOR	-3	horizontal splitting
CURSOR_HAND	-6	a pointing hand
CURSOR_FORBIDDEN	-16	forbidden action cursor
CURSOR_DEFAULT	-1	default = CURSOR_ARROW

If no argument is passed, only the current state is returned.

Returns:

<**retL**> reports the current mouse cursor visibility

Description:

MsetCursor() either determines the current mouse cursor visibility, enables or disables the visibility, or set new cursor shape. To set the shape of text cursor in GUI mode, use SetGuiCursor() and/or SET GUICURSOR ON/TO.

Example:

```
an example is available in .../examples/guicursor.prg
```

Compatibility: New in FS5, available in CL53 which supports <expL1> only

Related: Mpresent(), Mrow(), Mcol(), Mstate(), MsetPos(), Mshow(), SetGuiCursor()

MSETPOS ()

Syntax:

NIL = MsetPos (expN1, expN2, [expL3])

Purpose:

Set a new position for the mouse cursor

Arguments:

<**expN1**> is the row/y coordinate

<**expN2**> is the column/x coordinate

<**expL3**> is a pixel specification. If .T., the <expN1> and <expN2> are given in pixel, if .F. in row/col, otherwise the current SET PIXEL status is used. If <expL3> is not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

Returns:

This function always returns NIL

Description:

MsetPos() moves the mouse cursor to a new position on the screen. After the mouse cursor is positioned, Mrow() and Mcol() are updated accordingly. To control the visibility or shape of the mouse cursor, use MsetCursor()

Compatibility:

New in FS5, available in CL53 which supports the first 2 params

Related:

Mpresent(), Mcol(), Mrow(), MsetCursor(), Mstate()

MSHOW ()

Syntax 1:

```
retN = Mshow ( )
```

Syntax 2:

```
retN = Mshow (expN1, expN2, expN3)
```

Syntax 3:

```
retN = Mshow (expN3)
```

Purpose:

Display the mouse pointer

Arguments:

<**expN1**> is the new mouse row coordinate.

<**expN2**> is the new mouse column coordinate.

If SET PIXEL or SET COORD is not set, the GUI row/col coordinates are calculated from current SET FONT (or oApplic:Font)

<**expN3**> is the required mouse shape. The CURSOR_* and LLM_* constants are defined in mouse.fh

mouse.fh constant	Clipper constant	Description
CURSOR_ARROW	LLM_CURSOR_ARROW	standard arrow cursor
CURSOR_UPARROW		upwards arrow
CURSOR_CROSS	LLM_CURSOR_CROSS	crosshair
CURSOR_WAIT	LLM_CURSOR_WAIT	hourglass/watch
CURSOR_IBEA		ibeam/text entry
CURSOR_SIZE_VER		vertical resize
CURSOR_SIZE_HOR		horizontal resize
CURSOR_SIZE_RDIAG		diagonal resize (/)
CURSOR_SIZE_LDIAG		diagonal resize (\)
CURSOR_SIZE_ALL		all directions resize
CURSOR_INVISIBLE		blank/invisible cursor
CURSOR_SPLITVER		vertical splitting
CURSOR_SPLITHOR		horizontal splitting
CURSOR_HAND	LLM_CURSOR_HAND	a pointing hand
CURSOR_FORBIDDEN		forbidden action cursor

Returns:

<**retN**> is the current mouse shape at the time of this call

Description:*a*

Mshow() is a combination of MsetCursor() and MsetPos()

The mouse support is available in GUI mode only. In the Terminal and Basic i/o mode, the call is ignored and 0 is returned.

Compatibility:

New in FS5, available in CL53

Related:

Mpresent(), MsetCursor(), MsetPos(), Mcol(), Mrow(), Mstate()

MSTATE ()

Syntax:

```
retA = Mstate ( )
```

Purpose:

Return the current mouse state

Returns:

<**retA**> is one-dimensional array containing:

Pos	mouse.fh constant	Clipper constant	Description	Note
[1]	CURSOR_MSTATE_X	LLM_STATE_X	X position	(1)
[2]	CURSOR_MSTATE_Y	LLM_STATE_Y	Y position	(1)
[3]	CURSOR_MSTATE_ROW	LLM_STATE_ROW	row position	(2)
[4]	CURSOR_MSTATE_COL	LLM_STATE_COL	column posit.	(2)
[5]	CURSOR_MSTATE_LEFT	LLM_STATE_LEFT	left button	(3)
[6]	CURSOR_MSTATE_RIGHT	LLM_STATE_RIGHT	right button	(3)
[7]	CURSOR_MSTATE_VISIBLE	LLM_STATE_VISIBLE	visible	(4)
[9]	CURSOR_MSTATE_SHAPE	LLM_STATE_SHAPE	shape	(5)
[10]	CURSOR_MSTATE_MID	n/a	mid button	(3)
[11]	BUTTON_MSTATE_LCOUNT	n/a	Left button press count	(6)
[12]	BUTTON_MSTATE_LROW	n/a	Left button last row	(7)
[13]	BUTTON_MSTATE_LCOL	n/a	Left button last col	(7)
[14]	BUTTON_MSTATE_RCOUNT	n/a	Right button press count	(6)
[15]	BUTTON_MSTATE_RROW	n/a	Right button last row	(7)
[16]	BUTTON_MSTATE_RCOL	n/a	Right button last col	(7)
[17]	BUTTON_MSTATE_MCOUNT	n/a	Mid button press count	(6)
[18]	BUTTON_MSTATE_MROW	n/a	Mid button last row	(7)
[19]	BUTTON_MSTATE_MCOL	n/a	Mid button last col	(7)

Note (1) coordinate in pixel
(2) coordinate in row/col
(3) CURSOR_MSTATE_BUTDOWN or LLM_BUTTON_DOWN = button down
CURSOR_MSTATE_BUTUP or LLM_BUTTON_UP = button up
(4) .T. = mouse cursor visible, .F. = hidden
(5) mouse cursor shape, see MsetCursor(), Mshow()
(6) count of button press 0..n since application start
(7) coordinate of last button press in pixel or 0

Description:

Mstate() returns information on the mouse state, i.e., the current screen position of the pointer, the state of the left and right mouse buttons, the visibility status of the mouse pointer ant the current shape.

Note: you may determine the mouse click also from

```
key := Inkey(n, INKEY_ALL_BUT_MOVE)
```

where the <key> returns more information e.g. double click, up or down press of the left, middle or right button etc.

Example:

```
aMstat := Mstate()
? "Current mouse pos: X/Ypix=" + ltrim(aMstat[CURSOR_MSTATE_X]) + ;
  "/" + ltrim(aMstat[CURSOR_MSTATE_Y]), ;
  "col=" + ltrim(aMstat[CURSOR_MSTATE_COL]), ;
  "row=" + ltrim(aMstat[CURSOR_MSTATE_ROW])
? "The mouse cursor " + if(aMstat[CURSOR_MSTATE_VISIBLE], ;
  "is", "is not") + " visible, and has shape " + ;
  ltrim(aMstat[CURSOR_MSTATE_SHAPE])
? "Mid mouse button is " + if(aMstat[CURSOR_MSTATE_MID] ;
  == CURSOR_MSTATE_BUTDOWN, "down", "up")
? "Left mouse button was " + ltrim(aMstat[BUTTON_MSTATE_LCOUNT]) +;
  "-times pressed"
if aMstat[BUTTON_MSTATE_LCOUNT] > 0
  ?? ", last press at row/col " + ;
     ltrim(Pixel2row(aMstat[BUTTON_MSTATE_LROW])) + "/" + ;
     ltrim(Pixel2col(aMstat[BUTTON_MSTATE_LCOL]))
endif
```

Compatibility:

New in FS5, available in CL53 except the above noted

Related:

Mpresent(), Mcol(), Mrow(), MsetCursor(), MLeftDown(), MrightDown(), MmidDown()

NETERR ()

Syntax:

retL = NETERR ([expL])

Purpose:

Checks the status of the USE, SET INDEX and APPEND BLANK commands in a multiuser environment for success or not. Optionally, sets/resets the value of NETERR().

Options:

<**expL**> is an optional new logical value set to the NETERR(). Setting the new NETERR() value allows the runtime error handler to control the way certain file errors are handled.

Returns:

<**retL**> is a logical value where TRUE signals an error when the command could not be executed. Possible causes of error:

Command	NETERR() = .T. when
APPEND BLANK	FLOCK() set by other user
USE...SHARED	USE..EXCL (or USE.. without SET EXCLUSIVE OFF) by other user
USE...EXCLUSIVE	USE..SHARED or ..EXCL executed by other user
SET INDEX TO...	Index is exclusive locked by other user

When the NETERR() value was set by <expL>, <retL> returns the previous set value.

Description:

After executing the commands above, the success should be checked to make sure the required action was executed. If NETERR() returns TRUE, the command should be tried again or the process terminated using BREAK or RETURN.

Tuning:

NetErr() always checks by USED() if the database in current work area is in use, and/or was successfully open. If not so, NetErr() returns failure (.T.). To avoid this check, set

```
_aGlobSetting[GSET_L_NETERR_USED] := .F.   // default is .T.
```

Multiuser:

In multiuser mode (SET EXCLUSIVE OFF or USE.. ..SHARED), the commands above do not generate a run-time error if they were not executed correctly. This allows the programmer to take appropriate action.

Example:

```
SET EXCLUSIVE OFF                          && = multiuser mode
USE address                                && see also the more
DO while NETERR()                          && comfortable check
   USE address                             && using "use_check"
ENDDO                                      && in USE example
SET INDEX TO name, id_no
IF NETERR()                                && check success of
   ? "Indices being used by other user" && index opening
   QUIT
ENDIF

APPEND BLANK                               && =automat. RLOCK()
DO WHILE NETERR()                          && if not successful
   APPEND BLANK                            && try again...
ENDDO                                      && until success
REPLACE name WITH "Smith"
UNLOCK                                     && free RLOCK()
```

Classification:

database

Compatibility:

The optional argument is new in FS and C5.

Related:

USE, SET EXCLUSIVE, SET INDEX, APPEND BLANK, FLOCK(), RLOCK()

NETNAME ()

Syntax:

`retC = NETNAME ()`

Purpose:

Retrieves the current workstation name and user login identification.

Returns:

<**retC**> is a character string containing the user identification (LOGNAME) and the current node name (same as $ uname -n). If the node name is unknown, a null string is returned.

On MS-Windows, NetName() returns the login name and the network or if such is not available, the local machine name in the syntax user@network or user@machine.

Example:

```
PUBLIC FlagShip
fullname = NETNAME()
if FlagShip
   username = SUBSTR (fullname, 1, AT("@",fullname) -1)
   ? fullname                              && JohnMiller@SUN1
   ? username                              && JohnMiller
else
   username = fullname
endif
if EMPTY(username)
   username = "unknown user"
endif
? "Good morning, " + username
```

Classification:

system access

Compatibility:

FlagShip returns the usual Unix login name such as "John@server". On MS-Windows, similar string is returned. With Clipper, this function returns the identification, but only on some networks.

Related:

Environment variable LOGNAME, UNIX: uname -n

NEXTKEY ()

Syntax:

`retN = NEXTKEY ([expN1], [expN2])`

Purpose:

Determines the next pending key or character in the keyboard buffer.

Options:

<**expN1**> specifies to return n-th occurrence (starting at 1) of a key fitting to the <expN2> filter mask. If not given, or <expN1> is less than 1, next key in buffer (or 0) is returned, i.e. NextKey(1) is equivalent to NextKey() and is the key returned by next Inkey(). Similarly, NextKey(2) is a key returned by over-next Inkey(), and so forth.

<**expN2**> is an input mask, filtering only specified events from the buffer. See INKEY_* constants in include/inkey.fh, default value is the value of Set (_SET_EVENTMASK), usually INKEY_ALL

Returns:

<**retN**> is a numeric value representing the ASCII value of the next key to be fetched. The value is equivalent to the INKEY() code, see <FlagShip_dir>/include/inkey.fh. If there are no key presses in the buffer or SET TYPEAHEAD is zero, zero is returned.

Description:

NEXTKEY() reads the next pending key without removing it from the keyboard buffer. NEXTKEY() can be used to check if any key has been pressed, before removing it by INKEY() or other keyboard commands, e.g. during the process of a loop or to check the next requested user action. This can be useful in cases where the user enters several keys in advance to avoid an additional overhead.

NEXTKEY() is similar to INKEY() function, but does not remove the pending key from the keyboard buffer and does not update the LASTKEY() value.

Example:

```
#include "inkey.fh"
USE mydbf
display_data()

WHILE !EOF()
   key := INKEY (0)                    // fetch the next input
   IF key == K_PGDN
      SKIP
      IF NEXTKEY() == K_PGDN           // do not display conti-
         LOOP                          // gnuous screen output
      ENDIF
   ENDIF
   display_data()                      // display the data
ENDDO
```

Classification:

programming, keyboard buffer access

Compatibility:

Both the optional parameters are new in FS5, same as mouse events, and not available in Clipper.

Related:

CLEAR TYPEAHEAD, SET TYPEAHEAD, KEYBOARD, INKEY(), LASTKEY()

NUM2HEX ()

Syntax:

`retC = Num2hex (expN1, [expN2], [expL3])`

Purpose:

Convert numeric (integer) value to a string representing it hexadecimal equivalence.

Arguments:

<**expN1**> is a numeric value to be converted to hex string The available decimals, if any, are truncated. Supported range is +/- 2147483647, i.e. approx. +/- 2 Giga/ billions.

<**expN2**> specifies the requested output length. If specified and the resulting string is shorter than <expN2>, it is filled left with "0" zeroes.

<**expL3**> if this modifier is .T., the hex string is prefaced by "Ox".

Description:

Num2hex() return a string representing the hexadecimal equivalence of the input value. See Hex2num() for reverse conversion.

Example:

```
? Num2Hex(1234)                    // "4D2"
? Num2Hex(1234.95)                 // "4D2"
? Num2Hex(1234, , .T.)             // "0x4D2"
? Num2Hex(1234, 6)                 // "0004D2"
? Num2Hex(1234, 6, .T.)            // "0x0004D2"
```

Compatibility:

New in FS5

Related:

Hex2num()

OBJCLONE ()

Syntax:

`retL = OBJCLONE (expO1, expO2)`

Purpose:

Duplicates an object, instead of make reference. Dangerous!

Arguments:

<**expO1**> is the source object to duplicate, it properties remain unchanged.

<**expO2**> is the destination object whose properties will be changed using those of <expO1>. Both <expO1> and <expO2> must be objects of the same class.

Returns:

<**retL**> is .T. on success, or .F. on failure. Additional messages are available in developer mode, i.e. when FS_SET("devel",.T.) is set.

Description:

OBJCLONE() is similar to ACLONE() for arrays. It duplicates the properties of <expO1> source object into the <expO2> destination. After executing ObjClone(), you may change any property in the <expO2> object without influencing the original <expO1> object, as opposite to an assignment, which creates a link (reference) only and hence the change in one object will influence both.

Since an object is often very complex and includes several internal data as well as local memory pointers, ObjClone() will copy only these data from <expO1> to <expO2>:

- all instances of type C,N,I,D,L but not of type A,O,S,U,special
- by ASSIGN property of the <expO2> object using the same named ACCESS of <expO1> or EXPORTed instance of <expO1> if such is available.

Other data in <expO2> object remain unchanged; methods are not executed.

WARNING, DANGER: The internal object/class data controls often the class behavior. Sometimes also the order of assignments is important. Copying the class internal properties from otherwise initialized object may influence the destination class or cause unexpected behavior. Be very careful to use OBJCLONE() on standard FlagShip classes, many of them holds internal pointers to e.g. GUI widgets - destroying the object copy may then cause segmentation violation. FlagShip internally does NOT use OBJCLONE() at all. You therefore should use OBJCLONE() only if explicitly required, and on well known (own) classes, checking the results thereafter. Better and always secure is to initialize the second object by the usual way, using instantiation and documented, public class properties.

Example:

```
LOCAL oFont1, oFont2, oFont3

oFont1 := Font{"Courier", 12}
oFont1:Bold := .T.          // font1 = Courier, 12, bold

oFont2 := Font{}            // instantiate new font object
? oFont1:Name, oFont2:Name  // = Courier Helvetica
? oFont1:Size, oFont2:Size  // = 12 10
? oFont1:Bold, oFont2:Bold  // = .T. .F.

ObjClone(oFont1, oFont2)    // copy instances of font1 to font2
oFont2:Name := "Arial"      // change property of font2
? oFont1:Name, oFont2:Name  // = Courier Arial
? oFont1:Size, oFont2:Size  // = 12 12
? oFont1:Bold, oFont2:Bold  // = .T. .T.

oFont3 := oFont1            // assignment (reference), not copy
oFont3:Name := "Times"      // changes property of font3 and font1
? oFont1:Name, oFont3:Name  // = Times Times
```

Classification:

programming

Compatibility:

New in FS5

Related:

ACLONE(), ISOBJCLASS(), ISOBJPROPERTY(), _DISPLOBJ()

OEM2ANSI ()

Syntax:

retC = Oem2ansi (<expC1>, [<expL2>], [<expC3>])

Syntax 2:

retC = OemToAnsi (<expC1>, [<expL2>], [<expC3>])

Purpose:

convert <expC1> string from PC8/ASCII/OEM character set to ISO/ANSI character set, including embedded \0 bytes

Arguments:

<**expC1**> is the character string to be converted.

<**expL2**> if .F. or not given, the translation should be "as close as possible" for PC8 characters not in ANSI charset. If .T., all unknown characters are replaced by <expC3>

<**expC3**> replacement for unknown/untranslatable chars, default is '?'

Returns:

<**retC**> is the resulting, translated string

Description:

The PC8/ASCII/OEM character set differs to ISO/ANSI for nearly all character values > 127. So is e.g. a-umlaut chr(132) in ASCII/PC8 set, but chr(228) in ISO/ANSI.

The used character set depends on the TERM setting (in Terminal i/o) or is per default ANSI/ISO in GUI mode. The PC8 char set is used in DOS and in Terminal i/o FS applications, the ANSI us used by X11 and Windows.

The inverse of Oem2ansi() is Ansi2oem(). Both functions are used in automatic database access/assign with SET ANSI ON.

Classification:

programming

Example:

```
see <FlagShip_dir>/examples/umlauts.prg for complete examples
```

Compatibility:

New in FS5

Related:

Ansi2oem(), SET ANSI

ONKEY ()

Syntax:

retB = OnKey ([expN1], [expB2])

Purpose:

Set/clear/return action performed on any key press

Options:

<**expN1**> is the Inkey() value of the key. If not specified or is 0, the action "on any key" is assumed.

<**expB2**> specifies a code block that will be automatically executed whenever the specified key is pressed during a wait status.

If <expN1> is numeric and <expB2> is not given, the currently assigned code block for <expN1> is returned.

If <expN1> is numeric and <expB2> is NIL, the corresponding key action of <expN1> is cleared, same as ON KEY <expN1>.

If both <expN1> and <expB2> are given and NIL, all ON KEY actions are cleared, same as ON KEY CLEAR

Returns:

<**retB**> is the action block currently associated with the specified key, or NIL if the specified key is not currently associated with a block.

Description:

This function is equivalent to ON KEY nKey [EVAL cBlock]. OnKey() with <expN1> is equivalent to SetKey() or to SET KEY command. It sets or retrieves the automatic action associated with a particular key during a wait status. A wait status is any mode that extracts keys from the keyboard except for INKEY(), but including ACHOICE(), INKEYTRAP(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT. Any number of keys may be assigned at any one time.

SETKEY() or ONKEY() is usually used to assign a UDP or UDF to a specific key.

ONKEY(0, [expB2]) is equivalent to ON ANY KEY ... and evaluates the assigned code block on any key which is not explicitly assigned by itself. Upon starting up, the F1 key is automatically assigned to a UDF or UDP named HELP, if such exist.

When an assigned key is pressed during a wait state, the EVAL() function evaluates the associated action block, passing the parameters PROCNAME(), PROCLINE(), and READVAR().

For more information, refer to the ON KEY or SET KEY command.

Classification: programming
Compatibility: new in FS5
Related: ON KEY, SET FUNCTION, SET KEY, SetKey(), Eval(), Inkey(), InkeyTrap(), GetKey(), IsFunction()

ORD*()

The Clipper's ORD*() functions are supported for backward compatibility purposes only, since FlagShip's oRdd:Ord*() methods provides more information and functionality. The <oRdd> object is available via DbObject() function.

OrdBagExt()

Return the default order bag RDD extension equivalent to IndexExt() or oRdd:OrderInfo(DBOI_EXPRESSION)

OrdBagName()

Return the order bag name of a specific order extracts the information from IndexNames()

OrdCond()

Specify conditions for ordering unsupported because of the weird syntax

OrdCondSet()

Set the condition and scope for an order supported, you also may use INDEX ON... TO...

OrdCreate()

Create an order in an order bag supported, you also may use INDEX ON... TO...

OrdDestroy()

Remove a specified order from an order bag supported, you also may use SET INDEX ...

OrdDescend()

Return and optionally change the descending flag of an order, equivalent to oRdd:OrderInfo(DBOI_ISDESC, ...)

OrdFor()

Return the FOR expression of an order, equivalent to oRdd:OrderInfo (DBOI_CONDITION, ...)

OrdIsunique()

Return the status of the unique flag for a given order equivalent to oRdd:OrderInfo (DBOI_UNIQUE, ...)

OrdKey()

Return the key expression of an order, equivalent to oRdd:OrderInfo (DBOI_EXPRESSION, ...)

OrdKeyAdd()

Add a key to a custom built order the DBFIDX driver always return .F.

OrdKeyCount()

Return the number of keys in an order, equivalent to oRdd:OrderInfo (DBOI_KEYCOUNT, ...)

OrdKeyDel()

Delete a key from a custom built order the DBFIDX driver always return .F.

OrdKeyGoto()

Move to a record specified by its logical record number equivalent to oRdd:OrderKeyGoto()

OrdKeyNo()

Get the logical record number of the current record, equivalent to oRdd:OrderInfo (DBOI_POSITION, ...)

OrdKeyVal()

Get key value of the current record from controlling order equivalent to &(IndexKey(0))

OrdListAdd()

Add orders to the order list supported, you also may use SET INDEX ...

OrdListClear()

Clear the current order list supported, you also may use SET INDEX ...

OrdListRebui()

Rebuild all orders in the order list of the current work area supported, you also may use REINDEX

OrdName()

Return the name of an order in the order list equivalent to OrdBagName()

OrdNumber()

Return the position of an order in the current order list extracts the information from IndexNames()

OrdScope()

Set or clear the boundaries for scoping key values equivalent to oRdd:OrderInfo(...)

OrdSetFocu()

Set focus to an order in an order list supported, you also may use SET ORDER ...

OrdSetRelat()

Relate a specified work equivalent to DbSetRelation()

OrdSkipUnique()

Move record pointer to the next or previous unique key skips directly +/- records on UNIQUE index, or simulates unique by comparing the index key

OS ()

Syntax:

`retC = OS ()`

Purpose:

Returns the operating system name and its release.

Returns:

<**retC**> is a character string containing the name of the current used Unix system and the OS release. On MS-Windows, OS() returns the Windows name and release, e.g. "Windows (Build 9200) Release 6.2" or "Windows Service Pack 1 (Build 7601) Release 6.1" and so on.

Description:

OS() returns the name of the system currently used. The return value <retC> consists of two parts:

- the FlagShip OS system used (see FSC section) and
- the current UNIX or Windows release, preceded by the verb " Release ", see example below.

Example:

```
myos = OS()
release = SUBSTR(myos, AT("Release ", myos) + 9)
fssyst  = SUBSTR(myos, 1, AT(" Release ",myos))

? myos                // SCO UNIX V/386 3.2 Release 3.4 5.6
? fssyst              // SCO UNIX V/386 3.2
? release             // 3.4 5.6
```

Classification:

system call

Compatibility:

The return value differs from Clipper, which return e.g. "DOS 5.0". The release is OS dependent and need not contain digits only.

Related:

NETNAME(), VERSION(), UNIX: man uname

OUTERR ()

Syntax:

`NIL = OUTERR (expList)`

Purpose:

Writes a list of values to the standard error device.

Arguments:

<**expList**> is a list of values or expression of any type to display.

Returns:

The function always returns NIL.

Description:

OUTERR() is identical to OUTSTD() and similar to the ? command or QOUT() function, except that it writes to the stderr device rather than the stdout device. It uses the low-level fprintf(stderr,...) function. The output may be rerouted to any other device or file by >, >> or | (see more in the ?# command), e.g.

```
$ ./a.out par1 par2 2>>err.log   // Unix, Linux
$ my.exe  par1 par2 2>>err.log   // MS-Windows
```

Program internal rerouting, using SET PRINTER or SET ALTERNATE commands is not supported by OUTERR().

The OUTERR() is similar to the C output printf() or the RUN ("echo" + " " + exp) command. Since OUTERR() by-passes the curses, the subsequent screen output may produce unpredictable results, if no Unix redirection is used. Issue the REFRESH command to restore the standard screen output in a such case.

Example:

```
? "Cannot open database..."
OUTERR ("Database open failure:", ;
        PROCNAME(), PROCLINE(), NETNAME(), DATE(), TIME())
REFRESH                                  // to be sure
```

Classification:

system call, output

Compatibility:

Available in FS and C5 only. Note the differences in rerouting the output using the DOS and UNIX redirections. Embedded zero bytes are not supported.

Related:

OUTSTD(), ?, ??, ?#, ??##, QOUT(), QQOUT(), DISPOUT(), REFRESH

OUTSTD ()

Syntax:

`NIL = OUTSTD (expList)`

Purpose:

Writes a list of values to the standard output device, exactly same as fprintf(stdout,...) in C

Arguments:

<**expList**> is a list of values or expression of any type to display.

Returns:

The function always returns NIL.

Description:

OUTSTD() is similar to the ? command or QOUT() function, except that it ignores the SET PRINTER or SET ALTERNATE setting. Using the header file "simpleio.fh" will redefine some console output commands to OUTSTD().

The direct rerouting of the standard output to any other device or file using the Unix/Windows commands >, >> or | is possible, e.g.

```
$ ./a.out par1 par2 > myfile.txt   // Unix, Linux
$ my.exe  par1 par2 > myfile.txt   // MS-Windows
```

but is not recommended, since the terminal control escape sequences will also be written to the redirected file. OUTSTD() uses the stdout stream, as the Unix curses do. Therefore all the console and screen output including @..SAY, @..GET, ?, QOUT(), WAIT etc. is rerouted. To redirect only a specific message to another device or file, use the OUTERR() function (or the usual ?, ?? commands with SET ALTERNATE, SET EXTRA or SET PRINTER) instead. In special cases, if the screen oriented output is not required at all, you may disable the Curses according to section SYS.2.7.

Example:

```
SET PRINTER ON
? "output appears to screen and printer"
OUTSTD ("screen output only")
SET PRINTER OFF
```

Classification: sequential output

Compatibility:

Available in FS and C5 only. The rerouting on the OS command line differs in DOS/Windows from that in UNIX. Embedded zero bytes are supported in FS by default.

Include: <FlagShip_dir>/include/simpleio.fh

Related:

OUTERR(), ?, ??, QOUT(), QQOUT(), DISPOUT()

PADC ()
PADL ()
PADR ()

Syntax:

```
retC = PADC (exp1, expN2, [expC3])
retC = PADL (exp1, expN2, [expC3])
retC = PADR (exp1, expN2, [expC3])
```

Purpose:

Pads character, date, and numeric values with a fill character.

Arguments:

<**exp1**> is a character, numeric, or date value to pad with a fill character.

<**expN2**> is the length of the character string to return.

Options:

<**expC3**> is the character to pad <exp1> with. If not specified, the default is a space character.

Returns:

<**retC**> is the padded string to a total length of <expN2>.

Description:

PADC(), PADL(), and PADR() are character functions that pad character, date, and numeric values with a fill character to create a new character string of a specified, fixed length. They can be used for alignment in ?? commands, complex index keys etc.

PADC() centers the expression <exp1> within <expN2> adding fill characters to the left and right sides,

PADL() adds fill characters on the left side,

PADR() adds fill characters on the right side, up to the <expN2> total length.

If the length of <exp1> exceeds <expN2>, the resulting <retC> string is truncated to <expN2> characters.

The inverse functions are ALLTRIM(), LTRIM() and RTRIM() which removes leading and trailing spaces from the string.

Example:

```
USE address
INDEX ON PADR( UPPER( TRIM(name) + ", " + first), 25)
DO WHILE ! EOF()
   ?  PADR (TRIM(name) + ", " + first), 40)
   ?? PADL (country, 5), city
   ?? PADC (turnover, 20, "*")
   SKIP
ENDDO
```

Classification:

programming

Compatibility:

Available in FS and C5 only. Embedded zero bytes are supported in FS by default.

Related:

ALLTRIM(), LTRIM(), RTRIM(), STR(), SUBSTR(), FS2:PadLeft(), FS2:PadRight()

PARAM ()

Syntax:

`ret = Param (expN1)`

Purpose:

Returns the by number specified parameter

Arguments:

<**expN1**> is the ordinal number of the function parameter. Valid values are 1 to pcount()

Returns:

<**ret**> is the received parameter value or NIL if invalid

Description:

Usually, you will receive parameters passed to an UDF (Procedure or Function) by name. In special cases, it is more convenient to use loops and receive the parameter value by its ordinal number.

Example:

```
x := myUdf (1, "hello", NIL, .F.)
:
FUNCTION myUdf(nParam, cString, anyVal, myLogic)
? nParam                  // 1
? cString                 // "hello"
? anyVal                  // NIL
? myLogic                 // .F.
for pos := 1 to Pcount()
   ? Param(pos)           // 1, "hello", NIL, .F.
next
return Pcount()
```

Classification:

programming

Compatibility:

New in FS5

Related:

PARAMETERS, local parameters of FUNCTION or PROCEDURE

PARAMETERS ()

Syntax:

```
retN = Parameters ()
```

Purpose:

Equivalent to PCOUNT()

Returns:

<**retN**> is a numeric value, representing the number of parameters passed. If no parameters are passed, zero is returned.

Description:

For FoxPro compatibility purposes, available when compiled with the -foxpro switch. Equivalent to the standard function Pcount()

Classification:

programming

Compatibility:

New in FS5

Related:

Pcount(), Param(), PARAMETERS

PCALLS ()

Syntax:

`retN = PCALLS ()`

Purpose:

Outputs the current callstack.

Returns:

<**retN**> is the calling depth of the current UDP or UDF, starting with 1 in the main program.

Description:

PCALLS() is mainly used for debugging purposes. It outputs the current callstack trace including the source file, UDF or UDP name and the line number.

For more information, refer to PROCFILE(), PROCNAME() and PROCLINE().

Example:

```
*** File testmain.prg
SET PROCEDURE TO testproc
// call another file and UDP
DO mytest
*** eof

*** File testproc.prg
PROCEDURE mytest
x = PCALLS()                // TESTPROC: Fn/Line MYTEST/3
                            // TESTMAIN: Fn/Line TESTMAIN/4
? x                         // 2
*** eof
```

Classification:

programming

Compatibility:

Available in FS only.

Related:

PROCNAME(), PROCLINE(), PROCFILE(), PROCSTACK()

PCOL ()

Syntax:

`retN = PCOL ()`

Purpose:

Reports the current printer column position.

Returns:

<**retN**> is a numeric value representing the current printer head column. This value, beginning with zero, indicates where the next character will be printed. If the last character was printed at column 36, for example, PCOL() returns 37.

Description:

PCOL() can be used to print a line of text relative to the previous column. PCOL() also counts the control codes. However, these are not actually printed, so that after sending such characters to the printer, PCOL() will not return the correct column of the printer head, but the size of the sent characters at this line. For adjusting in such cases, SETPRC() should be used.

PCOL() is updated only if either SET DEVICE TO PRINTER or SET PRINTER ON is in effect. PCOL() is the same as COL() except that it relates to the printer rather than the screen.

EJECT, besides advancing the paper, resets PCOL() and PROW() to zero. SETPRC() sets PROW() and PCOL() to any value without causing the paper to move.

When printing with @...SAY, the current SET MARGIN value is always added to the specified column position before output is sent to the printer.

Example:

```
SET DEVICE TO PRINT
EJECT
@ PROW(), 10 SAY "Conti"
@ PROW(), PCOL() SAY "nuing output"
SET DEVICE TO SCREEN
```

Classification:

programming

Compatibility:

Note the standard spooled output used by FlagShip, see LNG.5.1.6, LNG.3.6 and SET PRINTER command. Regardless of printer redirection, the PCOL() is set the same way as for a direct printer output.

Related:

PROW(), COL(), SETPRC(), EJECT, SET DEVICE, SET PRINT

PCOUNT ()

Syntax:

`retN = PCOUNT ()`

Purpose:

Retrieves the number of parameters which have been passed to a program, procedure, user defined function or a class method.

Returns:

<**retN**> is a numeric value, representing the number of parameters passed. If no parameters are passed, zero is returned.

Description:

PCOUNT() reports the position of the last argument passed to the UDP or UDF using a procedure or function call, or to the main program passed from the Unix or Windows command line.

This information is useful when determining whether arguments were left off the end of the argument list. Arguments skipped in the middle of he list are still included in the value returned. Such skipped parameters can be determined using a NIL comparison or checking them by TYPE() or VALTYPE().

Example 1:

```
*** Program "test.prg"
PARAMETERS input1, input2              // passed from command-line
param = PCOUNT()
input1 = IF (param < 1, "", input1)
input2 = IF (param < 2, "", input2)

DO myproc WITH input1, input2, 20
DO myproc WITH input1, , "test"

PROCEDURE myproc (par1, par2, par3)
// parameters 2 and 3 are optional
IF PCOUNT() < 1
   ? "error, no param 1 given"
   RETURN
ENDIF
// set defaults
par1 := IF (VALTYPE(par1) != "C", "default", par1)
par2 := IF (par2 == NIL,  0,  par2)
par3 := IF (par3 == NIL, .T., par3)
? par1, par2, par3                    // process the UDP...
RETURN
```

Example 2:

```
*** Program "test.prg"
PARAMETERS dummy
local cParam := ""
if !empty(dummy)
   // concatenate all input parameters, separate them by space
   local ii
   for ii := 1 to PCOUNT()
      cParam += PARAM(ii) + " "
   next
   cParam := alltrim(cParam)
   ? "Parameters received from commandline:", cParam
endif
```

Classification:

programming

Related:

PARAMETERS, PROCEDURE, FUNCTION, METHOD, PARAM()

PIXEL2COL ()

Syntax:

```
retN = Pixel2col (expN1)
```

Purpose:

calculates the column in col/row coordinates from the given pixel value corresponding to the current i/o Font.

Arguments:

<**expN1**> is a numeric constant or variable specifying the x-pixel position which should be recalculated to column.

Returns:

<**retN**> is numeric value (fraction of columns) corresponding to the given pixel value and the selected font.

Description:

The function is equivalent to oApplic:Pixel2col(expN1)

Classification:

programming

Compatibility:

New in FS5

Related:

Pixel2row(), Col2pixel(), Row2pixel(), oApplic:Pixel2col()

PIXEL2ROW ()

Syntax:

```
retN = Pixel2row (expN1)
```

Purpose:

calculates the row in col/row coordinates from the given pixel value corresponding to the current i/o Font.

Arguments:

<**expN1**> is a numeric constant or variable specifying the y-pixel position which should be recalculated to row.

Returns:

<**retN**> is numeric value (fraction of rows) corresponding to the given pixel value and the selected font.

Description:

The function is equivalent to oApplic:Pixel2row(expN1)

Classification:

programming

Compatibility:

New in FS5

Related:

Pixel2col(), Col2pixel(), Row2pixel(), oApplic:Pixel2row()

PRINTGUI ()

Syntax:

```
retL = PrintGui ( [exp1] )
```

Purpose:

Starts, continues, terminates or invokes GUI/GDI printer output. Applicable for GUI mode only, ignored otherwise (with developer warning).

Arguments:

<exp1> = .T. starts or continues GUI printer output, same as invoking SET GUIPRINT ON or SET PRINTER GUI ON **<exp1> = .F.** stops or temporary terminates GUI printer output, same as invoking SET GUIPRINT OFF or SET PRINTER GUI OFF **<exp1> = NIL** or not specified: stops GUI printer output, and prints the output to selected driver via _oPrinter:GUIexec()

Returns:

<**retL**> is a logical value signaling success or failure.

Description:

PrintGui() is used to control GUI/GDI printer (*) output to selected system driver. It is a shortcut for start/stop rendering the output by SET GUIPRINT, and to process printout by oPrinter:GUIexec().

(*)what is GUI/GDI printer? Extract from http://en.wikipedia.org/wiki/Graphics_Device_Interface#GDI_printers:

> *A GDI printer or Winprinter (analogous to a Winmodem) is a printer designed to accept output from a host computer running the GDI under Windows (in FlagShip: also in Linux). The host computer does all print processing: the GDI software renders a page as a bitmap which is sent to a software printer driver, usually supplied by the printer manufacturer, for processing for the particular printer, and then to the printer. The combination of the GDI and driver is bidirectional; they receive information from the printer such as whether it is ready to print, if it is out of paper or ink, and so on.*

You will create GUI printer output by the same way as a screen output by common ?, ??, qout(), @..SAY, @..DRAW etc. commands and functions. When SET CONSOLE is ON and/or SET DEVICE is SCREEN, the output goes (parallel) to screen. When SET GUIPRINT is ON or PrintGui(.T.) was invoked, the output is (parallel) rendered for printer output.

The first invocation of **PrintGui(.T.)** or the same SET GUIPRINT ON calls _oPrinter:Setup() dialog to select installed printer driver, if the _oPrinter:Setup() was not yet invoked manually. This setting remains active also after printing by PrintGui() but can also be changed by _oPrinter:Setup(), it however cleans the current print buffer, if any. It then starts rendering of printer output.

Invoking **PrintGui(.F.)** or the same SET GUIPRINT OFF stops rendering of printer output. The next invocation of PrintGui(.T.) or the same SET GUIPRINT ON continues output redirection to rendering buffer.

PrintGui() without parameter or the same _oPrinter:GUIexec() stops rendering, and prints the buffer by using already selected printer driver, then clears the buffer. Next PrintGui(.T.) will start new rendering for printer.

You can control printout of large text to several pages either manually by EJECT (using values from oPrinter:GuiMaxRow() and oPrinter: GuiRow()), or let issue FormFeed automatically by SET EJECT ON.

Following scenarios with PrintGui() are supported:

Screen only	[PrintGui(.F.) ;] [SET CONSOLE ON ; SET DEVICE TO SCREEN] [SET PRINTER OFF] ?, ?? and @... output
Screen + GUI Printer	PrintGui(.T.) [or SET PRINTER GUI ON] ; [SET CONSOLE ON ; SET DEVICE TO SCREEN] [SET PRINTER OFF] [SET EJECT ON] ?, ?? and @... output PrintGui()
GUI printer only	PrintGui(.T.) [or SET PRINTER GUI ON] ; SET CONSOLE OFF ; SET DEVICE TO PRINT [SET PRINTER OFF] [SET EJECT ON] ?, ?? and @... output PrintGui() ; SET CONSOLE ON ; SET DEVICE TO SCREEN
GUI Printer + ASCII file output	PrintGui(.T.) [or SET PRINTER GUI ON] ; SET CONSOLE OFF ; SET DEVICE TO PRINT ; [SET PRINTER TO (fileName) ;] SET PRINTER ON [SET EJECT ON] ?, ?? and @... output PrintGui() ; SET CONSOLE ON ; SET DEVICE TO SCREEN ; SET PRINTER OFF ; oPrinter:Exec()
GUI Printer + other ASCII Printer	PrintGui(.T.) [or SET PRINTER GUI ON] ; SET CONSOLE OFF ; SET DEVICE TO PRINT ; SET PRINTER TO LPT3 [or TO /dev/lp2] ; SET PRINTER ON [SET EJECT ON] ?, ?? and @... output PrintGui() ; SET CONSOLE ON ; SET DEVICE TO SCREEN ; SET PRINTER OFF

Supported commands and functions:

Following commands and functions are (parallely) processed for printing (see details in corresponding manual page) when PrintGui(.T.) or SET GUIPRINT ON is active:

?.., ??..	std. text output (opt. with color and font)
@.. BOX ..	draw box (with SET GUITRANSL BOX ON)
@.. DRAW ..	draw line, circle, arc, pie, polygon etc.
@.. TO ..	draw lines (with SET GUITRANSL LINES ON)
@.. SAY ..	coordinates oriented text output
@.. SAY IMAGE ..	draw bitmap images
.. PRINTCOLOR ..	command clause for printout color, or
.. GUICOLOR ..	printout color if PRINTCOLOR clause is n/a
.. FONT ..	command clause for screen & printout font
EJECT	execute FormFeed (new page)
SET EJECT OFF/ON	disable/enable automatic FormFeed
SET FONT ..	assign default font, same as _oPrinter:Font
SET PRINTER OFF/ON	additional redirection to printer or file
SET GUIPRINT OFF/ON	same as PrintGui(.F.) and PrintGui(.T.)
Devpos()	set (screen and) printer cursor position
Devout()	std. text output
PrintGui(.F./.T.)	stop/continue printer buffering
PrintGui()	stop buffering, print = oPrinter:GUIexec()
Qout(), Qqout()	std. text output
Setpos()	set cursor position

You additionally may set, check or execute at any time:

SET GUITRANSL ..	(dis)allow semi-graphic, lines, boxes
SET FONT BASELINE ..	enable/disable font alignment to base line
SET SOURCE ..	translate from/to ASCII/OEM or ISO/ANSI
SET ...	most of standard settings
Set(...)	corresponds to SET command
oPrinter:Driver*()	driver data
oPrinter:Font	printer default font
oPrinter:Margn*()	margins
oPrinter:Page*()	page settings
oPrinter:Setup()	dialog to select printer driver
oPrinter:SetupAborted	was printer select dialog aborted?
oPrinter:GUIabort()	abort rendering, clear buffer w/o printing
oPrinter:GUIcol()	get/set printer column
oPrinter:GUIdraw*()	draw lines, box, arc, circle, pie etc.
oPrinter:GUIexec()	same as PrintGui() = print buffer
oPrinter:GUIfixPage()	adjust page size
oPrinter:GUImaxCol()	max printer column
oPrinter:GUImaxRow()	max printer row
oPrinter:GUInewLine()	same as ? command, linefeed = new line
oPrinter:GUInewPage()	same as EJECT command, formfeed = new page
oPrinter:GUIpageNum	get/set current page (1..n or 0 on failure)

oPrinter:GUIrow()	get/set printer row
oPrinter:GUIsetColor()	set default printer color, see below
oPrinter:GUIsetFont()	set default printer font
oPrinter:GUIsetPos()	set printer's cursor position
oPrinter:GUItestPage()	print test page
oPrinter:GUItextOut()	same as ?? command, print text

The used color for GUI printouts is determined in this sequence:

a) PRINTCOLOR clause of @..SAY, @..DRAW, ?, ?? etc. commands
b) oPrinter:GUIsetColor() setting, if not empty
c) GUICOLOR clause of @..SAY, @..DRAW, ?, ?? etc. commands
d) SetColor() when SET GUICOLOR is ON
e) otherwise "N/W+"

Tuning:

FlagShip uses standard printer drivers, assigned by CUPS in Linux or by "Printers and Faxes" in MS-Windows. Some of these drivers does not return exact measurements, or the top/bottom/left/right margins differs for first and next pages. If you need exact page output, you may tune/override the driver settings by your own data (determined e.g. by the oPrinter:GUItestPage() method, see example below), and assigned to oPrinter:GuiFixPage() method. These page data are passed in an array of 5 or 6 elements:

```
aPrintData [1] = printer driver name (char, optional)
           [2] = paper size (char, same as oPrinter:PageSize)
           [3] = orientation (char, 1st char = "P" or "L")
           [4] = units of element data [5,6] (numeric = UNIT_ROWCOL,
                 UNIT_MM, UNIT_CM, UNIT_INCH, UNIT_PIXEL, UNIT_DOTS)
           [5] = sub-array of 4 numeric elements with paper size and
                 margins for the 1st (and next pages) in above
units:
                 [1] = non-printable left margin
                 [2] = printable width
                 [3] = non-printable top margin
                 [4] = printable height
           [6] = sub-array of 4 numeric elements with paper size and
                 margins for 2nd and next pages in above units, with
                 the same structure as element [5]. When element [6]
                 is omitted, data of element [5] are taken for the
                 1st and all following pages.
```

For example, set page data of LAN printer for Windows and/or Linux:

```
#ifdef FS_WIN32    /* compiled by VFS for MS-Windows */
  aPrintSetup :=  {"Canon MX850 series Printer", "a4", "portr", ;
                   UNIT_MM, {3.4, 203.1, 5, 286.8}, ;
                            {0, 203.1, 0, 286.8} }
#else              /* compiled by VFS for Linux */
  aPrintSetup := {"i865", "A4", "Portrait", ;
                  UNIT_MM, {13, 188, 8, 271} }
#endif
// oPrinter:Setup()   // optional
```

```
oPrinter:GuiFixPage(aPrintSetup)
PrintGui(.T.)          // start buffering for printer output
// ?, ??, @..SAY, @..DRAW etc.
PrintGui()             // end buffering, print it
```

Another alternative is to specify array of all used or all expected printers assigning it to the `_aGlobSetting[GSET_A_GUIPRINT_SIZE]` global variable. The first PrintGui() invocation (or explicit call to oPrinter:GUIstart() method) checks if oPrinter:GuiFix-Page() was assigned and if not so, compares all printer driver names (in any) in the `_aGlobSetting[GSET_A_GUIPRINT_SIZE]` with currently selected printer (case insensitive). The structure of each array element is equivalent to above aPrintData{...}, for example:

```
#ifdef FS_LINUX    /* compiled by VFS for Linux */
 _aGlobSetting[GSET_A_GUIPRINT_SIZE] := ;
    {{"printer", "A4", "Portrait", UNIT_MM, {13, 188, 8, 271}}, ;
     {"i865", "A4", "Portrait", UNIT_MM, {13, 188, 8, 271}} }
#else              /* compiled by VFS for MS-Windows */
 _aGlobSetting[GSET_A_GUIPRINT_SIZE] := ;
    {{"HP LaserJet 2100 PCL6", "A4", "Portrait", ;
          UNIT_MM, {5.5, 198, 6.4, 283.5}, {0, 198, 0, 283.5}}, ;
      {"Canon MX850 series Printer", "a4", "portr", ;
          UNIT_MM, {3.4, 203.1, 5, 286.8}, {0, 203.1, 0, 286.8}} }
#endif
```

Example 1:

```
// SET CONSOLE OFF           // if screen output is not required
PrintGui(.T.)                // select printer, start printout
for ii := 1 to 10
   ? "Print line",ltrim(ii)  // output to printer (and to screen)
next
PrintGui()                   // end printout, send it to printer
// SET CONSOLE ON            // enable screen output
wait
```

Example 2:

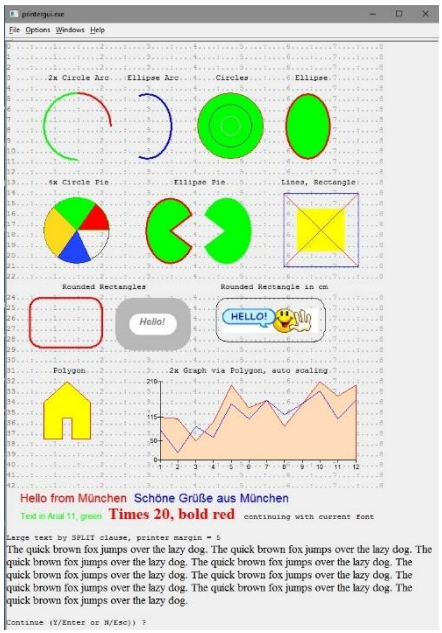
```
set source ANSI              // convert national characters
set font "courier", 10       // set default font = courier
local oFont := Font{}        // new font object
oApplic:Font:CloneTo(oFont)  // save current applic. font = courier
oPrinter:Setup()             // select printer in dialog
if oPrinter:SetupAborted     // check for user abort
   wait "sorry, printout aborted ..."
   quit
endif

PrintGui(.T.)                // start buffering for GUI printer
// SET CONSOLE OFF           // if console output is not required
// SET DEVICE TO             // if screen output is not required
oPrinter:GUItestPage()       // optional: print test page
EJECT                        // optional: formfeed after test page
@ 5,10 SAY "Text at row/col 5,10"
SET FONT "Arial",11          // set new default font
? "Hello from München"       // umlauts are here in ANSI/ISO
SET FONT BASELINE ON         // align font on base line
@ 8,3 SAY "Text in Arial 11, green" ;
```

```
            GUICOLOR "G+" PRINTCOLOR "G+"
?? " and in Times 20, bold red" ;
            FONT FontNew("Times",20,"B") ;
            GUICOLOR "R+" PRINTCOLOR "#FF0000"
?? " continuing with current font"
SET FONT BASELINE OFF        // align font on top (default)
@ 7,5,12,9 SAY IMAGE ;
      FILE "mypicture.jpg" SCALE UNIT=CM
PrintGui()                   // print to selected printer

// SET CONSOLE ON            // enable console output
// SET DEVICE TO SCREEN      // enable screen output
oFont:CloneTo(oApplic:Font)  // restore application font
wait "done ..."              // now using courier 10
```

Example 3:

see complete example in <FlagShip_dir>/examples/printergui.prg

Output:



Classification:

printer output in GUI mode

Compatibility:

New in VFS7, n/a in Clipper nor in FoxPro.

Related:

SET GUIPRINTER, SET PRINTER, SET EJECT, SET CONSOLE, SET DEVICE, set(_SET_GUIPRINTER), OBJ:Printer class

PRINTSTATUS ()

Syntax:

```
retL = PrintStatus ()
```

Purpose:

Equivalent to ISPRINTER()

Ensures compatibility with other dialects. FlagShip stores all printer output to the printer file, so, logically, the printer is always "ON LINE".

Returns:

<**retL**> is always TRUE.

Classification:

programming

Compatibility:

New in FS5

Related:

IsPrinter()

PROCFILE ()

Syntax:

`retC = PROCFILE ([expN1], [expL2])`

Purpose:

Retrieves the name of the current or previous source file being executed.

Options:

<**expN1**> is a numeric value representing the depth in the calling sequence of the required file name. Zero (the default) returns the name of the source file in which the currently executed procedure, function or program is contained; 1 is the source file of the sequence which called the present one, and so on.

<**expL2**> is an optional logical value. If not specified or is .F., up to 128 significant chars of the source name (w/o extension) are returned. The source name is stored "as is" during the compilation but without the file extension (.prg), and without translating it to lower or upper case. If <expL2> is .T., only the first 10 characters of the file name, translated to upper case, are returned. This is compatible to FlagShip 4.4x releases.

Returns:

<**retC**> is a character string, representing the name of the source file (without extension) where the current (or required) procedure, function or code block is active. When <expN> is greater than the calling stack depth, a null string "" is returned.

Description:

PROCFILE() is used for debugging purposes or for implementing context sensitive help.

Example:

see also the example in PROCLINE(), PROCNAME().

```
** file "test.prg"
? PROCFILE(), PROCNAME(), PROCLINE()      // test TEST 2
DO myudf1

PROCEDURE myudf
? PROCFILE(,.T.), PROCNAME(), PROCLINE()  // TEST MYUDF 6
RETURN
```

Classification:

programming

Compatibility:

Available in FlagShip only. The 2nd parameter is available in FS5 only.

Related:

PROCNAME(), PROCLINE(), PCALLS(), PROCSTACK()

PROCLINE ()

Syntax:

`retN = PROCLINE ([expN])`

Purpose:

Retrieves the source line number of the current or a previous program, UDP or UDF.

Options:

<**expN**> is a numeric value representing the depth in the calling sequence. Zero (the default) means the current program, procedure or function, 1 the procedures and functions which called this one and so on.

Returns:

<**retN**> is a numeric value, representing the current source code line number within the .prg file. If the program was compiled with the "-nl" option, PROCLINE() always returns 0. When <expN> is greater than the caller calling stack depth, zero is returned.

Description:

PROCLINE() is used for debugging purposes or for implementing context sensitive help.

A line number is relative to the beginning of the original source file, including all comments, directives and empty lines. A multi-statement line is counted as a single line.

On a statement which continues throughout several lines, the line number with which the statement starts is returned. If the action being queried is a code block evaluation, PROCLINE() returns the line number of the procedure in which the code block was originally defined.

Example:

```
see also the example in PROCNAME(), PROCFILE().

IF NETERR()
   ? "Error in procedure ", PROCNAME()
   ?? ", at the line ", PROCLINE()
ENDIF
```

Classification:

programming

Related:

PROCNAME(), PROCFILE(), PCALLS(), PROCSTACK()

PROCNAME ()

Syntax:

retC = PROCNAME ([expN])

Purpose:

Retrieves the name of the current or previous program, UDP or UDF being executed.

Options:

<**expN**> is a numeric value representing the depth in the calling sequence. Zero (the default) means the current UDP/UDF name, 1 the procedure and function which called the present one and so on.

Returns:

<**retC**> is a character string in uppercase, representing the name of the current (or required) procedure, function or code block. When <expN> is greater than the caller calling stack depth, a null string "" is returned.

Description:

PROCLINE() is used for debugging purposes or for implementing context sensitive help.

For the default action, PROCLINE() reports the name of the current procedure or UDF. If <expN> is one (or two etc.), the name of the UDF or UDP that invoked the current (or previous) procedure is returned.

If the activation being queried is a code block evaluation, PROCNAME() returns the internal name of the code block, like "CB_1_234". The first number in the name (here 1) is the ordinal number of the code block in the line 234. The second number (here 234) specifies the line number in which the block is declared.

Example:

```
 1: *** Program "test.prg"
 2: LOCAL bl := {|x| abc(x) }
 3: ? PROCNAME(), PROCLINE()  && TEST  3
 4: ? show_calls()            && TEST/4
 5: DO proc1                  && TEST/5-->PROC1/10
 6: EVAL (bl, "test")         && TEST/6-->CB_1_2/2-->ABC/15
 7: QUIT
 8:
 9: PROCEDURE proc1
10: ? show_calls()            && TEST/5-->PROC1/10
11: abc ()                    && TEST/5-->PROC1/11-->ABC/15
12: RETURN
13: *
14: FUNCTION abc (par)
15: ? show_calls()
16: RETURN .T.

// #define CLIPPER_COMPATIBLE_OUTPUT
FUNCTION show_calls           && display callstack trace
LOCAL ii, output := "", temp
```

```
FOR ii = 1 to 20              && up to depth 20
   temp = PROCNAME (ii)
   IF EMPTY(temp)
      EXIT
   ENDIF
#ifdef CLIPPER_COMPATIBLE_OUTPUT
   IF SUBSTR(temp,1,3) == "CB_"
      temp = "(b)" + PROCNAME(ii+1)
   ENDIF
#endif
   temp = temp + if (PROCLINE(ii) > 0, ;
          "/" + LTRIM(STR(PROCLINE(ii))), "")
   output = temp + IF (!EMPTY(output),"-->","") + output
NEXT
RETURN output
```

Classification:

programming

Compatibility:

The optional argument is not available in C87. The returned value of a code block evaluation differs from C5, see example above.

Related:

PROCLINE(), PROCFILE(), PCALLS(), PROCSTACK()

PROCSTACK ()

Syntax:

`retC = ProcStack ([expN1], [expC2], [expL3], [expL4])`

Purpose:

Returns a string corresponding to the current callstack.

Options:

<**expN1**> is a numeric value representing the depth in the calling sequence. Zero (the default) means the current UDP/UDF name, 1 the procedure and function which called the present one and so on.

<**expC2**> is a string used as separator. If not specified, the default is "`->`" or "`<-`" depending on the selected stack direction

<**expL3**> is a logical value specifying the stack direction. A true value .T. is the default and means "forward" and return "`MAIN/111->SUB1/222->SUB2/333`", whilst .F. return the backward direction "`SUB2/333<-SUB1/222<-MAIN/111`" starting in the current UDF Sub2()

<**expL4**> is a logical value specifying including source file name in the output, the default value is .F. When .T. is specified, the callstack output is then e.g. "`MAIN(myapp.prg)/111->SUB1(other.prg)/222->SUB2(MyProc.prg)/333`" or vice versa, depending on <expL3>.

Returns:

a string corresponding to the current callstack.

Description:

ProcStack() is mainly used for debugging purposes. It returns the current callstack trace including the source file, UDF or UDP name and the line number, optional with the source name.

For more information, refer to PCALLS(), PROCFILE(), PROCNAME(), PROCLINE().

Classification:

programming

Compatibility:

New in FS5, the 4th parameter is new in FS8.

Related:

PROCNAME(), PROCLINE(), PROCFILE(), PCALLS()

PROPER ()

Syntax:

retC = Proper (expC1, [expC2], [expA3])

Purpose:

Set the first character to upper case for all words in <expC1>, except words in <expA3> and all other chars in <expC1> to lower case

Arguments:

<**expC1**> is the string to be converted

<**expC2**> is a string of delimiters; default delimiters are ".,:;'- "

<**expA3**> is a 1-dimensional array of character elements, containing words which should not be converted. It also may contain a specific conversion mask:

- if the element is "sTr*" converts " strNext" to " sTrnext"
- if the element is "sTr+" converts " strnext" to " sTrNext"

The default is {"von","Mc+"}, which may be disabled by assigning {} or {""} or {NIL} as 3rd parameter.

Returns:

<**retc**> is the translated string.

Classification:

programming

Compatibility:

New in FS5

Related:

Upper(), Lower()

PROW ()

Syntax:

`retN = PROW ()`

Purpose:

Reports the current printer row position.

Returns:

<**retN**> is a numeric value representing the current printer head line, starting from zero.

Description:

PROW() reports the row position of the printhead after the last print operation. PROW() is updated only if either SET DEVICE TO PRINTER or SET PRINTER ON is in effect. PROW() is the same as ROW() except that it relates to the printer rather than the screen.

EJECT, besides advancing the paper, resets PCOL() and PROW() to zero. SETPRC() sets PROW() and PCOL() to any value without causing the paper to move.

Example:

```
SET DEVICE TO PRINT
EJECT
@ PROW(), 10 SAY "First line"
@ PROW()+1, 10 SAY "Next line"
SET DEVICE TO SCREEN
```

Classification:

programming

Compatibility:

Note the standard spooled output used by FlagShip, see LNG.5.1.6, LNG.3.6 and SET PRINTER command. Regardless of printer redirection, the PROW() is set the same way as for a direct printer output.

Related:

PCOL(), ROW(), SETPRC(), EJECT, SET DEVICE, SET PRINT

PUSHBUTTON()

Syntax:

```
retO = PUSHBUTTON ([expN1], [expN2], [expC3],
   [expL4], [expB5], [expL6])
```

Purpose:

Creates PushButton, same as instantiating the object via PushButton{…} class.

Options:

<**expN1**> is a vertical coordinate (row) of the upper left edge. Either in pixels or col/row coordinates, dependent on the current state of SET PIXEL on|OFF or the <expL6> parameter. Modifiable by oRet:Row or oRet:Y(nRow,[lPixel]).

<**expN2**> is a horizontal coordinate (column) of the upper left edge. Either in pixels or col/row coordinates, dependent on the current state of SET PIXEL on|OFF or the <expL6> parameter. Modifiable by oRet:Col or oRet:X(nCol,[lPixel]).

<**expC3**> is the informational text to be printed in the button. If not given, null-string "" (i.e. no text) is displayed. See additional details in oPushButt:Caption.

<**expL4**> sets this button as default, i.e. button that is activated when the users hits the Enter or Return key. If this argument is not specified or of other value than .T., the button can only be activated by mouse click. To perform the "click" by keyboard, use Tab or cursor keys in conjunction with Return or space key.

<**expB5**> is an optional code block, equivalent to oPushButton:Notify assignment. Processed on button click. To activate this codeblock, subsequent oPushBut:Show() or oPushBut:Display() is required. The current object is passed to the codeblock.

<**expL6**> if true(.T.), the row and column data are in pixel; if false (.F.), data are in row/col coordinates, if not specified the current SET PIXEL is used.

Returns:

<**retO**> is an instantiated PushButton object variable, which instances can be additionally assigned/executed. At least oRet:Show() needs to be executed to display the button.

Description:

Creates PushButton icon to be processed by mouse click (or keyboard) and displays it by subsequent invocation of oRet:Show(). See other properties in section OBJ.

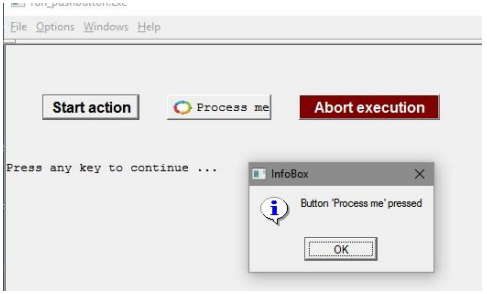
Example 1:

See examples in section OBJ.PushButton and CMD.@...GET PUSHBUTTON

Example 2: In addition to the @...GET PUSHBUTTON, the PushButton can also be used stand-alone independent of READ:

```
LOCAL oPush1, oPush2, oFont
PRIVATE oPush3
SET FONT "Courier", 10              // default font
oFont := FontNew("Arial",12,"B")    // font for PushButton (GUI)
```

```
oPush1 := PushButton(3,5,"Start action", .T. , ;
                     {|obj| myAction(obj, .T.)} )
oPush1:Font := oFont             // button font (GUI)
// oPush1:GuiColor := "B/G+"
oPush1:Width(strlen2pix(oPush1:Caption) +10, .T.)
oPush1:Height(28, .T.)
oPush1:Show()                    // display/process button

oPush2 := PushButton(3,22,"Process me", , {|obj| myAction(obj)} )
oPush2:Bitmap := "process.gif"   // if bitmap n/a, display caption
oPush2:Show()                    // display/process button

oPush3 := PushButton(3,40,"Abort execution")
oPush3:Notify := {|obj| myAction(obj)} // handler
oPush3:Colorspec := "W+/R,R/GR"  // button color (terminal)
oPush3:GuiColor  := "W+/R,R/GR"  // button color (GUI)
oPush3:Font := oFont             // button font  (GUI)
oPush3:Enable(.F.)               // disable button 3 until "Start"
oPush3:Tooltip := "This button is yet DISABLED"
oPush3:Show()                    // display/process button

setpos(6,0)
wait

FUNCTION myAction(obj, lDefa)    // processed by click on button
LOCAL cInfo := "Button '" + obj:Caption + "' pressed"
if valtype(lDefa) == "L" .and. lDefa
   cInfo += ";this is the default button"
endif
if !obj:Enable()
   cInfo += ";Button is yet disabled"
endif
if left(obj:Caption, 5) == "Start"  // if "Start" was clicked,
   oPush3:Enable(.T.)               // enable button 3
   oPush3:Tooltip := "This button is enabled now"
endif
Infobox(cInfo)                      // display message
return
```



Classification: programming
Compatibility: available in VFS8 and newer
Related: OBJ.PushButton, @...GET PUSHBUTTON

QOUT ()
QOUT2()
QQOUT ()
QQOUT2()

Syntax:

```
NIL = QOUT (expList)
NIL = QOUT2 ([expC1], [expC2], expList)
```

Syntax:

```
NIL = QQOUT (expList)
NIL = QOUTT2([expC1], [expC2], expList)
```

Purpose:

Displays a list of expressions to the console.

Arguments:

<**expList**> is a comma separated list of values or expressions to evaluate and display. The expressions can be of any data type, including memos.

If no argument is specified, QOUT() sends a NEW LINE code to the console, QQOUT() does nothing.

Options:

<**expC1**> specifies the color for displaying the <expList> data. Only the first color pair (standard) is significant. This argument is always considered in Terminal mode. If this argument is not given, the current color setting is used. In GUI mode, first the <expC2> argument is checked. If <expC2> is not set or is empty, the <expC1> value or the current color is used - but only when SET GUICOLOR is ON. Specifying COLOR and GUICOLOR allows you to handle different colors for GUI and Terminal mode without switching the SET GUICOLOR and SET COLOR setting.

<**expC2**> specifies the color for displaying the <expList> data considered in GUI mode. Only the first color pair (standard) is significant. If <expC2> is not empty, it is used regardless the current SET GUICOLOR on/off setting. If omitted and SET GUICOLOR is ON, either the <expC1> is used if not empty, or the current SetColor() is used. The <expC2> value apply for GUI mode only, and is ignored otherwise.

Returns:

The functions always return NIL.

Description:

QOUT() is the equivalent function of the ? command, while QQOUT() is the same as the ?? command. Both are console functions to display the results of one or more expressions. QOUT() outputs a linefeed (the NEW LINE code) before displaying the expressions, while QQOUT() displays the results of <expList> at the current ROW() and COL() position.

The advantage of QOUT() and QQOUT() is their usage within an expression or code block, or in iterative functions like AEVAL() or DBEVAL().

Redirection of QOUT() and QQOUT() output to PRINTER, ALTERNATE, and EXTRA text files or devices is supported in the same way as with the ? and ?? commands. ROW() and COL() are updated while being displayed to the console, and PROW() and PCOL() are updated while being displayed to the printer.

In GUI mode, you may include RichText/HTML tags into the output string and either use SET HTMLTEXT ON or preface the string by "<HTML>" to interpret the tags. See more in SET HTMLTEXT.

Example:

```
LOCAL arr := {1, 2, "text", .T., DATE() }, ii := 1
AEVAL (arr, {|x| QOUT("arr[",STR(ii++,1),"]"), QQOUT(x) })

Qout2("r/b", "r+/w", "Hello", "world")
```

Classification:

sequential output

Compatibility:

Available in FS and C5 only. Embedded zero bytes are supported in FS by default. Qout2() and Qqout2() is available in FS5 only.

Related:

?, ??, @...SAY, TEXT, COL(), ROW(), SET CONSOLE, SET PRINTER, SET ALTERNATE, SET EXTRA, SET HTMLTEXT, SYS.2.7

RAT ()

Syntax:

retN = RAT (expC1, expC2)

Purpose:

Searches through a character string for the last instance of the substring given.

Arguments:

<**expC1**> is the character string to look for.

<**expC2**> is the character string to search through.

Returns:

<**retN**> is a numeric value representing the position of the first letter of the last instance of the given substring. If the search is not successful, RAT() returns zero.

Description:

RAT() is similar to AT() but searches from right to left to locate the last instance of the substring. RAT() is also like the $ operator which determines whether a substring is contained within a string.

Example:

```
RAT("cd", "abcdabcdabcd")              && 11
AT ("cd", "abcdabcdabcd")              &&  3
RAT("ef", "abcdabcdabcd")              &&  0
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

$, AT(), STRTRAN(), SUBSTR(), LEFT(), RIGHT()

RDDLIST ()

Syntax:

`retA = RDDLIST ( )`

Purpose:

Returns an array of linked in RDD drivers.

Returns:

<**retA**> is an array, containing the names of available RDDs.

Description:

RDDLIST() is used to check if a specific RDD can be used. You may alternatively use ISFUNCTION ("<rddName>NEW") instead.

Example:

```
? RDDsetDefault()                          // DBFIDX
RDDsetDefault("Cb4cdx")
aeval (RDDLIST(), {|x| QQOUT(x)})          // DBFIDX CB4CDX CB4MDX

isMdxok := ISFUNCTION ("cb4mdxNEW")        // CB4MDX linked?

isMdxok := (ascan (RDDLIST(), ;            // alternative
             {|x| upper(x) == "CB4MDX"}) > 0)

if isMdxok
   RDDsetDefault("Cb4mdx")
endif
use mydbf1                                 // "Cb4mdx" RDD is used
```

Classification:

database

Compatibility:

available in FS, C5 and VO.

Source:

<FlagShip_dir>/system/rddsys.prg

Related:

USE, DBUSEAREA(), RDDSETDEFAULT(), RDD.2.2

RDDNAME ()

Syntax:

`retC = RDDNAME ( )`

Purpose:

Returns the name of the RDD driver used in the current work area.

Returns:

<**retC**> is a string containing the RDD name.

Description:

RDDNAME() is used to determine the current RDD. If you wish to get the name in another work area, use (alias)->(RddName()).

Example:

```
use mydbf1 alias olddb new via CB4CDX
use mydbf2 alias newdb new

? RddName()                          // DBFIDX
? (olddb)->(RddName())               // CB4CDX

? DbObject():RddName                 // DBFIDX
? (olddb)->(DbObject():RddName)      // CB4CDX
```

Classification:

database

Compatibility:

available in FS5.

Related:

USE, RDDLIST(), DBUSEAREA(), RDDSETDEFAULT(), RDD.2.2

RDDSETDEFAULT ()

Syntax:

retC = RDDSETDEFAULT ([expC1])

Purpose:

Returns and optionally change the default RDD for the application.

Options:

<**expC1**> is the name of the RDD that will be used to activate and manage all subsequent database actions, when no RDD is explicitly specified. You may specify null string "" to reset it to the first announced driver.

Returns:

<**retC**> returns the current RDD used, when <expC**1**> was not given, otherwise the name of previously RDD set.

Description:

The default "DbfIdx" driver is set automatically in the INIT (user modifiable) procedure RDDINIT, available in the rddsys.prg file (see details in the section RDD). You may invoke RDDSETDEFA(), or the equivalent DBSETDRIVER() also later to set your preferred database driver. To force the linker to include the new driver into your executable, specify also EXTERNAL <rddName>NEW.

Example:

```
? RDDsetDefault()                        // DBFIDX
use mydbf1                               // std. RDD is used
** EXTERN Cb4cdxNEW                      // if RDD not expl.linked

if ascan(RDDLIST(), {|x| x == "CB4CDX"}) == 0
   alert("sorry - enable EXTERN... to link the Cb4Cdx driver")
   QUIT
endif
? RDDsetDefault("Cb4cdx")                // DBFIDX
? RDDsetDefault()                        // CB4CDX
use mydbf2                               // "Cb4cdx" RDD is used
set index to myindex1, myindex2          // *.cdx indices used
use mydbf3 VIA ("DbfIdx")                // std RDD is temp.used
set index to myindex3, myindex4          // *.idx indices used here
```

Classification:

database

Compatibility:

available in FS, C5 and VO.

Source:

<FlagShip_dir>/system/rddsys.prg

Related:

USE, DBUSEAREA(), DBSETDRIVER(), RDDLIST(), oRdd:Driver

READEXIT ()

Syntax:

`retL = READEXIT ([expL])`

Purpose:

Enables or disables the down-arrow and up-arrow keys to be exit keys from a READ.

Options:

<**expL**> enables (if TRUE) or disables (if FALSE) the down-arrow and up-arrow keys to be exit keys from a READ. The default setting is FALSE.

Returns:

<**retL**> is a logical value representing the current setting on entering the function.

Description:

READEXIT() can be set TRUE in case of a READ which will usually be browsed through without editing.

Example:

```
old_exit = READEXIT(.T.)
cur_dat = DATE()
@ 1,0  SAY "Enter the correct date:" GET cur_dat
READ
READEXIT(old_exit)
```

Classification:

programming

Compatibility:

To provide compatibility to dBASE III, execute a READEXIT(.T.) at the beginning of the main procedure.

Related:

READ, READINSERT()

READGETPOS ()

Syntax:

retN = READGETPOS ([expA1], [expC2], [expC3], [expN4], [expN5])

Purpose:

Returns GET element number within the GetList array.

Options:

<**expA1**> is an array of GET objects. If not specified, the default GetList variable is used.

<**expC2**> is the variable name looking for.

<**expC3**> is a caption text of @GET item looking for. If given, the <expC2> must be NIL.

<**expN4**>,<**expN5**> are @GET coordinates of item looking for. If given, <expC2> and <expC3> must be NIL.

When no parameters are passed or only <expA1> is given, the current GET item position is returned.

Returns:

<**retN**> is the current item or item number found in the GetList array in range 1 to len(expA1). On error, 0 is returned.

Description:

ReadGetPos() returns the currently processed GET by default READ, or searches for specified object property. It may be used in UDF specified by SET KEY or by WHEN or VALID clause of @..GET/READ.

Example:

```
SET KEY K_F7 to myUdf1
set font "courier"
var1 := name := addr := space(20)
@ 1,0 say "Press F7 to check GETs"
@ 5,2 say "var1" get var1 when myUdf2(getlist)
@ 6,2 say "name" get name valid myUdf2()
@ 7,2 say "addr" get addr
READ
setpos(18,0)
wait

FUNCTION myUdf1(cProc, nLine, cVarname)
return myUdf2()

FUNCTION myUdf2(aGet)
local obj, iPos, rr := row(), cc := col()
if empty(aGet)
   aGet := GetList
endif
```

```
@ 10,0 clear
@ 10,0 say "Item 'var1' is GET number : " + ;
           ltrim( ReadGetPos(aGet, "var1") )
@ 11,0 say "Item 'name' is GET number : " + ;
           ltrim( ReadGetPos(aGet, "name") )
@ 12,0 say "Item 'addr' is GET number : " + ;
           ltrim( ReadGetPos(aGet, "addr") )
@ 13,0 say "Item 'foo'  is GET number : " + ;
           ltrim( ReadGetPos(aGet, "foo") )
iPos := ReadGetPos(aGet, , , 6, 7)
@ 14,0 say "Item at coordinate 6,7 is : " + ltrim( ipos )
if iPos > 0 .and. len(aGet) > 0
   obj := aGet[iPos]
   ?? " and is of type", obj:ClassName(), ;
      "with VarName", upper(obj:Name)
endif
@ 16,0 say "Current GET item number   : " + ;
           ltrim( ReadGetPos(aGet) )
obj := GetActive()
@ 17,0 say "Current item is at coordinate " + ;
           ltrim(obj:row) + "," + ltrim(obj:col)
?? " and is of type", obj:ClassName(), ;
   "with VarName", upper(obj:Name)
setpos(rr,cc)
return .T.
```

Classification:

programming

Compatibility:

Available in VFS 7 and newer.

Source:

available in <FlagShip_dir>/system/getsys.prg

Related:

READ, ReadModal(), ReadSelect()

READINSERT ()

Syntax:

retL = READINSERT ([expL])

Purpose:

Toggles the current insert mode setting.

Options:

<**expL**> toggles the insert mode on (if TRUE) and off (if FALSE). The default setting is .F. or the last user-selected mode in READ or MEMOEDIT().

Returns:

<**retL**> is a logical value representing the last status before the current toggling.

Description:

READINSERT() can be used to set the insert mode before, or to restore the old setting after a READ or MEMOEDIT(). The insert mode is a global setting belonging to the current application and not the specific operation.

READINSERT() can be executed prior to or during a READ or MEMOEDIT(). If used with READ, it can be invoked within a WHEN or VALID clause of @...GET or within a SET KEY procedure. If used with MEMOEDIT(), it can be invoked with the user function as well as a SET KEY procedure.

When Set(_SET_INSERT_INKEY) is disabled (the default), the Insert mode is toggled in READ and MemoEdit() by press on the [INSERT] key. When Set(_SET_INSERT_INKEY) is enabled, you may toggle the insert mode also by key press and Inkey() outside of READ and MemoEdit().

To determine the current state of Insert mode, use ReadInsert(), Set(_SET_INSERT) or InkeyStatus(STATUS_INSERT) functions.

Example:

```
READINSERT(.T.)
? IF(READINSERT(),"On","Off")           && "On"
```

Classification:

programming

Related:

READ, MEMOEDIT(), READEXIT(), SET KEY, @...GET, InkeyStatus()

READKEY ()

Syntax:

`retN = READKEY ()`

Purpose:

Determines what key terminated a READ.

Returns:

<**retN**> is a numeric value representing the key pressed to exit a READ, according to dBASE convention:

Exit Key	retN	Updated	INKEY()
Up-arrow	5	261	5
Down-arrow	2	258	24
PgUp	6	262	18
PgDn	7	263	3
Ctrl-PgUp	31	287	31
Ctrl-PgDn	30	286	30
Esc	12	268	27
Ctrl-W	14	270	23
Type past end	15	271	ASC(key)
Return	15	271	13

Description:

READKEY() emulates the same named dBASE III+ function. Its purpose is to determine what key the user pressed to terminate a READ. If UPDATED() is TRUE, READKEY() returns the code plus 256.

READKEY() has been superseded by LASTKEY() which determines the last keystroke fetched from the keyboard buffer. To determine the update state of READ, issue the UPDATED() function.

Example:

```
@ 10,0 GET name
@ 11,0 say first
READ
? READKEY()
? LASTKEY(), UPDATED()
```

Classification:

programming

Source:

available in <FlagShip_dir>/system/getsys.prg

Related:

LASTKEY(), UPDATED(), READ, READEXIT(), INKEY() codes

READKILL ()

Syntax:

`retL = READKILL ( [expL] )`

Purpose:

Set/get the READ termination flag.

Options:

<**expL**> enables (if TRUE) or disables (if FALSE) a flag terminating the current READ.

Returns:

<**retL**> is a logical value representing the current termination flag. Upon normal condition, the return value is .F., except CLEAR GETS or ReadKill(.T.) was executed which then return .T. The ReadKill() status remain valid until the next READ is invoked.

Description:

ReadKill(.T.) is very similar to CLEAR GETS; both will terminate the current READ performed via standard getsys.prg. But as opposite to CLEAR GETS, ReadKill(.T.) will not delete the GetList[] entries when invoking ReadSave(.T.), so you may re-issue READ anew without filling the GET fields - except when READ CLEAR or DESTROY was used.

Example:

Terminates the READ by F5 key or entry "exit".

```
LOCAL var2 := 10.5, var1 := "text   "
SET KEY K_F5 to MyExitRead()
@ 1,2 GET var1  VALID if(trim(var1)=="exit",MyExitRead(),.T.)
@ 2,2 GET var2
READ
if ReadKill()       // killed via F5 key or "exit" entry?
   if empty(var1)
      alert("you need to enter any value in the first field")
      READ
   endif
endif
wait color "r+" guicolor "r+"
FUNCTION MyExitRead(p1,p2,p3)
   local iChoice
   iChoice := alert("Terminate READ?", ;
                    {"Ignore","on next key","Now"} )
   do case
   case iChoice == 2
      ReadKill(.T.)            // terminate READ
   case iChoice == 3
      keyboard chr(K_ENTER)    // simulate Enter key
      ReadKill(.T.)            // terminate READ
   endcase
   if empty(var1) .and. iChoice >= 2
      ReadSave(.T.)            // don't clear GETs, re-used
   endif
```

```
return .T.
```

Classification:

programming

Compatibility:

Available in Clipper 5.x and FS5.

Source:

available in <FlagShip_dir>/system/getsys.prg

Related:

READ, CLEAR GETS

READMODAL ()

Syntax:

```
retL = READMODAL (expA1, [expL2], [expL3], [expL4],
                  [expL5], [expL6], [expL7], [expCN8] )
```

Purpose:

Activates a full-screen editing mode for an array of GET objects, as the READ command for the default GETLIST array.

Arguments:

<**expC1**> is an array containing a list of GET objects to edit. Usually, the array elements of `GETLIST[]` array are filled automatically by the @...GET command and executed by READ command.

In FlagShip, the READ and ReadModal() support also other objects like TopBar, MenuItem, RadioButton, RadioGroup, CheckBox, ListBox etc.

<**expL2**> is equivalent to the CLEAR | DESTROY clause of READ. If .T., clears GUI Get widgets on exit, otherwise let the widgets visible same as in Terminal i/o (default)

<**expL3**> is equivalent to the ALIGN | NOLAIGN clause of READ. If .T., aligns Get fields with GuiAlign(), setting .F. will omit aligning, otherwise consider the SET GUIALIGN setting (default)

<**expL4**> is equivalent to the SKIPOVER | NOSKIPOVER clause of READ. It allows to skip over validated fields when .T., otherwise stay in the field which does not meet the VALID criteria (def). Skipping over a field can mostly occur in GUI by mouse click on different field or widget in the GetList.

<**expL5**> is equivalent to the EXITCHECK clause of READ. If .T., check all VALID conditions at exit, otherwise don't check over- skipped fields on exit (the default)

<**expL6**> is equivalent to the SAVE clause of READ. If .T., the `GetList[]` elements will not be deleted on exit. You may force this behavior by invoking ReadSave(.T.) during READ process.

<**expL7**> is equivalent to the CYCLE clause of READ. If .T., moving forward over the last GET item, or backward over the first GET will not terminate READ but move to first or last item respectively. The default value is .F.

<**expCN8**> is equivalent to the SELECT clause of READ. It causes READ to start with GET item specified by variable name <expC8>, or with item number <expN8> within the GetList array. If the @GET item is disabled, next enabled item is used. If SELECT is not given, READ starts at the first enabled item within current @..GET list. See also ReadSelect() function.

Returns:

The function always returns NIL.

Description:

READMODAL() is a user interface function that implements the full- screen editing mode for GETs, and is part of the open architecture GET system, compatible to C5.

READMODAL() is like the READ command, but takes any GETLIST (or other named) array as an argument and does not reinitialize the GETLIST[] array when it terminates. Hence, using this function, multiple lists of GET objects can be maintained simultaneously.

Using @...GET/READ commands, a PUBLIC array called GETLIST is used. When the program starts up and when the READ command is executed without a SAVE option, the array is cleared to zero length. Each time an @...GET command executes, it creates a GET object and adds it to the currently visible GETLIST array.

Using a PRIVATE, LOCAL or STATIC array named GETLIST, nested GET/READs are possible to any depth. The other alternative is to use any named array, creating the GET object by the GETNEW() class function and filling the object using export variables. Refer to the section OBJ.GET for more information.

In the source file **getsys.prg** (usually stored in the <FlagShip_dir>/system directory), the following modifiable functions are available to customize the GET/READ system:

GETACTIVE()	Return the currently active GET object
GETAPPLYKEY()	Apply a key to a GET object from within a GET reader
GETDOSETKEY()	Process SET KEY during GET editing
GETPOSTVALIDATE()	Post-validate the current GET object
GETPREVALIDATE()	Pre-validate a GET object
GETREADER()	Execute standard READ behavior for single GET object
GETPREVALIDATE()	Pre-validate a GET object
READFORMAT()	Return, and optionally set the code block that implements a format (.fmt) file
READKILL()	Return, and optionally set whether the current READ should be exited
READMODAL()	Execute standard READ behavior for an array of GET objects
READUPDATED()	Return, and optionally set whether a GET has changed during a READ

Refer also to the section OBJ.GET for more information about the class functions and export variables.

Example 1:

Create two GET objects in array named MYGETS and issue an independent READ.

```
LOCAL v1 := 0, v2 := "abcd", mygets[2]
mygets[1] := GETNEW (2,0, {|x| IF(x==NIL,x,v1:= x)}, "v1")
mygets[2] := GETNEW (3,0, {|x| IF(x==NIL,x,v2:= x)}, "v2")
READMODAL (mygets)
```

Example 2:

The same example, but nested GET/READs are used instead.

```
LOCAL v1 := 0, v2 := "abcd", getlist := {}
@ 2,0 GET v1
@ 3,0 GET v2
READ
```

Classification:

screen oriented input/output, output bufferable via DISPBEGIN() ... DISPEND()

Compatibility:

available in C5 (param 1 only). Parameters 2 to 6 are new in FS5

Class:

uses GET class objects, prototyped in <FlagShip_dir>/include/ getclass.fh

Source:

available in <FlagShip_dir>/system/getsys.prg

Related:

@..GET, READ, <FlagShip_dir>/include/std.fh, OBJ.2

READSAVE ()

Syntax:

`retL = READSAVE ( [expL] )`

Purpose:

Set/retrieve the flag signaling clear of GetList[] during exit of READ.

Options:

<**expL**> signals that the GetList[] elements should be kept (if TRUE) at exit of the current READ.

Returns:

<**retL**> is a logical value representing the current status of a flag signaling to clear GetList[] entries during exit of READ. Upon normal condition, the return value is .F., except READ SAVE was invoked which return .T. and will not clear the GetList[] data. The ReadSave() status remain valid until the next READ is invoked.

Description:

ReadSave(.T.) is very similar to READ SAVE; both will avoid the deletion of GETs from the GetList[] array on exit from READ. But as opposite to READ SAVE, the ReadSave() flag is temporarily only and is reset to .F. by the next READ invocation. You may invoke ReadSave(.T.) during a process of the current READ when you wish to re-use all already specified @..GET entries in subsequent READ.

Example:

see ReadKill().

Classification:

programming

Compatibility:

Available since FS5.

Source:

The function is available in <FlagShip_dir>/system/getsys.prg

Related:

READ, READ SAVE, ReadKill()

READSELECT ()

Syntax:

retN = READSELECT ([expA1], expCN2)

or:

retN = READSELECT (expCN2)

Purpose:

Select specified GET element during READ.

Arguments:

<**expA1**> is an array of GET objects. If not specified, the default public/private GetList variable is used.

<**expCN2**> is either string containing variable name, or numeric value specifying item number within <expA1> to be selected. The numeric is either relative to current GET position (+/-expN2, where for example -1 will select previous item, or 2 the over-next item), or absolute item position when 1000 is added to, i.e. expN2 = n+1000 will select GetList[n] and returns n.

Returns:

<**retN**> is the newly selected item in the GetList array in range 1 to len(expA1). On error, 0 is returned.

Description:

With ReadSelect() you may "skip" to new GET item, other than in the default order, once the entry in current GET field is finished. It may be used in UDF specified by SET KEY or by WHEN or VALID clause of @..GET/READ. The WHEN and VALID clauses of skipped-over fields are not recognized, except you specify NOSKIPOVER clause in READ. When the resulting item skips below 1, first item is selected; when it skips behind last <expA1> item, the last item is selected.

Example:

```
#include "getexit.fh"  // for GE_* values
LOCAL getlist := {}
LOCAL first := name := addr := space(20)
LOCAL lSave := lExit := .F.
@ 5, 2 say "first" get first
@ 6, 2 say "name " get name valid checkName(name, first, getlist)
@ 7, 2 say "addr " get addr
@ 5,40 GET lSave PUSHBUTTON caption "Save" SKIP ;
                  notify {|obj| obj:OnClickAction := GE_WRITE}
@ 7,40 GET lExit PUSHBUTTOcanntN caption "Exit" SKIP ;
                  notify {|obj| obj:OnClickAction := GE_ESCAPE }
          // or:  notify {|obj| obj:OnClickKeys := chr(K_ESC) }
READ
setpos(16,0)
wait

/* ensure first name is filled too when name is not empty,
 * or jump to Exit button if name and first are empty
 */
```

```
FUNCTION checkName(value_name, value_first, getlist)
local iNewPos := 0
if !empty(value_name) .and. empty(value_first)
   iNewPos := ReadSelect(getlist, -1)
elseif empty(value_name) .and. empty(value_first)
   iNewPos := ReadSelect(getlist, "lExit")
endif
return .T.
```

Classification:

programming

Compatibility:

Available in VFS 7 and newer.

Source:

available in <FlagShip_dir>/system/getsys.prg

Related:

READ, ReadModal(), ReadGetItem()

READUPDATED ()

Syntax:

`retL = READUPDATED ( [expL] )`

Purpose:

Determines if there was a change in any of the processed GETs during the last or the current READ or set this flag manually.

Options:

<**expL**> sets or clears the READ Updated flag manually.

Returns:

<**retL**> is logically TRUE if there were changes, or FALSE if there were none.

Description:

ReadUpdated() is equivalent to UPDATED() except it lets you change the current update flag. Per default, this flag is set .T when any characters were successfully entered into a GET from the keyboard during the last or the current READ.

Each time READ executes, UPDATED() is set to FALSE. Then, any change to a GET entered from the keyboard or pushed to the keyboard buffer by KEYBOARD sets UPDATED() to TRUE. This happens after the GET exits or before a VALID clause or the SET KEY procedure is executed. Changes of the get:BUFFER in a post-valid function will be not recognized by UPDATED().

Once UPDATED() is set to TRUE, it retains this value until the next READ is executed or manually reset by ReadUpdated(.F.)

Example:

see Updated().

Classification:

programming

Compatibility:

Available in Clipper5 and FS5.

Source:

The function is available in <FlagShip_dir>/system/getsys.prg

Related:

READ, Updated()

READVAR()

Syntax:

`retC = READVAR ()`

Purpose:

Retrieves the name of the current GET or MENU variable.

Returns:

<**retC**> is a character string containing the name of the current GET or MENU variable in uppercase. If there are other wait statuses, than the READ or MENU TO, a null string "" is returned.

Description:

READVAR() can be used in SET KEY procedures, and in VALID or WHEN functions to provide menus or context sensitive help depending on the current GET or MENU TO variable. It returns the variable name during execution of MENU TO and READ command, otherwise (and after CLEAR GETS which terminates READ) a null string "" is returned.

Tuning:

If you wish to hold the last used GET variable name also outside of READ, set

```
  _aGlobSetting[GSET_L_READVAR_HOLD] := .T.   // default is .F.
```

Example:

A simple context-specific help:

```
@ 1,2 GET name WHEN myhelp()
@ 2,2 GET city WHEN myhelp()
READ

FUNCTION myhelp
LOCAL isvar := READVAR()
@ 20,0 CLEAR
DO CASE
CASE isvar == "NAME"
  @ 21, 0 SAY "Enter the user name ..."
CASE isvar == "CITY"
  @ 21, 0 SAY "Enter the city ..."
ENDCASE
RETURN .T.
```

Classification:

programming

Related:

@..GET, READ, MENU TO, SET KEY

RECCOUNT ()

Syntax:

retN = RECCOUNT ()

or:

retN = LASTREC ()

Purpose:

Retrieves the number of physical records in the current database file.

Returns:

<**retN**> is a numeric value representing the number of records which are physically present in the current database file. SET FILTER and SET DELETED ON have no effect on the return value of RECCOUNT().

If there is no database in use in the current working area, zero is returned.

Description:

RECCOUNT() and LASTREC() are equivalent database functions which determine the number of physical records in the current database.

To execute the function in a working area different from the current one, use alias->(RECCOUNT()).

Example:

```
USE article
? RECNO(), RECCOUNT(), LASTREC()            &&   1 100 100
GOTO BOTTOM
? RECNO(), RECCOUNT(), EOF()                && 100 100 .F.
SKIP
? RECNO(), RECCOUNT(), EOF()                && 101 100 .T.
APPEND BLANK
? RECNO(), RECCOUNT(), EOF()                && 101 101 .F.
SET FILTER TO Theme = "Computers"
COUNT TO Others
? Others, RECCOUNT()                        &&  77 101
```

Classification:

database

Related:

LASTREC(), SET FILTER, SET DELETED, USE, oRdd:RecCount

RECNO ()

Syntax:

`retN = RECNO ()`

Purpose:

Retrieves the current record number of the selected working area.

Returns:

<**retN**> is a numeric value representing the current physical record number. If no database is open, RECNO() will return a zero.

If the database is empty, RECNO() returns 1, EOF() and BOF() both return TRUE, and LASTREC() returns zero.

If an attempt was made to move past the first record, BOF() returns TRUE and RECNO() returns 1. If the record pointer is moved past the last record in the database (i.e. on a phantom record), EOF() returns TRUE and RECNO() returns LASTREC()+1.

Description:

RECNO() is a database function that returns the current record number in the current working area. Each database file is physically ordered by a record number starting from 1.

The record numbering allows direct access to a record without sequentially scanning the database file to reach the specified record position using the GOTO command. A related command is SET RELATION...TO RECNO().

To execute the function in a working area different from the current one, use alias->(RECNO()).

Example:

```
USE Article
GO BOTTOM
? RECCOUNT(), RECNO(), EOF()                    && 100 100 .F.
SKIP
? RECCOUNT(), RECNO(), EOF()                    && 100 101 .T.
GO 52 ; ? RECCOUNT(), RECNO(), EOF()       && 100  52 .F.
GOTO TOP
? RECCOUNT(), RECNO(), BOF()                    && 100   1 .F.
SKIP -1; ? RECCOUNT(), RECNO(), BOF()      && 100   1 .T.
```

Classification:

database

Related:

BOF(), EOF(), RECCOUNT(), LASTREC(), GOTO, SKIP, oRdd:Recno

RECSIZE ()

Syntax:

`retN = RECSIZE ()`

Purpose:

Retrieves the record size of the current database file.

Returns:

<**retN**> is a numeric value representing the size in bytes of the database record in the current working area. If there is no database in use, zero is returned.

Description:

RECSIZE() is a database function that determines the length of a record by summing the field lengths, and adding one byte for the record's deleted status.

The size of the database file can then be calculated by multiplying RECSIZE() by RECCOUNT() and then adding HEADER().

To execute the function in a working area different from the current one, use alias->(RECSIZE()).

Example:

```
USE article
? "The size of one record is", ltrim(RECSIZE()),"bytes"
filesize = HEADER() + (RECCOUNT() * RECSIZE())
? "Total", DBF(), "database size is", ltrim(filesize), "bytes"
wait
```

Classification:

database

Related:

DISKSPACE(), HEADER(), RECCOUNT(), oRdd:RecSize

REPLICATE ()

Syntax:

`retC = REPLICATE (expC1, expN2)`

Purpose:

Forms a new character string by repeating the given string the indicated number of times.

Arguments:

<**expC1**> is the character string to repeat.

<**expN2**> is the number of times to repeat <expC1>.

Returns:

<**retC**> is the newly-created character string. If the numeric argument <expN2> is negative or zero, a null string "" is returned.

Description:

REPLICATE() can be used where a repetition of the same string is needed. REPLICATE() is like the SPACE() function which returns a specified number of space characters. Also PADx() functions expand existing strings with given characters or spaces.

Example:

```
KEYBOARD REPLICATE(CHR(13), 4)          // move to the
@ ... GET...                            // fifth field
READ

? REPLICATE("*-",3) + "*"               // *-*-*-*
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

SPACE(), PADR(), PADC(), PADL()

RESTSCREEN ()

Syntax 1:

```
NIL = RESTSCREEN ([expN1],[expN2],[expN3],[expN4],
   varS5, [expL6])
```

Syntax 2:

```
NIL = RESTSCREEN (varS5)
```

Purpose:

Displays a screen region saved in a memory variable at a specified area of the screen.

Arguments:

<**expN1**...**expN4**> defines the top, left, bottom and right window coordinates to restore. If the arguments are omitted, 0, 0, MAXROW(), MAXCOL() is assumed. If <expL6> is not set, the GUI coordinates are calculated according to current SET FONT (or oApplic:Font)

If the screen contents in <varS5> were saved without coordinates to preserve the entire screen, no screen coordinates are necessary with RESTSCREEN(). In such a case, RESTSCREEN(,,,,var) or RESTSCREEN(var) performs the same functionality.

When the coordinates were given but are out of range, no action is taken and a warning message displays with set FS_SET("devel",.T.).

<**varS5**> is a variable of the type SCREEN containing the saved screen region.

<**expL6**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

Returns:

The function always returns NIL.

Description:

RESTSCREEN() can be used to display a screen region saved with SAVESCREEN(), at the same or different position on the screen. The new screen region must be the same size as the saved one, or you may get ambiguous results.

Typical usage of RESTSCREEN() is to allow the user to drag a window around the screen, or to save and restore small portions of screens that will be ruined by displaying small message boxes.

Performance Hints *for Terminal i/o*

The time required for redrawing the screen is proportional to the line speed setting (stty) and the size of re-drawn screen. Usually, Curses optimizes the output and will replace (transmit) characters different from current ones or characters with a different foreground or background color only.

In any case, saving and restoring a partial screen may speed the output significantly. To avoid "flickering" on a slow communication line, you may buffer the output via DISPBEGIN() ; RESTSCREEN(..) ; DISPEND().

You may tune the behavior of RestScreen() by setting

```
_aGlobSetting[GSET_L_RESTSCR_CLS] := .F.    // default is .T.
```

which avoids clearing the restored area beforehand and is therefore significantly faster.

Example:

Display an external text using MEMOEDIT:

```
scr1 = SAVESCREEN(10, 5, 14, 60)
MEMOEDIT (MEMOREAD ("extern.txt"), 10,5, 14,60, .F.)
RESTSCREEN(10, 5, 14, 60, scr1)
```

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND(). Usable only with enabled screen oriented output (the default, see sect. SYS.2.7).

Compatibility:

In FlagShip, SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported for any terminal (using the window structure of curses by or internal GUI bitmap). To modify the saved screen contents, see the function _retscw() in section EXT (for Terminal i/o mode only).

Syntax 2 is not supported by C5.

To save the contents of a screen, FlagShip uses the variable of type "S" in contrast to the Clipper's type "C", which are not reversely compatible. See also compatibility notes in RESTORE FROM command, if screen variables are saved/restored using .mem file as well as the conversion functions SCRDOS2UNIX() and SCRUNIX2DOS() .

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode, see SaveScreen() for details. Parameter 6 is new in FS5

Related:

SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), SCRDOS2UNIX(), SCRUNIX2DOS(), SCREEN2CHR(), CHR2SCREEN()

RIGHT ()

Syntax:

retC = RIGHT (expC1, expN2)

Purpose:

Extracts the specified number of characters from the right of a character string.

Arguments:

<**expC1**> is the character string to extract characters from.

<**expN2**> is the number of characters to extract from the right side.

Returns:

<**retC**> is a character string, which consists of <expN2> rightmost characters of <expC1>. If <expN2> is less than 1, a null string "" is returned. If <expN2> is greater than the length of <expC1>, the entire string is returned.

Description:

RIGHT() is the same as SUBSTR() with a negative first numeric argument.

The complementary operation to RIGHT() is LEFT().

Example:

```
? SUBSTR("John is right", -5)          && right
? RIGHT ("John is right", 5)           && right
```

Classification:

programming

Compatibility:

Embedded zero bytes are supported by default.

Related:

LEFT(), AT(), RAT(), LTRIM(), RTRIM(), STUFF(), SUBSTR(), PADR(), FS2:ReplRight(), FS2:RemRight(), FS2:RemAll(),FS2:PosRepl()

RLOCK ()

Syntax:

retL = RLOCK ([expA1])

or:

retL = RLOCK ([expN1], [expN2,..])

or:

retL = LOCK ()

Purpose:

Locks the current or specified record(s) in the selected working area to enable exclusive write access in multiuser mode.

Options:

<**expA1**> is an array of numeric values, specifying the record numbers to be locked. If SET MULTILOCKS is OFF, only the first element is considered. The current record pointer RECNO() remains unchanged.

<**expN1**> is a record number or list of records to be locked. If SET MULTILOCKS is OFF, only the first parameter is considered. The current record pointer RECNO() remains unchanged.

If no parameters, NIL or an empty array is specified, the current record is locked.

Returns:

<**retL**> is a logical value, which signals success or error. If TRUE, the record(s) is (are) successfully locked. Otherwise, the record (or some records from the list) could not be locked. RLOCK() will return FALSE when another user/process:

- has successfully FLOCKed the database,
- has successfully RLOCKed the same record,
- has APPENDed this record without consequently UNLOCKing it.

Description:

RLOCK() is a network function that locks the current or specified record(s), preventing other users from updating the record(s) until the lock is released.

After a successful RLOCK() or LOCK(), the record(s) may be changed using an assignment to a field variable, REPLACE, DELETE, RECALL or @..GET../READ.

For each invocation of RLOCK(), there is one attempt to lock the current or specified record (of the list), and the result is returned as a logical value. A movement in the database does not affect the record lock setting.

If the user has locked the database by FLOCK(), the file lock remains active and RLOCK() return TRUE. Note, for a large array of records, the FLOCK() should be preferred, since it may be significantly faster and requires only one system locking slot (see also the Compatibility paragraph below).

RLOCK() is effective only on the database in the selected working area (and the associated .dbt and .idx files), not on related databases. To execute the function in a working area other than the current one, use alias->(RLOCK()).

To determine if the record or file is locked, without activating the locking, use ISDBRLOCK() or ISDBFLOCK() functions. For a list of records locked by the current application, use DBRLOCKLIST().

Multiple record locking

For multiple record changes (e.g. using the NEXT, FOR or WHILE clause), either FLOCK() or SET MULTILOCKS ON have to be used.

When SET MULTILOCKS is OFF (the default), any attempt to RLOCK(), AUTORLOCK(), FLOCK() or APPEND BLANK will release all previous locks set by the current application. A record remains locked until another record is locked, an UNLOCK or DBRUNLOCK() is executed, the current database file is closed, the program terminated, or a FLOCK() is obtained on the current database.

When SET MULTILOCKS is set ON, any consecutive attempt to RLOCK(), AUTORLOCK() or APPEND BLANK will add the record number to an internal list of locked records, while FLOCK() releases all previous record locks first. Note, each user or application has its own locking table. The records remain locked until an UNLOCK or DBRUNLOCK() is executed, the current database file is closed, the program terminated, or a FLOCK() is obtained on the current database.

Note, that one (system) lock slot is used for locking of a group of consecutive records (in natural order), while the non- consecutive records in the rlock-list require one lock slot each (see also sect. SYS.3). Using the FLOCK() may be significantly more effective, than a large rlock-list.

Multiuser:

In multiuser mode (SET EXCLUSIVE is OFF or USE.. ..SHARED), the record or file must be locked to allow a **write** access to the record or the whole file. Otherwise, if not locked by the programmer, FlagShip invokes AUTORLOCK() if possible (i.e. if SET AUTOLOCK is ON, which is the default).

Note, that RLOCK() or AUTORLOCK() will not check, whether the data of the current (or specified) record was changed by others since the last read access. Therefore, it may be safer to perform RLOCK() before reading/accessing the data to ensure, you are always working on the current database record (see example below); or use RLOCKVERIFY() to check it.

RLOCK() provides a shared lock, allowing other users to read the locked record, but they cannot write to it.

The lock is not necessary for a read access only, like the SKIP, SEEK, GOTO commands, or on assigning a field to a memory variable, etc. but may be used to write-protect an array of records, e.g. during a complex calculation or transaction on them.

If RLOCK() is used on a database opened READONLY, several users / applications may successfully lock the same data, if they also use the database in READONLY

mode. The default read/write database access is stronger and allows only one RLOCK() for the same record, regardless of the open mode.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

See also SET COMMIT for tuning.

Example 1:

```
SET EXCLUSIVE OFF                        && multiuser mode
USE address
WHILE NETERR()                           && success ?
   USE address                           && no, try again
END
SET INDEX TO address
WHILE NETERR()                           && success ?
   SET INDEX TO address                  && no, try again
END

APPEND BLANK                             && automatic RLOCK
WHILE NETERR()                           && success ?
   @ 0,0 SAY "attempt to APPEND record"
   APPEND BLANK                          && no, try again
ENDDO
REPLACE name WITH "Smith"                && write access
UNLOCK
GOTO BOTTOM
DO WHILE .not. RLOCK()                   && wait until RLOCK
   @ 0,0 SAY "waiting for RLOCK"         && come trough,
   INKEY (1)                             && with small delay
   @ 0,0 SAY SPACE(40)                   && and user info
ENDDO

IF ! DELETED()                           && = read access
   DELETE                                && = write access
ELSE
   RECALL                                && = write access
ENDIF
UNLOCK
```

Example 2:

The usage of RELATIONs and writing to multiple databases:

```
LOCAL name
SELECT 5
USE custom INDEX custom SHARED
IF NETERR()
   ? "cannot open custom.dbf"
   RETURN
ENDIF
SELECT 1
USE address SHARED
IF NETERR()
   ? "cannot open custom.dbf"
   RETURN
ENDIF
SET RELATION TO FIELD->name INTO custom
```

```
GOTO TOP
BEGIN SEQUENCE
   IF !RLOCK()                          && lock address.dbf
     BREAK WITH "address"
   ENDIF
   IF ! (custom->(RLOCK()))             && lock custom.dbf
      BREAK with "custom"
   ENDIF
   REPLACE count WITH count + 1
   REPLACE custom->turnover WITH 0
   UNLOCK ALL                           && unlock area 1 and 5
RECOVER USING name
   ? "cannot RLOCK file", name + ".dbf - no update"
END
```

Example 3:

Multiple record locking, e.g. to replace a region of records, while other records are modifiable by other users. See also additional examples in the SET MULTILOCKS description.

```
LOCAL ii, aLockNum := {}
USE address INDEX addr1,addr2 SHARED NEW
SEEK "MŠLLER"
if !found() ; return ; endif
while !eof() .and. left(upper(Name),6) == "MŠLLER"
   aadd(aLockNum, recno())
   skip
enddo
SET MULTILOCKS ON
while !RLOCK(aLockNum)                    // locks an array
   ? "trying to lock an array of records"
enddo
FOR ii := 1 to LEN(aLockNum)
   GOTO aLockNum[ii]
   REPLACE name with "Mueller"
NEXT
UNLOCK                                    // unlock all
```

Example 4:

Multiple record locking, the same example as above in other programming technique

```
LOCAL ii, aList
USE address INDEX addr1,addr2 SHARED NEW
UNLOCK
SEEK "MŠLLER"
if !found() ; return ; endif
SET MULTILOCKS ON
while !eof() .and. left(upper(Name),6) == "MŠLLER"
   RLOCK()                                // add lock to list
   skip
enddo
aList := DBRLOCKLIST()                    // retrieve the list
FOR ii := 1 to LEN(aList)
   GOTO aList[ii]                         // replace regardless
   REPLACE name with "Mueller"            // the logical rec.no.
NEXT
UNLOCK                                    // unlock all
```

Example 5:

Note that even the fastest, usual programming technique will fail in this case, because of not matching the WHILE condition after the first replacement (the logical record number changes whilst replacing the index key). You would have to use REPLACE...FOR... without index (slow), or the GOTO aLockNum[ii] ; REPLACE... combination from the above example instead.

```
* SET MULTILOCKS OFF                          // the default
USE address INDEX addr1,addr2 SHARED NEW
SEEK "MŠLLER"
if !found() ; return ; endif
while !eof() .and. left(upper(Name),6) == "MŠLLER"
   RLOCK()                                   // lock single record
   REPLACE name with "Mueller"
   UNLOCK                                    // unlock it
   skip                                      // next logical record
enddo                                        // Index != MŠLLER
```

Example 6:

Use "early" locking to ensure that the data was not changed by others during the editing process:

```
USE stock INDEX stkartno SHARED NEW
SEEK article
if found() .and. RLOCK()                     // lock before access
   aRecData := FIELDGETARR()
   for ii := 1 to len(aRecData)
      @ ii,0 GET aRecData[ii]
   next
   READ                                      // modify data
   FIELDPUTARR(aRecData)                     // and store back
   COMMIT
   UNLOCK                                    // unlock record
endif
```

Classification: database

Compatibility:

FlagShip uses Unix and Windows compatible locking methods (via fcntl). The usage of RLOCK(), FLOCK() and UNLOCK is nevertheless source- and logic-compatible to other Xbase dialects. FlagShip, Clipper, dBase, and Foxbase/FoxPro locking is incompatible to each other. The check using ISDBRLOCK() and the **AutoLock** feature is available in FS only. Multiple record locks are not available in C5, but are compatible to FoxPro and VO. Note, the FoxPro's character parameter specifying the area is not supported; use alias->(RLOCK(...)) instead.

Refer to section SYS.3 (tunable Unix parameters) to set or increase the number of maximal locks available. For consecutive (physical) records, only one system lock is required, while for non-consecutive records, each lock engages one system lock slot.

Related:

ISDBRLOCK(), DBRLOCKLIST(), DBRUNLOCK(), USE..EXCLUSIVE, SET COMMIT, SET EXCLUSIVE, UNLOCK, FLOCK(), ISDBFLOCK(), SET AUTOLOCK, SET MULTILOCK, RLOCKVERIFY(), oRdd:Rlock(), DBRLOCK()

RLOCKVERIFY ()

Syntax:

`retL = RLOCKVERIFY ( )`

Purpose:

Determines if the current record is still unmodified since last accessed by the current application and whether a record update is safe. If so, the record is RLOCKed. This is a superset of the RLOCK() function.

Returns:

<**retL**> is a logical value, which signals the successful lock on an unmodified record.

If TRUE, the record contents is equal to that of the last field access/assign/positioning of the current application, and the record is successfully RLOCKed for subsequent field REPLACEment.

FALSE signals, that the record was changed by another user or process in the meantime, the subsequent RLOCK and replacement may not be safe and therefore no RLOCK is issued. The application should then determine the changed field to ensure a correct transaction, see example below. Since the fields are not automatically updated after RLOCKVERIFY(), you may store the "old" data, issue SKIP(0) and retrieve the new values. A FALSE value may also signal, that RLOCK() failed.

Description:

RLOCKVERIFY() is a network function that allows you late locking with safe replacement. It first checks the data of the current record, if it is still the same as when the last field was accessed (by this application), or got changed by another user in the meantime. If all the record data is equal to the "old" state, RLOCK() is performed.

Otherwise, the data got changed in the meantime, or the RLOCK() failed. See the example below, for how to compare the "old" and "new" record data.

Multiuser:

See detailed information about locking in the RLOCK() description. RLOCKVERIFY() offers you an alternative (late) locking method, as opposed to the (early) locking via RLOCK(). If the database is opened in EXCLUSIVE mode, RLOCKVERIFY() always returns TRUE.

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
LOCAL anyfiled, cName, aOldData, aNewData, dele
LOCAL ii, ok := .T.
USE address SHARED
GOTO 5
cName := address->Name            // field access
anyfield := FIELDGET(2)           // the last field access
```

```
WAIT                                // the record is NOT
                                    // locked and may be
? cName, anyfield                   // changed by others

if RLOCKVERIFY()
   REPLACE ....                     // nothing was changed
   UNLOCK
else
   ? "Cannot replace, because:"
   aOldData := FIELDGETARR()        // save old data
   dele := deleted()                // and deleted status
   SKIP 0                           // re-read the record
   aNewData := FIELDGETARR()        // get new data
   for ii := 1 to len(aOldData)
      if !(aOldData[ii] == aNewData[ii])
         ? "Field " + fieldname(ii) + " changed from", ;
           aOldData[ii], "to", aNewData[ii]
         ok := .F.
      endif
   next
   if dele != deleted()
      ? "Record was " + if(!deleted(),'un-','') + 'deleted'
      ok := .F.
   endif
   if ok
      ? "RLOCK() failed, try later"
   endif
endif
```

Classification:

database

Compatibility:

This function is available in FlagShip only. It is equivalent to oRDD:RlockVerify() method, available also in VO.

Related:

RLOCK(), oRdd:RlockVerify()

ROUND ()

Syntax:

`retN = ROUND (expN1, expN2)`

Purpose:

Rounds the result of the specified numeric expression to the specified number of decimal places.

Arguments:

<**expN1**> is the numeric expression to round.

<**expN2**> is the number of decimal places to return. Specifying a zero or negative argument allows rounding the whole number digits.

Returns:

<**retN**> is a numeric value, representing the argument rounded to the specified number of decimal places.

The display of the return value depends on the SET FIXED state. If SET FIXED is ON, the display is determined by SET DECIMALS.

Description:

ROUND() is a numeric function that rounds the given value up or down to specified decimal digits, when a number has to be used with less precision than it originally was.

Digits between five to nine, inclusive, are rounded up. Digits below five are rounded down.

To truncate a floating point value down to integer, INT() may be used.

Example:

```
SET FIXED ON
SET DECIMALS TO 2
? ROUND(1.414, 1)                    &&    1.40
? ROUND(1.514, 1)                    &&    1.50
? ROUND(1.414, 0)                    &&    1.00
? ROUND(10.4,  0)                    &&   10.00
? ROUND(10.5,  0)                    &&   11.00
? ROUND(15.14, -1)                   &&   20.00
? ROUND(1234.567, -3)                && 1000.00
```

Classification:

programming

Related:

INT(), INT2NUM(), NUM2INT(), SET FIXED, SET DECIMALS, STR(), VAL()

ROW ()

Syntax:

```
retN = ROW ([expL1], [expN2])
```

Purpose:

Reports the current row where the cursor is located on the screen.

Options:

<**expL1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col,otherwise the current SET PIXEL status is used.

<**expN2**> is optional numeric value to change the Row() value.

Returns:

<**retN**> is a numeric value, representing the current cursor row in the range between zero and MAXROW(). When the global variable `_aGlobSetting[GSET_L_ROUND-ROWCOL]` is set .T. (default is .F., see initio.prg), the returned <retN> value is rounded to integer. When SET COORD UNIT is set to PIXEL, MM, CM or INCH, and <expL1> is not given, the returned value corresponds to these units (GUI only). If not set, the GUI row coordinate corresponds to current SET FONT (or oApplic:Font)

Description:

ROW() returns the cursor row position as an numeric value. The value of ROW() is updated by both console and full-screen commands and functions sent to screen, even if SET CURSOR is OFF.

ROW() is used with COL() and all variations of the @ command to position the cursor to a new line relative to the current line.

ROW() is related to PROW() and PCOL() which track the current printhead position instead of the screen cursor position.

If you are using sub-windows via standard MDIopen() or the Wopen() from FS2 Toolbox, Row() and Col() reports the cursor position in the current sub-window.

In GUI mode, any output is pixel oriented. The ROW(.T.) return value therefore corresponds to the real display position, i.e. where the next output displays. For your convenience and to achieve cross compatibility to textual based applications, FlagShip supports also coordinates in common row/col values. When <expL1> is not given and SET PIXEL is OFF, and SET COORD UNIT is not set, the returned ROW() value corresponds to manual Pixel2row(Row(.T.)) calculation.

When you change the font size or select different font name using SET FONT command or the FONT class, the current ROW() position remain unchanged, but the next line and ROW() coordinate may differ from the current setting. If you wish to align characters on it base line, use SET ROWALIGN and/or SET ROWADAPT commands. See further details in LNG.5.3, LNG.5.3.1, LNG.5.3.2, Row2Pixel() and SET FONT.

In Terminal and Basic i/o mode, the row height is fix 1 independent on the used font, hence ROW() and ROW(.T.) returns the same value in row/col coordinates.

Example:

```
@ 10,10 SAY "123"
@ ROW()+1,10 SAY "456"
@ MAXROW(),0 SAY "Help"
@ ROW(),COL()+1 SAY "message"
```

Classification:

programming, for screen oriented output

Compatibility:

For a terminal independent application, it is wise to use MAXROW() instead of the fixed screen length, since some Unix terminals support only 24 instead of 25 lines.

The return value is usable only with enabled screen oriented output (the default, see sect. SYS.2.7).

The optional parameter is new in FS5

Related:

COL(), MAXCOL(), MAXROW(), PCOL(), PROW(), @...SAY, DEVPOS(), SETPOS(), SYS.2.7

ROW2PIXEL ()

Syntax:

```
retN = Row2pixel ([expN1], [expO2])
```

Purpose:

Calculates pixels of given row corresponding to the given or current i/o Font.

Arguments:

<**expN1**> is a numeric constant or variable specifying the row size which should be recalculated to pixels. If not given or < 0, the pixel size of one row is returned.

<**expO2**> is a Font object to be used for the pixel calculation. If not specified, the default font set by SET FONT is used.

Returns:

<**retN**> is an integer value corresponding to the height of given rows in pixels.

Description:

In GUI mode, any output is pixel oriented. For your convenience and to achieve cross compatibility to textual based applications, FlagShip supports also coordinates in common row/column values. It then internally re-calculates the given rows by using Row2pixel() and columns by using Col2pixel() function. The line and character spacing is affected by the currently used default font.

This function allows you to manually calculate the Y pixel position from given row coordinate. It is applicable for ROW(), SETPOS() and other positioning commands and functions. The return value is equivalent to return of oApplic:Row2Pixel(expN1) method.

In Terminal and Basic i/o mode, the row height is fix 1 independent on the used font and hence Row2pixel(expN1) returns <expN1>.

See further details about font and coordinates in LNG.5.3, LNG.5.4, Row() and SET FONT. For minimal porting effort, best to use fixed fonts (SET FONT "Courier", 12).

Classification:

programming

Compatibility:

New in FS5

Related:

Col2pixel(), Pixel2col(), Pixel2row(), oApplic:Row2Pixel()

ROWADAPT ()

Syntax:

`NIL = RowAdapt ( )`

Purpose:

Force a ROW() adaption when the previous screen output includes HTML tags or different FONTs.

Description:

When performing screen output in GUI mode using HTML tags or with different FONTs, the COL() position is calculated automatically, but the ROW() setting considers the
 and <P> HTML tags or the larger/smaller font only when SET ROWADAPT is ON. Otherwise the ROW() remain unchanged - or is increased by one line height (using the default font size) when the ? command or QOUT() was invoked.

When SET ROWADAPT is OFF, you may force the ROW() adaption manually by invoking RowAdapt() just after the output. When SET ROWADAPT is ON, RowAdapt() takes no additional adaption.

The SET ROWADAPT and RowAdapt() adaption takes effect only for sequential screen output (i.e. for ?, ??, @...SAY commands and Qout(), Qqout(), DevOut(), DevOutPict() functions) in GUI mode.

Classification:

programming, screen output

Compatibility:

New in FS5

Related:

?, ??, @..SAY, SET HTMLTEXT, SET ROWADAPT, Qout(), Qqout()

ROWVISIBLE ()

Syntax:

`retN = RowVisible ([expL1])`

Purpose:

Reports the currently visible rows on the screen.

Options:

<**expL1**> is a pixel specification. If .T., the returned value is assumed in pixel, if .F. in row/col, otherwise the current SET PIXEL status is used.

Returns:

<**retN**> is a numeric value, representing the currently visible rows in the range between zero and MAXROW(). If <expL1> is not set, the GUI coordinate corresponds to current SET FONT (or oApplic:Font)

Description:

ROWVISIBLE() returns the currently visible rows in the range of zero to MAXROW(). In GUI mode, it considers the current/resized window size. In Terminal or Basic mode, the value is equivalent to MAXROW().

RowVisible() is equivalent to `m->oAppWindow:CurrSize(APP_Y_USER)`

Example:

```
? "MaxRow()=",ltrim(MaxRow(),"RowVisible()=",ltrim(RowVisible())
if AppIoMode() == "G"
   wait "Resize the screen height by mouse, then press any key..."
endif
? "MaxRow()=",ltrim(MaxRow(),"RowVisible()=",ltrim(RowVisible())
wait
```

Classification:

programming, for screen oriented output

Compatibility:

RowVisible() is new in FS6

Related:

ROW(), MAXROW(), ColVisible(), oAppWindow:CurrSize()

RTRIM ()

Syntax:

retC = RTRIM (expC|expN)

or:

retC = TRIM (expC|expN)

Purpose:

Removes all trailing spaces from a string.

Arguments:

<**expC**> is the character string to be trimmed.

<**expN**> is a numeric value which is converted to string first and then trimmed.

Returns:

<**retC**> is a copy of the string entered, with all trailing blanks removed. In case of a null string "" or all spaces, TRIM() returns a null string "".

Description:

RTRIM() and TRIM() are alternative names for the same function. They are useful for removing trailing blanks when several strings are used to form an expression like an address or title.

Related operations of RTRIM() are LTRIM(), which removes leading spaces, and ALLTRIM(), which removes both leading and trailing spaces.

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

Example:

```
first = "John    "
last  = "Smith   "
? first + last                        && "John    Smith   "
? first - last                        && "JohnSmith       "
? TRIM(first) + " " + last            && "John Smith      "
? TRIM(first) + " " + TRIM(last)      && "John Smith"

? LEN(TRIM(SPACE(20))                 && 0
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default. FS5 accept numeric parameter as well.

Related:

ALLTRIM(), LTRIM(), SUBSTR(), PADx(), FS2:RemRight()

SAVESCREEN ()

Syntax:

```
retS = SAVESCREEN ([expN1], [expN2], [expN3],
   [expN4], [expL5])
```

or:

```
retS = SAVESCREEN ()
```

Purpose:

Saves a specified screen region to a variable of type SCREEN for later display.

Options:

<**expN1**>...<**expN4**> defines the top, left, bottom and right window coordinates of the screen region to be saved. If the arguments are omitted, 0, 0, MAXROW(), MAXCOL() is assumed. If <expL5> is not set, the GUI coordinates are internally calculated in pixel from current SET FONT (or oApplic:Font)

When the coordinates were given but are out of range, no action is taken, <retS> returns NIL, and descriptive warning message displays with set FS_SET("devel",.T.).

<**expL5**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used. Apply for GUI only.

Returns:

<**retS**> is a variable of type SCREEN containing the visible output and color attributes of the specified screen region. This variable cannot be used for string or arithmetic operations. It may however, be translated to a character and back, using SCREEN2CHR(), CHR2SCREEN() respectively. To manipulate the variable contents, see example in section EXT, function _retscw(). This apply for Terminal i/o mode, the bitmap representation used in GUI mode cannot be manipulated.

Description:

SAVESCREEN() saves a screen region to a SCREEN variable, in order for it to be restored with RESTSCREEN(). Restoring small screen regions using SAVE/ RESTSCREEN() is faster (and avoids the screen flash) than using RESTORE SCREEN for the whole screen.

Screen regions are usually saved and restored when using a pop-up menu routine or dragging a screen object. For the manipulation of the screen contents, see section EXT.2 (_retscr) and EXT.4. For Performance Hints, see the RESTSCR() description.

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode. In GUI, it is compressed or un- compressed bitmap object an hence cannot be extracted by Chr2Screen, Screen2chr() or other low-level curses routines like ScrUnix2Dos() and ScrDos2Unix(). It also cannot be saved to or restored from a memo file.

In GUI, you may decide if the bitmap is stored "as is" (default) or in compressed format which requires significantly less memory. On the other hand, a compressed format may cause some side-effects depending on the used graphic card and the selected color depth. If you have many Save/RestScreen() in the application, try to set SET SCRCOMPRESS ON (default is OFF) and watch if not side effects (like slight color shifting) occurs after RestScreen(), otherwise let the compression disabled by SET SCRCOMPRESS OFF.

Example:

This example saves the contents of a window while listing the contents of a database file inside the window, restoring the screen region at the end:

```
LOCAL scr1 := SAVESCREEN(15, 0, 20, 79)
@ 15,0 CLEAR TO 20,79
USE authors
DO WHILE .NOT. EOF()
   SCROLL (15, 0, 20, 79, 1)
   @ 20,10 SAY TRIM(firstname) +" " + lastname
   IF INKEY() = 27                          && Esc
      INKEY(0)                              && Pause
   ENDIF
   SKIP
ENDDO
USE
RESTSCREEN(15, 0, 20, 79, scr1)
```

Classification:

programming (for screen oriented output), usable only with enabled screen oriented output (the default, see sect. SYS.2.7).

Compatibility:

In FlagShip, SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported for any terminal (using the window structure of curses). To modify the saved screen contents, see the function _retscw() in section EXT.

For saving a screen contents, FlagShip uses the variable of type "S" in contrast to the Clipper's type "C", which are not binary compatible. See also compatibility notes in RESTORE FROM command, if screen variables are saved/restored using .mem file as well as the conversion functions SCRDOS2UNIX() and SCRUNIX2DOS().

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode, see description above. Parameter 5 is new in FS5

Related:

RESTSCREEN(), SAVE SCREEN, RESTORE SCREEN, CHR2SCREEN(), SCREEN2CHR(), SCRDOS2UNIX(), SCRUNIX2DOS()

SCRDOS2UNIX ()

Syntax:

```
retS = SCRDOS2UNIX (expN1, expN2, expN3, expN4,
   expC5, [expL6])
```

Purpose:

Converts screen contents in MS-DOS format (created e.g. by SAVESCEREEN() in Clipper) into FlagShip format.

Available in Terminal i/o only, accepted as dummy in GUI and basic mode. When FS_SET("devel") is set .T., GUI and Basic mode raises developer warning.

Arguments:

<**expN1**>...<**expN4**> are the top, left, bottom and right window coordinates of the stored screen region. These coordinates do not need to exactly match those of SAVESCREEN(), but only the window size (rows * columns) has to match.

<**expC5**> is a string containing the screen contents of the stored screen area <expN1> ... <expN4> in MS-DOS format. The significant string size is always (expN3 - expN1 +1) * (expN4 - expN2 + 1) * 2; rest of the string is ignored.

Option:

<**expL6**> is a logical switch controlling the display. If FALSE, only a conversion is performed for later display by RESTSCREEN(). If not specified or given TRUE, the converted screen contents is displayed at coordinates <expN1>...<expN4>.

Returns:

<**retS**> is a variable of type SCREEN containing the screen contents converted to Unix WINDOW format, including color and special character conversion. This return value may be then re- displayed/restored by the standard FlagShip function RESTSCREEN(). On conversion error, NIL is returned. <retS> is NIL in GUI and basic mode.

Description:

SCRDOS2UNIX() is a screen conversion routine. It allows to display screen contents stored on MS-DOS system into a character variable and restored on Unix e.g. from a database field or .mem variable.

Since the string of a DOS screen contents may contain binary zeroes, it is recommended to set fs_set("zero", .T.) while manipulating the string. Also, storing the string into a .mem field may cause problems in some circumstances, since the 1Ahex (26dec) is a valid color attribute. But you may use SCREEN2CHR() for such purposes.

Although the combination of SCRUNIX2DOS() and SCRDOS2UNIX() may be used instead of SAVESCREEN() and RESTSCREEN() combination, the latter pair should be preferably used, since it is significantly faster and more comfortable.

Note, that problems (screen garbage) may occur when the Unix terminal does not support the size of the stored screen area. See also example below.

GUI mode: The ScrDos2Unix() cannot be used in GUI mode, where the screen variable <varS> is compressed or uncompressed bitmap object an hence cannot be extracted or converted by this function.

Example:

Displays screen contents stored in database fields on MS-DOS system.

```
FIELD scrfld, bottom, right
USE scrdata

#ifndef FlagShip
// ----- MS-DOS part -----
   append blank
   replace scrfld with SAVESCREEN(0,0,24,79)   // len= 4000
   replace bottom with 24
   replace right  with 79
#else
// ----- Unix part -----
   Local scrVar
   Local saveZero := fs_set ("zero", .T.)
   if bottom > MAXROW()
      ? "cannot restore",bottom +1, ;
        "lines on terminal with",MAXROW()+1,"lines"
   elseif right > MAXCOL()
      ? "cannot restore",right +1, ;
        "columns on terminal with",MAXCOL()+1,"columns"
   else
      scrVar := ScrDos2Unix (0,0,bottom,right, scrfld, .F.)
      if valtype(scrVar) != "S"
         ? "Error on restoring - window size ?"
      else
         RestScreen(0,0,bottom,right, ScrVar)
      endif
   endif
   fs_set ("zero", saveZero)
#endif
```

Classification:

programming (for screen oriented output), usable only with enabled screen oriented output (see sect. SYS.2.7).

Compatibility:

This function is available in FlagShip only. Embedded zero bytes are only supported, when FS_SET("zero") is set.

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode, see SaveScreen() for details. Therefore, this function is working properly only in Terminal i/o

Related:

SCRUNIX2DOS(), SAVESCREEN(), RESTSCREEN(), SCREEN2CHR(), CHR2SCREEN()

SCRUNIX2DOS ()

Syntax:

```
retC = SCRUNIX2DOS ([expN1], [expN2], [expN3],
   [expN4], [expS5])
```

Purpose:

Converts the screen contents from FlagShip format into string in MS-DOS format.

Available in Terminal i/o only, accepted as dummy in GUI and basic mode. When FS-SET("devel") is set .T., GUI and Basic mode raises developer warning.

Options:

<**expN1**>...<**expN4**> are the top, left, bottom and right window coordinates of the stored screen region. These coordinates do not need to exactly match those of SAVESCREEN(), but only the window **size** has to match. If not specified, (0, 0, maxrow(), maxcol()) is assumed. If SET COORD is not set, the GUI coordinates are internally calculated from current SET FONT (or oApplic:Font)

<**expS5**> is a SCREEN variable containing the screen contents in Unix format, e.g. from SAVESCREEN(). If not given or specified NIL, SCRUNIX2DOS() performs SAVESCREEN() at the given coordinates <expN1> ...<expN4>.

Returns:

<**retC**> is a variable of type character, containing the screen contents converted from FlagShip's format to MS-DOS, including color conversion. This return value may be then re-displayed by the standard Clipper function RESTSCREEN(). The returned string size is always (expN3 - expN1 +1) * (expN4 - expN2 + 1) * 2. On error, null string is returned. <retC> is "" in GUI and basic mode.

Description:

SCRUNIX2DOS() is a screen conversion routine. It allows to extract text or color attributes from screen contents. In the string <retC> is every odd (uneven) byte the text character, every even byte the color attribute.

Storing the string into a memo field may cause problems in some circumstances, since the 1Ahex (26dec) is a valid color attribute. But you may use SCREEN2CHR() for such purposes.

GUI mode: The ScrUnix2Dos() cannot be used in GUI mode, where the screen variable <expS> is compressed or uncompressed bitmap object an hence cannot be extracted or converted by this function.

Example 1:

Extract text from partition of screen

```
 // Note: this works in Terminal i/o mode, does NOT work in GUI
mode!
 set color to "W+/B"
 cls
 @ 3,10 say "Hello world!"
```

```
@ 4,10 say "Next line"

xScr := SaveScreen(3,10,4,22)
cScr := ScrUnix2dos(3,10,4,22, xScr)
cTxt := ""
for ii := 1 to len(cScr) -1 step 2
   cTxt += substr(cScr, ii, 1)
next
setpos(10,0)
? "text='" + cTxt + "'"      // 'Hello world! Next line    '
wait
```

Example 2:

```
/*****************************************************************
  * PrintScr(r1,c1, r2,c2, file) -> .T. or .F.
  * PrintScreen to text file, applicable in Terminal i/o mode only
  * In GUI mode, you may use _oPrinter:ExecPrintScreen(.T.) instead
  *
  * r1,c1 = optional coordinates top left
  * r2,c2 = optional coordinates bottom right
  * file  = name of the text file
  */
 FUNCTION printscr(r1,c1, r2,c2, file)
    local ii, rlen, clen, scr, ctext, cc, linelen

    if AppIoMode() != "T"              // Terminal i/o mode?
       return .F.                      // no, exit
    endif

    r1 := if(valtype(r1) != "N", 0, r1)
    c1 := if(valtype(c1) != "N", 0, c1)
    r2 := if(valtype(r2) != "N", maxrow(), r2)
    c2 := if(valtype(c2) != "N", maxcol(), c2)
    file := if(valtype(file) != "C" .or. empty(file), ;
               "printscr.txt", file)

    rlen := r2 -r1 +1
    clen := c2 -c1 +1

    scr := savescreen(r1,c1, r2,c2)    // save (partition of) screen
    cText := scrunix2dos(0, 0, rlen, clen, scr)  // convert to text

    set printer to (file)
    set printer on
    set console off

    linelen := 0
    for ii := 1 to len(cText) step 2   // extract characters
       cc := substr(cText,ii,1)
       linelen++
       if asc(cc) == 0
          ?
          linelen := 0
       elseif linelen > clen
          ? cc
```

```
         linelen := 1
      else
         ?? cc
      endif
   next
   set printer off
   set console on
   return .T.
```

Classification:

programming (for screen oriented output), usable only with enabled screen oriented output (see sect. SYS.2.7).

Compatibility:

This function is available in FlagShip only. Embedded zero bytes are only supported, when FS_SET("zero") is set (default).

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode, see SaveScreen() for details. Therefore, this function is working properly only in Terminal i/o

Related:

SCRDOS2UNIX(), SAVESCREEN(), RESTSCREEN(), SCREEN2CHR(), CHR2SCREEN()

SCREEN2CHR ()

Syntax:

`retC = SCREEN2CHR (varS)`

Purpose:

Converts a screen variable to a character string. This function is applicable for Terminal i/o mode only.

Arguments:

<**varS**> is a SCREEN variable (type "S") generated with SAVE SCREEN TO or SAVESCREEN().

Returns:

<**retC**> is a character string containing the converted variable of type "S".

Description:

SCREEN2CHR() can be used to store a screen variable into a memo field, character database field, etc.

The inverse function of SCREEN2CHR() is CHR2SCREEN().

GUI mode: The Screen2chr() cannot be used in GUI mode, where the screen variable <varS> is compressed or uncompressed bitmap object an hence cannot be extracted or converted by this function.

Example:

```
myscreen = SAVESCREEN (10,5, 20,40)
// any screen operation
RESTSCREEN (10,5, 20,40, myscreen)

USE initdata SHARED
SET INDEX TO userno
SEEK user
REPLACE memofield WITH CHR2SCREEN (myscreen)
// and later
RESTSCREEN (10,5, 20,40, SCREEN2CHR (memofield))
```

Classification:

programming (for screen oriented output), usable only with enabled screen oriented output (see sect. SYS.2.7).

Compatibility:

This function is available in FlagShip only. The returned string is also not compatible to the result of SAVESCREEN() in MS-DOS (use SCRUNIX2DOS() instead). To modify the contents of the screen variable, see example in section SYS, function _retscw(), applicable for terminal mode only.

In GUI, the structure of the screen variable <retS> is incompatible to <retS> from Terminal i/o mode, see SaveScreen() for details.

Related: CHR2SCREEN(), RESTSCREEN(), SAVESCREEN(), SCRDOS2UNIX(), SCRUNIX2DOS( )

SCROLL ()

Syntax:

```
NIL = SCROLL ([expN1], [expN2], [expN3], [expN4],
        [expN5], [expN6], [expL7])
```

Purpose:

Clears or scrolls a specified screen region up and down, or right and left.

Options:

<**expN1**>...<**expN4**> defines the top, left, bottom and right window coordinates of the screen region to be scrolled. If the arguments are omitted, 0, 0, MAXROW(), MAXCOL() is assumed. If <expL7> or SET COORD is not set, the GUI coordinates are internally calculated from current SET FONT (or oApplic:Font)

<**expN5**> defines the vertical scrolling direction. If it is positive, the window scrolls up for the specified number of lines. If it is negative, the window scrolls down. If it is zero, or neither <expN5> nor <expN6> are specified, or NIL, the region is cleared.

<**expN6**> defines the horizontal scrolling direction. If it is positive, the window scrolls left for the specified number of columns. If it is negative, the window scrolls right. If it is zero, or <expN5> and <expN6> are both not specified or NIL, the region is cleared.

<**expL7**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used.

Returns:

The function always returns NIL.

Description:

SCROLL() is a screen function that scrolls a screen region up or down a specified number of rows. When a screen scrolls up, the first line of the region is erased, all other lines are moved up, and a blank line is displayed in the current standard color on the bottom line of the specified region. If the region scrolls down, the operation is reversed. If the screen region is scrolled more than one line, this process is repeated.

Scrolling the screen right, a space is inserted at the left of each line of the region, the right characters moved out are lost. Scrolling left, the operation is reversed, inserting spaces to the right.

SCROLL() can be used to implement listing on a part of the screen, while the rest of the screen remains still.

GUI note: you cannot scroll-out widgets (like @..GET, ListBox, PushButton, Tbrowse etc.), but plain output (like ?, ??, Qout(), @..SAY, @..DRAW etc.) only, see also section LNG.5.3. If you wish to clear widgets, use CLS, CLEAR or @..CLEAR TO..

Example:

```
USE authors
@ 10,10 CLEAR TO 15,40
@ 10, 10 TO 15,40 DOUBLE
DO WHILE .NOT. EOF()
   SCROLL (11,11,14,39,1)
   @ 14,12 SAY lastname
   IF INKEY(0.5) = 27                    && Pause
   SCROLL (11,11,14,39,1)
      @ 14,12 SAY  "..any key to continue or ESC to exit.."
      IF INKEY(0) = 27                   && Exit
         EXIT
      ENDIF
   ENDIF
   SKIP
ENDDO
USE
```

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND()

Compatibility:

The optional sixth argument is new in FS and C5. Note, FlagShip handles the screen output using the default UNIX curses library. It sends control escape sequences to the terminal according to the current TERM description in the compiled terminfo file. This is similar to the usage of ANSI driver on DOS. Therefore, scrolling may be significantly slower than the direct updating of the video buffer used in MS-DOS dialects. On the other hand, it works with any terminal, connected any way, even by serial cable or by modem.

Parameter 7 is new in FS5

Related:

@...CLEAR, CLEAR SCREEN, @...TO...DOUBLE, @...BOX

SECONDS ()

Syntax:

`retN = SECONDS ([expL1])`

Purpose:

Reports the number of seconds elapsed since midnight, or continued since 1-January-1970.

Options:

<**expL1**> is optional logical value. If true (.T.), the return value represents seconds (and fraction of seconds if possible) since January 1, 1970, localized. If false (.F.), the return value are un-localized (UTC) seconds since 1/1/1970, same as the system function time().

Using this argument is advisable for long time measurements to avoid manual handling of the midnight wrap, and for measurements with a high accuracy.

Returns:

<**retN**> is a numeric value, representing the seconds according to the system time, based on the 24 hour clock. The resolution depends on the used system and is usually in milliseconds or microseconds. In some older UNIX, only full second intervals can be reported.

Description:

SECONDS() is used to calculate the length of time intervals. It is related to the TIME() function which returns the system time as a string in the form "hh:mm:ss" or optionally as "hh:mm:ss.uuuu".

If only seconds intervals are returned on your system, and smaller time intervals on a millisecond basis are needed and not supported by seconds(), use SECONDSCPU() - see the compatibility note below.

Example:

```
old_sec = SECONDS()
SET PRINTER TO /dev/lp0
printMyDoc()
elap_time = SECONDS() - old_sec
? "Printer was in action for ", elap_time, " seconds."
```

Classification:

programming, system

Compatibility:

In some older Unix systems (not so in Linux), the elapsed seconds since midnight are available without decimals only (see UNIX "time"). To determine the true CPU time for measuring purposes, SECONDSCPU() should be used instead, which also returns hundredths of seconds and the **true** performance. In contrast to SECONDSCPU(), the SECONDS() do not give relevant information on the program performance in multiuser and multitasking environments.

For the most newer Unix systems, Linux and for MS-Windows, Seconds() returns fractions of seconds. The optional parameter is available in FS5, FS6 and newer.

On Unix/Linux, the system functions time(), localtime(), and when available also gettimeofday() or curtime() are used internally, refer to "man 2 time" and "man 3 localtime" for further details. On MS-Windows, the system functions time(), localtime() and GetLocalTime() are used.

Related:

TIME(), SECONDSCPU(), SLEEP(), SLEEPMS() FS2:SecToTime(), FS2:TimeToSec(), FS2:WaitPeriod()

SECONDSCPU ()

Syntax:

`retN = SECONDSCPU ([expN])`

Purpose:

Reports how many CPU and/or system seconds have elapsed since the beginning of the program execution.

Options:

<**expN**> is an optional numeric expression which controls the counter information (see also "times" in the UNIX manual):

expN	tms_	Action
1	utime	user CPU time of the current process
2	stime	system CPU time behalf of the current process
3	u + s	sum of utime + stime (the default setting)
11	cutime	sum of the user CPU time of the current + child process (e.g. RUN program)
12	cstime	sum of the system CPU time of the current + child process (e.g. RUN program)
13	cu + cs	sum of cutime + cstime

If no argument is given, 3 is assumed, so SECONDSCPU() will return the sum of the CPU and system access time of the current process.

Returns:

<**retN**> is a numeric value, representing the required CPU time (see table above) of the program execution. The format is "seconds.hundredths". The number returned is the number of CPU seconds which have elapsed since the program start.

Description:

SECONDSCPU() is used to determine, how much CPU time the current program has received since the program start. In heavily-loaded Unix systems, the total execution time of a short program may result in a long SECONDS() period, whereas the real CPU execution time of this process as measured by SECONDSCPU() or the system routine "time" is only a few milliseconds.

The lowest measured time unit is one "clock tic", which is system dependent and often 1/60 seconds (0.016 seconds). See also the UNIX manual "times".

Example:

Measuring of the elapsed and CPU time within an application:

```
full_time = SECONDS()
cpu_time  = SECONDSCPU(1)
sys_time  = SECONDSCPU(2)
tot_time  = SECONDSCPU(3)

FOR i = 1 to 500
  ? "line", i
```

```
NEXT

cpu_time  = SECONDSCPU(1) - cpu_time
sys_time  = SECONDSCPU(2) - sys_time
tot_time  = SECONDSCPU(3) - tot_time
full_time = SECONDS() - full_time

? "Total time spend       =", full_time && 18
? "Total CPU time spend   =", tot_time  &&  3.26
? "Real execution time    =", cpu_time  &&  0.02
? "addit. system i/o time =", sys_time  &&  3.24
```

Classification:

programming, system

Compatibility:

SECONDSCPU() is not available with Clipper. Because of the single- user-process from MS-DOS, the difference of two SECONDS() functions there will return comparable performance results to the difference of two SECONDSCPU() on Unix. For compatibility to C5, use:

```
#ifndef FlagShip
#define SECONDSCPU SECONDS
#endif
```

For compatibility to other xBase dialects, add a small routine, which is included in <FlagShip_dir>/fstodos.prg, to the DOS program:

FUNCTION secondscpu parameters p1 return SECONDS()

Related:

TIME(), SECONDS()

SELECT ()

Syntax:

retN = SELECT ([expC])

Purpose:

Determines the working area number of the current or for specified alias.

Options:

<**expC**> is the alias name of the specified working area. If the argument is not specified, or is an empty string, the number of the current working area is returned.

Returns:

<**retN**> is the working area number in the range 1 to 65534. If alias <expC> does not exist, or no database is open in the current working area, zero is returned.

Description:

SELECT() is a database function that determines the working area number of an alias, or the current one. This function do **not** change the current working area.

Note the functional difference between the SELECT (expN) command, which is translated by the preprocessor to the DBSELECTAREA() function, and the similar syntax of the SELECT(expC) function. Hence, note:

```
num = SELECT ("myAlias")   // function, do not change work area
SELECT ("myAlias")         // command, changes work area!
```

So be careful or use ALIAS() to avoid confusion. The syntax `num := SELECT(alias)` is equivalent to alias->(SELECT()).

Multiuser:

When performing operations on the SAME physical database (used con- currently in different working areas), see chapter LNG.4.8.7.

Example:

```
USE article NEW ALIAS art
old_area = SELECT()                     // 1
USE magazine NEW
*  do something ...
SELECT (old_area)                       // here command SELECT !
? SELECT(), SELECT ("magazine")         // 1  2  (= functions)

cAlias := " "
? SELECT(cAlias)                        // 1 = same as SELECT()
cAlias := "magazine"
? SELECT(cAlias)                        // 2
cAlias := "art"
? SELECT(cAlias)                        // 1
cAlias := "foo"
? SELECT(cAlias)                        // 0
```

Classification: database

Related: ALIAS(), SELECT, DBSELECTAREA(), USE, RDD.1.2

SET ()

Syntax:

`ret = SET (expN1, [exp2], [expL3])`

Purpose:

Inspects or changes a system setting of the SET commands.

Arguments:

<**expN1**> is a numeric value that identifies the setting to be inspected or changed.

Options:

<**exp2**> is an optional argument that specifies a new value for the SET toggle. The type of this argument depends on the specifier <expN1>.

<**expL3**> is a logical value that indicates whether or not files are opened in the ADDITIVE mode for the commands/settings

SET ALTERNATE	SET (_SET_ALTFILE, ...)
SET PRINTER	SET (_SET_PRINTFILE, ...)
SET EXTRA	SET (_SET_EXTRAFILE, ...)

A TRUE value indicates to open the file in append mode (ADDITIVE), while a FALSE value will truncate the file, if it exists. If this argument is not specified, the default is the append mode.

Returns:

<**ret**> is the current value of the specified setting. The type of the return value depends on the specifier <expN1>. On invalid arguments, the call to SET() is ignored and NIL is returned. Note that the returned value is the previous setting when <exp2> is given and the current setting when only <expN1> was passed.

Description:

SET() is a system function that lets you inspect or change the values of the internal system settings. For more information, refer to the associated command:

Constant in set.fh	Type	Assoc. Command
_SET_ALTERNATE	Logical	SET ALTERNATE
_SET_ALTFILE	Character	SET ALTERNATE TO
_SET_ANSI	Logical	SET ANSI
_SET_AUTOCOMMIT	Logical	SET AUTOCOMMIT
_SET_AUTOLOCK	Numeric	SET AUTOLOCK
_SET_BELL	Logical	SET BELL
_SET_CANCEL	Logical	SETCANCEL()
_SET_CENTURY	Logical	SET CENTURY
_SET_CHARSET	Numeric	SET CHARSET
_SET_COLOR	Character	SETCOLOR()
_SET_CONFIRM	Logical	SET CONFIRM
_SET_CONSOLE	Logical	SET CONSOLE

_SET_COORD_UNIT	Numeric	SET COORDINATE
_SET_CURSOR	Numeric	SETCURSOR()
_SET_DATEFORMAT	Character	SET DATE
_SET_DB3COMPAT	Logical	SET DB3COMPAT
_SET_DBREAD_ISO	Logical	SET DBREAD
_SET_DBWRITE_ISO	Logical	SET DBWRITE
_SET_DEBUG	Logical	ALTD()
_SET_DEBUG_PROFILE	Numeric	(see below)
_SET_DECIMALS	Numeric	SET DECIMALS
_SET_DEFAULT	Character	SET DEFAULT
_SET_DELETED	Logical	SET DELETED
_SET_DELIMCHARS	Character	SET DELIMITERS TO
_SET_DELIMITERS	Logical	SET DELIMITERS
_SET_DEVICE	Character	SET DEVICE
_SET_EJECT	Logical	SET EJECT
_SET_EOFAPPEND	Logical	SET EOFAPPEND
_SET_EPOCH	Numeric	SET EPOCH
_SET_ESCAPE	Logical	SET ESCAPE
_SET_EVENTMASK	Numeric	SET EVENTMASK
_SET_EVENT_DURATION	Numeric	SetEvent(,n)
_SET_EVENT_INTERVAL	Numeric	SetEvent(n)
_SET_EVENT_MODE	Numeric	SetEvent(,,n)
_SET_EXACT	Logical	SET EXACT
_SET_EXCLUSIVE	Logical	SET EXCLUSIVE
_SET_EXIT	Logical	READEXIT()
_SET_FIXED	Logical	SET FIXED
_SET_FONTBOLD	Logical	SET FONT
_SET_FONTITALIC	Logical	SET FONT
_SET_FONTNAME	Character	SET FONT
_SET_FONTSIZE	Numeric	SET FONT
_SET_GOTOP	Logical	SET GOTOP
_SET_GUIALIGN	Logical	SET GUIALIGN
_SET_GUIASCII	Logical	SET GUITRANSL ASCII
_SET_GUICOLORS	Logical	SET GUICOLORS
_SET_GUICURSOR	Logical	SET GUICURSOR
_SET_GUICURSORTYPE	Numeric	SET GUICURSOR TO
_SET_GUIDRAWBOX	Logical	SET GUITRANSL BOX
_SET_GUIDRAWLINE	Logical	SET GUITRANSL LINES
_SET_GUIDRAWTEXT	Logical	SET GUITRANSL TEXT
_SET_GUILOWCP437	Logical	SET GUITRANSL LOWCP437
_SET_GUIPRINTER	Logical	SET GUIPRINTER
_SET_HTMLTEXT	Logical	SET HTMLTEXT
_SET_INPUT	Logical	SET INPUT
_SET_INSERT	Logical	READINSERT()
_SET_INSERT_INKEY	Logical	(see below)
_SET_INTENSITY	Logical	SET INTENSITY
_SET_LARGEFILE	Logical	SET LARGEFILE

_SET_LASTKEYBUFFSIZE	Numeric	(see below)
_SET_LOCK_MODE	Numeric	(see below)
_SET_LOCK_RETRY_SEC	Numeric	(see below)
_SET_MARGIN	Numeric	SET MARGIN
_SET_MCENTER	Logical	SET MESSAGE
_SET_MEMOEDITWARN	Numeric(Logical)	(see below)
_SET_MESSAGE	Numeric	SET MESSAGE
_SET_MESSAGE_GUI	Logical	SET MESSAGE (see below)
_SET_MULTILOCKS	Logical	SET MULTILOCKS
_SET_NFS_FORCE	Logical	SET NFS
_SET_NOINPUTCHAR	Numeric	(see below)
_SET_OPEN_ERROR	Logical	SET OPENERROR
_SET_PATH	Character	SET PATH
_SET_PIXEL	Logical	SET PIXEL
_SET_PRINTASCII	Logical	(see below)
_SET_PRINTCOLOR	Character	for SET GUIPRINT ON
_SET_PRINTER	Logical	SET PRINTER
_SET_PRINTFILE	Character	SET PRINTER TO
_SET_READAHEAD	Logical	(see below)
_SET_ROWADAPT	Logical	SET ROWADAPT
_SET_ROWALIGN	Logical	SET ROWALIGN
_SET_SCOREBOARD	Logical	SET SCOREBOARD
_SET_SCRCOMPRESS	Logical	SET SCRCOMPRESS
_SET_TERM_DATA	Character	(see below)
_SET_SOFTSEEK	Logical	SET SOFTSEEK
_SET_SOURCEASCII	Logical	SET SOURCE ASCII
_SET_UNIQUE	Logical	SET UNIQUE
_SET_UNIQUE_IGNORE	Logical	(see below)
_SET_VMEMOBIN	Logical	(see below)
_SET_WAIT_ECHO	Logical	(see WAIT command)
_SET_WAIT_IGNFUN	Logical	(see WAIT command)
_SET_WARN_IGNORE	Logical	(see below)
_SET_WRAP	Logical	SET WRAP
_SET_ZEROBYTEOUT	Character	SET ZEROBYTEOUT
_SET_COUNT	Numeric	count of _SET_...

Additional settings are available using the FS_SET() and SETKEY() function. Several global constants can be modified via assignment to _aGlobSetting[...], see source in <FlagShip_dir>/system/initio.prg

Description for settings available by Set() only:

`Set(_SET_DEBUG_PROFILE, [iType])` `-> iType`
Profiles time spent in each UDF to stderr device, you may redirect it to a file, e.g. 'a.out 2>file.name'

<iType> = 0: disable, default
1: protocol UDF entry, short form with param count

2: protocol UDF entry, long form with param values and callstack
4: protocol UDF exit
5: protocol UDF entry + exit, short form
6: protocol UDF entry + exit, long form

Set(_SET_INSERT, [logVar]) -> logVar
Get/set the insert mode for GET/READ and MemoEdit(). Equivalent to ReadInsert() function, refer to detailed description there. See also Set(_SET_INSERT_INKEY) for ability to change the insert mode by other key press.

Set(_SET_INSERT_INKEY, [logVar]) -> logVar
Get/set ability to change the insert mode by key press outside of GET/READ and MemoEdit(). The functionality of Set(_SET_INSERT) and ReadInsert() is not affected herewith. The default value is .F. to remain backward compatible. The current state of Insert switch can be determined by Set(_SET_INSERT) or InkeyStatus (STATUS_INSERT).

Set(_SET_LASTKEYBUFFSIZE, [intVar]) -> intVar
Get/set the size of lastkey() buffer, default = 10

Set(_SET_LOCK_MODE, [numVar]) -> nMode
Get/set used locking scheme/mode. The default mode is 1.

0 = use only "old" locking of FS4.4x to FS5.x
1 = compatibility mode: use "new" locking mode 2 when possible but 0 if the database is already in use by "old" applications. You may check currently used mode by DbObject():Info(DBI_LOCK_MODE)
2 = use new locking scheme only. Preferred for MS-Windows and Unix or Linux and only applicable for large files > 2Gb. An "old" application will then deny USE or DbUseArea() for database in use by others in this mode.

Set(_SET_LOCK_RETRY_SEC, [numVar]) -> nSeconds
Get/set max time frame in seconds for re-trying internal locks (0..600 seconds, default = 3.0)

Set(_SET_MEMOEDITWARN, [nOn|lOn]) -> nOn
Display MemoEdit() warning on ESC in non read-only mode:

nOn = 2 : always display warning and prompt (default)
1 : display warning and prompt only if text was changed
0 : don't display warning at all
lOn = .T. same as nOn = 2 (display warning and prompt)
= .F. same as nOn = 0 (don't display warnings)

Set(_SET_MESSAGE_GUI, [logVar]) -> logVar
For SET MESSAGE TO in GUI mode: invoke SET(_SET_MESSAGE_GUI, .T.) before MENU TO… to display @..PROMPT messages in the row specified by SET MESSAGE instead in the status bar line. Default <logVar> is .F.

Set(_SET_NOINPUTCHAR, [intVar]) -> intVar
Inkey() key value returned when SET INPUT is OFF, default = 27

Set(_SET_PRINTASCII, [lOn]) -> lRet
Set/get automatic ISO->ASCII translation via Ansi2oem() for printer output. Apply for Terminal and GUI i/o mode. Set automatically .T. by SET SOURCE ISO in GUI mode, default = .F. = off

Set(_SET_READAHEAD, [lOn]) -> lRet
Enable/disable read ahead into keyboard buffer, default is ON (.T.)

Set(_SET_TERM_DATA) -> cRet
Returns current terminal data and translation tables used in terminal i/o mode. An example of the (here splitted) output is:

```
"TerminalOutMapping=/usr/share/terminfo/FSchrmap.def" + ;
"[fslinxterm] (38 chars redefined) " + ;
"KeyboardMapping=/usr/share/terminfo/FSkeymap.def" + ;
"[fslinxterm] (128 keys redefined)"
```

Additional information is available via FS_SET()

Set(_SET_UNIQUE_IGNORE, [lOn]) -> lOn
Enable/disable unique index skipping on index created with UNIQUE flag. Advantageous e.g. for doublet checking. The default is .F., i.e. the Unique flag is considered if indexed so. With .T., all records are visible, same as on index created without the Unique flag.

Set _SET_VMEMOBIN, [lOn]) -> lOn
Enable/disable binary storage in VC* memo field (e.g. to store image data), <lOn> = .T. avoids translation to ASCII / ISO for the field. The default is .F., which means, the data are translated if required. The translation for VB* field is automatically disabled, similar to `Set(_SET_VMEMOBIN, .T.)`.

Set(_SET_WARN_IGNORE, [lOn]) -> lOn
Enable/disable additional warning when pressing "Ignore" on RTE errors. Default is enabled warning, i.e. lOn == .T.

Example 1:

```
? SET (_SET_CONSOLE)                    // .T.
? SET (_SET_PRINTER)                    // .F.
? SET (_SET_DECIMALS)                   // 2
SET (_SET_FIXED, .T.)                   // = SET FIXED ON
SET (_SET_ALTFILE, "xyz", .T.)          // = SET ALTERNATE
                                        //   TO xyz ADDITIVE
? SET (_SET_ALTFILE)                    // xyz
```

Example 2:

Check correct setting or restore old

```
iOldDeci := SET(_SET_DECIMALS, 4)       // iOldDeci = before set
 iNewDeci := SET(_SET_DECIMALS)          // returns new setting
 ? "Decimals set from", ltrim(iOldDeci), "to", ltrim(iNewDeci)
 if iNewDeci != 4
    ? "failure on new decimals setting"
 endif
 SET(_SET_DECIMALS, iOldDeci)            // reset old
```

Example 3:

General purpose function to save and restore all settings:

```
LOCAL actset := {}
#ifdef FlagShip
# include "set.fh"
   FS_SET ("lower, .T.)                 // additional settings
#else
# include "Set.CH"
#endif

SET ....
actset := saverestset ()                // save current settings
SET ...
actset[_SET_DEVICE] := NIL              // leave unchanged
saverestset (actset)                    // restore settings

FUNCTION saverestset (oldset)           // parameter is optional
*****************************           // to restore only
LOCAL ii, retarr[_SET_COUNT]
IF oldset != NIL .and. VALTYPE(oldset) == "A"
   FOR ii := 1 TO LEN(oldset)           // restore
      IF oldset[ii] != NIL
         retarr[ii] := SET (ii, oldset[ii])
      ENDIF
   NEXT
ELSE                                    // save current
   FOR ii := 1 TO _SET_COUNT
      retarr[ii] := SET (ii)
   NEXT
ENDIF
RETURN retarr
```

Classification:

programming

Compatibility:

Available in FS4 and C5 only. The numeric values of <expN1> are version-dependent and should never be used directly; the manifest constants are available in "set.fh" (or the C5 equivalences in "set.ch"). Note that the "set.ch" is only a subset of "set.fh", which should be preferably used in FlagShip.

Several flags are available only in FlagShip

Include:

The #include-able constants and manifests are available in the <FlagShip_dir>/include/set.fh file.

Related:

SET commands, SETKEY(), FS_SET()

SETANSI ()

Syntax:

```
retL = SetAnsi ([expL1])
```

Purpose:

Change the behavior how to read from and store data into database.

Options:

<**expL1**> is a logical value, FALSE value (the default) sets the ANSI <-> PC8 translation OFF, TRUE value enables the translation.

Returns:

<**retL**> is the current setting.

Description:

With SET ANSI ON or SetAnsi(.T.), a database access of character or memo translates the PC8/ASCII/OEM charset (used e.g. in DOS) via Oem2Ansi() into ANSI/ISO charset (used for display in GUI mode or in X11 terminal without a corresponding mapping). On replacing a char or memo fields in the database, the reverse Ansi2oem() translation is taken.

This means, special characters like a-umlaut, stored in the database as chr(132) in PC8/ASCII/OEM charset are translated during a read access to chr(228) in ANSI/ISO charset, to be displayed on the screen as a-umlaut in GUI environment or on X terminal. Reverse, with SET ANSI ON or SetAnsi(.T.), the a-umlaut chr(228) available in a variable or given in input, is stored in the dbf as chr(132) during the replace stage.

The SET SOURCE ASCII/PC8/OEM command sets also SET ASCII ON = SetAnsi(.F.) which may be used e.g. for Clipper data, which always use PC8/ASCII charset in the database, i.e. chr(132) for a-umlaut. The SET SOURCE ANSI/ISO command sets SET ANSI ON = SetAnsi(.T.) and handles a-umlaut as chr(228)

Example:

See example in CMD.READ and <FlagShip_dir>examples/setansi.prg for a complete example with description

Classification:

programming, database

Compatibility:

New in FS5

Related:

SET DBREAD, SET DBWRITE, SET KEYTRANSL|CHARSET, Ansi2oem(), Oem2Ansi()

SETBLINK ()

Syntax:

`retL = SETBLINK ([expL1])`

Purpose:

In MS-DOS: Toggles asterisk (*) interpretation in SETCOLOR() string between blinking and background intensity. No effect in FlagShip.

Options:

<**expL1**> is a logical value, TRUE value (the default) sets the blinking, FALSE sets background intensity.

Returns:

<**retL**> is the current setting.

Description:

Since the behavior of blinking is dependent on the UNIX terminal and on definitions in the "terminfo" file, the SETBLINK() function is implemented in FlagShip for compatibility purposes only.

To change the environment variable TERM and reinitialize the screen, use FS_SET("setenvir") function.

Classification:

programming

Source:

This user modifiable function is available in <FlagShip_dir>/ system/scrnmode.prg

Related:

SET COLOR, FS_SET("setenv"), FS_SET("term")

SETCANCEL()

Syntax:

`retL = SETCANCEL ([expL])`

Purpose:

Toggles program termination with Ctrl-K on or off.

Options:

<**expL**> toggles the termination capability on (if TRUE), or off (FALSE). The default setting is TRUE.

Returns:

<**retL**> is a logical value, representing the last state before the current toggling.

Description:

SETCANCEL() is a keyboard function that toggles the state of the termination key (^K or other key set by FS_SET("break")), and reports the current state of SETCANCEL().

Using SETCANCEL(.F.) will suppress the user's ability to terminate a program using the break key (e.g. being in REPLACE or INDEX), until resetting it by SETCANCEL (.T.).

Example:

```
PUBLIC FlagShip, Clipper
termkey = if (FlagShip, "Ctrl-K", "Ctrl-C")
yes_no = IF(SETCANCEL(), "", "not ")
?  "In case of entering an endless loop", ;
   "you can " + yes_no, ;
   "terminate the program with " + termkey
```

Classification:

programming

Compatibility:

Note the different termination keys in FlagShip (Ctrl-K) and Clipper (Ctrl-C). In FlagShip, the break key may be redefined using FS_SET("break").

Related:

SET ESCAPE, SET KEY, ALTD(), FS_SET()

SETCOLOR ()

Syntax:

retCA = SETCOLOR ([expCAON1], [expL2])

Purpose:

Reports the current color setting and optionally sets a new one.

Options:

<**expC1**> is a character string containing the new color setting. It contains the standard, enhanced, border, background and unselected color settings in the same format as SET COLOR TO. If the argument is not specified, only the current color setting will be returned. Instead of color symbol, you also may specify RGB string in #RRGGBB notation, see details in SET COLOR. If the <expC1> is "", SetColor() resets default colors from current `_aGlobSetting[GSET_G_AC_DEFCOLOR]` or `_aGlobSetting[GSET_T_C_DEFCOLOR]` (for GUI or Terminal i/o resp.), same as SET COLOR TO w/o argument.

<**expA1**> is an alternative entry via 2 or 3-dimensional array of RGB triplets. See SET COLOR for more details.

<**expO1**> is an alternative entry via Color object

<**expN1**> is an alternative entry using numeric value for the standard color. The valid <expN1> range is 0..255, and the color pair is coded as foreground-color + (background-color * 16), e.g. "R+/GR" = "12/6" = 12 + 6*16 = 108, or "W/N" = "7/0" = 7 + 0*16 = 7. Note, that the numeric value (0..15) is also accepted within the color string, e.g. "10/1" is equivalent to "G+/B" or to numeric 26. Due of the numeric nature, the numeric <expN1> value support only the standard color pair. See SET COLOR for the numeric values. If the <expN1> is -1, SetColor() resets default colors same as SetColor("").

<**expL2**> is a flag for the returned type. If .F. or NIL, SetColor() return translation of closest RGB to color char, .T. return array of RGB triplets

Returns:

<**retC**> is a character string, containing the current color setting. If a RGB string or RGB array was specified, either the matching symbol is returned if an exact match was found, otherwise the partial color is returned as RGB string in #RRGGBB notification.

<**retA**> is an array of RGB triplets, returned if <expL2> is .T.

Description:

SETCOLOR() supports the same format of color setting as SET COLOR TO, and can be used to save, and later restore a color setting.

For more information, refer to the SET COLOR command.

Tuning:

You may set your own default color by assigning e.g.

```
  _aGlobSetting[GSET_G_AC_DEFCOLOR] := "N/?,N/W+,N/W+,N/W+,N/W,N/W+"
  _aGlobSetting[GSET_T_C_DEFCOLOR ] := "W/N,N/W,W/N,W/N,N/W"
```

for GUI and/or Terminal i/o respectivelly and invoking SET COLOR TO or SetColor(-1) to change color defauls.

Example:

```
Save and restore the color settings:

PROCEDURE warning (message)
LOCAL actcolor := SETCOLOR ()
SET COLOR TO +R/B,RB+/N                 && SETCOLOR("+R/B,RB+/N")
?? CHR(7)
@ 0,0 SAY "Warning: " + message + " ..any key"
INKEY (0)
SETCOLOR(actcolor)
@ 0,0
RETURN
```

Classification:

programming, usable only with enabled screen oriented output (see sect. SYS.2.7).

Compatibility:

The additional types of 1st parameter as well as the 2nd parameter are new in FS5

Related:

SET COLOR TO, SET INTENSITY, SETSTANDARD, SETENHANCED, SETUNSELECTED

SETCOLORBACKGROUND ()

Syntax:

```
retA = SETCOLORBAckground ([expCAO1], [expL4])
retA = SETCOLORBAckground ([expN1], [expN2], [expN3],
                           [expL4])
```

Purpose:

Sets and/or reports the current GUI background color.

Options:

<**expC1**> is a character string containing the new color setting of the GUI window background and for default background of ?, ??, @..SAY etc. commands. If the string includes slash "/" (see SET COLOR TO), only the background color is considered. Instead of color symbol, you also may specify RGB string in #RRGGBB notation, see details in SET COLOR.

<**expA1**> is an alternative entry via one, two or three-dimensional array of RGB triplets. See SET COLOR for more details. If one-dim array is used, the form is equivalent to the return value.

<**expO1**> is an alternative entry via Color object

<**expN1**> is an alternative entry using numeric value for the background color. If <expN2> and <expN3> is used, this value specify the amount of red fraction in the color as a value between 0 and 255. If <expN2> and <expN3> are not used, see details in SETCOLOR() or SET COLOR TO.

<**expN2**> is the amount of green fraction in the color as a value between 0 and 255.

<**expN3**> is the amount of blue fraction in the color as a value between 0 and 255.

<**expL4**> is optional logical value. If `.T.`, the SetColorBackground() is considered also for default background in COLOR clause with SET GUICOLOR ON. If this parameter is set `.F.`, SetColorBackground() will not be considered for default background in ?, ??, @..SAY etc = behaves backward compatible to VFS5..VFS7.

Returns:

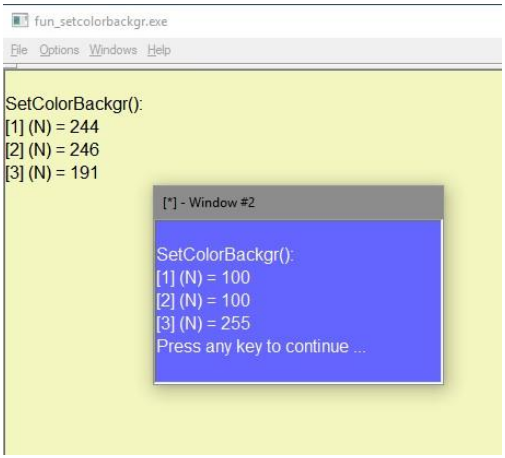
<**retA**> is an array of RGB triplets, where retA[1] is the amount of red fraction in the color as a value between 0 and 255, retA[2] the amount of green fraction and retA[3] the amount of blue fraction determined at the time of entering the function. On error (e.g. in Terminal i/o mode), an empty array is returned.

Description:

SETCOLORBA() allows you to set the background color of the whole user area (widget, control) by a subsequent CLS or CLEAR command, e.g. when you wish to use other than the standard gray background. This function is considered in GUI mode only and is ignored in other i/o modes, where an empty array is returned. The passed color <exp1> is also used for default (i.e. not specified) background in GUICOLOR clause of ?, ??, @..SAY - see also <expL4> for the COLOR clause.

Example:

```
aOld := SetColorBackgr("N/W+")        // white background
aNew := SetColorBackgr()
CLS
qout("Old:") ; aeval(aOld, {|x| qqout(x)})   // 220 220 220
qout("New:") ; aeval(aNew, {|x| qqout(x)})   // 255 255 255
wait

SetColorBackgr(100, 100, 255)         // light blue background
CLS
qout("Changed 100,100,255:")
aeval(SetColorBackgr(), {|x| qqout(x)})
SET COLOR TO "W+/#6464FF"
SET GUICOLOR ON
qout("Changed:") ; aeval(SetColorBackgr(), {|x| qqout(x)})
wait

// apply also for windowing

SET COLOR TO "N/#F4F6BF"              // light yellow
CLS
? "SetColorBackgr():" ; _DisplArrStd(SetColorBackgr())
SET COLOR TO "W+/#6464FF"             // light blue
iWin1 := Wopen(5,10,12,30)
CLS
? "SetColorBackgr():" ; _DisplArrStd(SetColorBackgr())
wait
Waclose()

SET GUICOLOR OFF
SetColorBackground(aOld)              // orig. gray background
cls
```



***Classification*:** programming, usable only in GUI mode.
Compatibility: New in FS5, parameter 4 is new in FS8
Related: SET COLOR TO, SETCOLOR(), SET GUICOLOR, CLS, oApplic:ColorBackground

SETCOL2GET ()

Syntax:

`retC = SETCOL2GET ([expC])`

Purpose:

Transforms the specified or the current color setting into the get:COLORSPEC or the @...GET...COLOR notation.

Options:

<**expC**> is a character string containing the color attributes given in the standard SETCOLOR() or SET COLOR TO notation. If the argument is not specified, the current color setting will be used.

Returns:

<**retC**> is a character string, containing the color attributes in the get:COLORSPEC notation: <inactive_field>, <active_field>.

Description:

The color specification of the get:COLORSPEC instance and the COLOR clause of the @...GET... command has a different format from the standard SETCOLOR() or SET COLOR TO... notation:

SET COLOR TO	=	<standard>,<enhanced>,,,<unselected>
get:COLORSPEC	=	<unselected>,<enhanced> [,<ignored>...]

SETCOL2GET() converts the SETCOLOR() attributes into the COLORSPEC notation. If the <unselected> attribute is not specified in <expC>, the <enhanced> color pair will be used twice. The SETCOL2GET() function does not change the current SETCOLOR().

Example:

```
SETCOLOR ("W/B,N/W,,,W+/BG")
? SETCOL2GET()                          // +W/BG,N/W
? SETCOLOR()                            // W/B,N/W,,,W+/BG
? SETCOL2GET ("W/B,N/W,,,W+/BG")        // +W/BG,N/W
? SETCOL2GET ("R/W,W+/R,,,GR+/B")       // +GR/B,+W/R
? SETCOL2GET ("R/W,W+/R")               // +W/R,+W/R
```

Classification:

programming

Compatibility:

Available in FlagShip only.

Related:

SET COLOR TO, SETCOLOR(), SETSTANDARD, SETENHANCED, SETUNSELECTED, COLORSELECT()

SETCURSOR ()

Syntax:

`retN = SETCURSOR ([expNL1])`

Purpose:

Activates/disables cursor visibility. In MS-DOS: sets the cursor shape.

Options:

<**expN1**> is a number indicating the shape of the cursor, e.g. none, underline, half block etc. Zero specifies SET CURSOR OFF, other values SET CURSOR ON.

<**expL1**> is an alternative syntax, where .F. disables cursor (same as nMode == 0), and .T. enables it (same as nMode >= 1)

Returns:

<**retN**> is the current cursor (shape) set.

Description:

The cursor setting is dependent on the Unix terminal. Therefore, only on and off are supported by FlagShip.

SETCURSOR(0) is the same as SET CURSOR OFF, and any positive integer value of <expN1> is the same as SET CURSOR ON.

Classification:

screen oriented output

Compatibility:

Most UNIX versions (curses libraries) cannot disable the cursor, so the cursor will be visible on the rightmost bottom position of the screen.

The alternative logical parameter is new in FS5

Related:

SET CURSOR, SET CONSOLE, SetGuiCursor(), SET GUICURSOR

SETENV()

Syntax:

retC = SETENV (expC1, [expC2], [expL3])

Purpose:

Determines or sets an Unix or Windows environment variable for the current process, and optionally, reinitializes the screen. Note that SetEnv() is a shortcut for FS_SET("setenv", expC1, [expC2], [expL3]) available for your convenience.

Arguments:

<**expC1**> is the name of the Unix/Windows shell environment variable and is case-sensitive.

Options:

<**expC2**> is the new value, to be assigned to the environment variable <expC1>. If the argument is not given or NIL is specified, the contents of the environment variable remain unchanged. Otherwise, if the variable <expC1> does not exist, a new one is created.

<**expL3**> specifies if the curses (and the screen, character mapping) are to be reinitialized. Any value other than TRUE leaves the initialization unchanged.

Returns:

<**retC**> is the current contents of the specified Unix or Windows environment variable after executing the function.

Description:

If only the argument <expC1> is specified, the function determines the current environment setting, same as the GETENV() function.

Otherwise, this SETENV() function sets a new value for a specific Unix/Windows environment variable creating this variable if it does not exist. The new setting is active during the execution of the current process (application) and its recently activated child processes (e.g. using the RUN command).

Note that you cannot set/change parent's shell environment from the running application (sub-process), see additional **tech info** in the FS_SET("setenv") description.

Example:

```
old := GetEnv("TESTENV")
? "Environment variable TESTENV " + ;
  if(empty(old), "is not available", "= '" + old + "'")

new := SetEnv("TESTENV", "Hello World")
? "Setting environment variable TESTENV, result:", new
? "Test: '" + GetEnv("TESTENV") + "'"
wait
```

Classification:

programming

Compatibility:

Shortcut to FS_SET("setenv", ...). Available in VFS7 and newer.

Related:

GetEnv(), FS_SET("setenv")

SETEVENT ()

Syntax:

```
retN = SetEvent ([expN1], [expN2], [expN3])
```

Purpose:

Change the behavior of GUI in handling events. Ignored in Terminal and Basic i/o.

Options:

<**expN1**> is the interval in seconds (and fractions of) specifying the time difference between processing events. The common value if between 0.1 to 3 seconds. Accepted are values > 0.002 and < 3600, i.e. more than 2 milliseconds but less than 60 minutes. The default setting is 0.2, i.e. the pending events are processed every 200 milliseconds. Equivalent to Set(_SET_EVENT_INTERVAL).

<**expN2**> is the duration in seconds (and fractions of) specifying the maximal time slice of processing pending events in one specific call. If set <= 0, the default time slice is triple of <expN1>. Default value is 0. Equivalent to Set(_SET_EVENT_DURATION).

<**expN3**> is a numeric value specifying the mode how to process GUI events:

1 = process every pending event, may slow-down the execution speed
2 = check for and process events in intervals of <expN1> seconds.

Default value is 2. Equivalent to Set(_SET_EVENT_MODE).

Returns:

<**retN**> is the current setting of the interval at the time of entering this function.

Description:

In GUI, the application communicates with the X11 server or with the MS-Windows via "events". The system specific event queue must be checked and processed frequently to pass key pressed and mouse movements to the application, and to display the screen output created by the application. For that purposes, FlagShip generates in any .prg file a check for event queue.

But testing the system event queue is time expensive; calling it too often may slow down the processing speed significantly. Therefore, FlagShip does not test the event queue not before the <expN1> interval is expired. The evens get not lost: when the event queue contain more events than the time slice <expN2> can process, the pending events are hold in the queue and are processed at the next .prg statement and/or the event queue call.

Modifying the <expN1> time value may some cases increase the performance of your application significantly.

The <expN2> value should be in relationship to <expN1> and to the amount of expected events to be processed. Setting this value too small and/or smaller than

<expN1> may cause very slow or an delayed output. The default value is twice of <expN1> which ensures all usual events are processed timely.

For example in a large loop without any in/output, you may set <expN1> to a high value, e.g. to 0.5 to 2 seconds. On the other hand, if a part of the application is very i/o intensive, you may set lower <expN1> value (e.g. 0.01) and/or higher <expN2> value. But don't forget to reset the values thereafter to allow the default event queue processing. Otherwise the application will behave to be sluggish.

The time slice of event processing can also be set by compiler switches -ev=<valueMs>, -ne (disable event checking) and -d (debugger). When <expN2> is set by Set(_SET_EVENT_DURATION) or by SetEvent(), it has preference over the compiler switch.

Example:

```
? SetEvent()                  // 0.2
local iEvent := SetEvent(1)   // check events every second
for i := 1 to 1000000
   ...init variables etc.
next
SetEvent(iEvent)              // reset to default
```

Classification:

programming, database

Compatibility:

New in FS5

Related:

Set(), compiler switches -ev=, -ne, -d

SETFONT ()

Syntax:

retO = SetFont ([expO1])

Syntax 2:

retO = SetFont ([expC1], [expN2], [expC3])

Purpose:

Retrieves current and/or sets new default font name and/or size and/or attribute, used for all consecutive console output like ?, ??, @..SAY, @..GET, QOut(), QQOUT() etc. This is very similar to SET FONT command, or oApplic:Font. Apply also for printer output with SET GUIPRINT ON. Applicable for GUI mode, ignored otherwise.

Options:

<**expO1**> is the new Font object to be assigned as default.

<**expC1**> is the new font family. The available family depends on the installed fonts. Usually, at least "Helvetica" or "Arial", "Times" and "Courier" fonts are available. If not given or is empty, the current font family is used.

<**expN2**> is the new font size in points. The common size is 10 (points), larger size is 12, smaller is 8 points. If not given or is zero, the current font size is used.

<**expC3**> is the font attribute to be set. Any combination of letters "N" (normal), "B" (bold), "I" (italic), "U" (underline), "S" (striked thru) is supported. If not given or is empty, the current font attribute is used.

If no arguments are given, only the current font is returned.

Returns:

<**retO**> is a copy of the current font, same as oApplic:Font In other than GUI i/o mode, NIL is returned.

Description:

SetFont() behaves similarly to SET FONT command, but accepts also font object. See the SET FONT command for detailed description.

Note that changing the font will result in recalculating of the current row, column and line height, they all depends on used font. For using different font attributes within current text, use the FONT clause of ?, ?? or @...SAY commands instead of changing default font by SetFont(), see second example below.

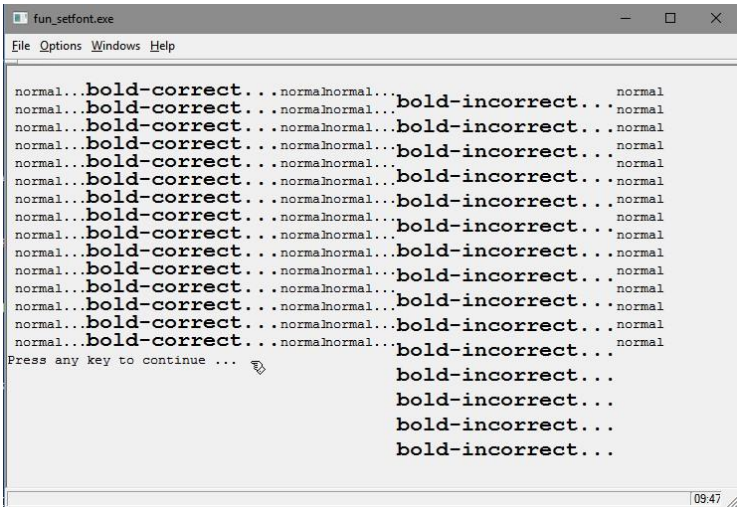
Example 1:

```
local oFont9bold := Font{"Courier", 9, "B"}  // temp inline font
local oSaveFont
? "Output with default font =", oApplic:Font:FontFamily, "/" , ;
  ltrim(oApplic:Font:Size), "/", oApplic:Font:AttribChar()
oSaveFont := SetFont(oFont9bold)         // change default font
? "Output with Courier 9, bold"
SetFont(oSaveFont)                       // restore default font
? "Output with default font"
```

```
SET FONT BASELINE ON                        // align on base line
?? " - continuing with Courier 9, bold" FONT (oFont9bold)
SET FONT BASELINE OFF                       // restore default
```

Example 2:

```
// correct handling of font attrib change within text
local oFont10norm := Font{"Courier", 10, "N"}
local oFont16bold := Font{"Courier", 16, "B"}
SetFont(oFont10norm)                        // change default font
SET FONT BASELINE ON                        // align on base line
for ii := 1 to 15
   @ ii,1 say "normal..."
   @ ii,col() say "bold-correct..." FONT (oFont16bold)
   @ ii,col() say "normal"
next

// incorrect change of font attrib, causes coord shift
SetFont(oFont10norm)                        // set default font
for ii := 1 to 15
   @ ii,40 say "normal..."                  // by default coordinates
   SetFont(oFont16bold)                     // set new default font
   @ ii,col() say "bold-incorrect..."       // by new coordinates
   SetFont(oFont10norm)                     // reset default font
   @ ii,col() say "normal"                  // by default coordinates
next
SET FONT BASELINE OFF                       // restore default
wait
```



Classification:

screen oriented output in GUI mode and PrintGui() printer output

Compatibility:

Available in FlagShip only

Related:

SET FONT, SET FONT BASELINE, oApplic:Font, OBJ.Font class

SETGUICURSOR ()

Syntax 1:

```
retA = SETGUICURSOR ([expN1], [expN2], [expN3],
            [expL4], [expL5], [expL6])
```

Syntax 2:

```
retA = SETGUICURSOR (expA1)
```

Syntax 3:

```
retA = SETGUICURSOR (expL1)
```

Syntax 4:

```
retA = SETGUICURSOR ( )
```

Purpose:

In GUI mode, sets the text cursor visibility and/or displays it at given position.

Syntax 1 sets the text cursor at specified position,
Syntax 2 sets/restores the text cursor at specified position,
Syntax 3 enables or disables text cursor visibility,
Syntax 4 reports the current status only.

Arguments:

<**expL1**> in syntax 3 is a logical value. True .T. enables text cursor visibility at previously set position, .F. disables the text cursor.

<**expA1**> in syntax 2 is an array of the same structure as the return value <retA> and may be used to restore previously determined (and then changed) cursor setting.

<**expN1**> in syntax 1 is numeric value specifying the row (or y) position where the text cursor should display. Depending on <expL5> or SET PIXEL, the value is in rows or in pixels. If <expN1> is not given or is NIL, the previously set value is used.

<**expN2**> is numeric value specifying the column (or x) position where the text cursor should display. Depending on <expL5> or SET PIXEL, the value is in columns or in pixels. If <expN2> is not given or is NIL, the previously set value is used.

<**expN3**> is a numeric value specifying the cursor shape. If not given or is NIL, the previously set value is used. The CURSOR_* constants are defined in mouse.fh

mouse.fh constant	value	Description
CURSOR_ARROW	-1	standard arrow cursor
CURSOR_UPARROW	-12	upwards arrow
CURSOR_CROSS	-8	crosshair
CURSOR_WAIT	-9	hourglass
CURSOR_IBEAM	-11	i-beam (I)
CURSOR_SIZE_VER	-2	vertical resize
CURSOR_SIZE_HOR	-3	horizontal resize
CURSOR_SIZE_RDIAG	-5	diagonal resize (/)
CURSOR_SIZE_LDIAG	-4	diagonal resize (\)

CURSOR_SIZE_ALL	-13	all directions resize
CURSOR_INVISIBLE	-17	blank/invisible cursor
CURSOR_SPLITVER	-14	vertical splitting
CURSOR_SPLITHOR	-3	horizontal splitting
CURSOR_HAND	-6	a pointing hand
CURSOR_FORBIDDEN	-16	forbidden action cursor
CURSOR_UNDERSCORE	-21	underscore
CURSOR_BOX	-22	box in size of one largest character
CURSOR_DEFAULT_TEXT	-21	default = CURSOR_UNDERSCORE

<**expL4**> is a logical value. True .T. forces the text cursor to blink, false .F. displays solid text cursor. If not given or is NIL, the previously set value is used.

<**expL5**> is a pixel specification. If .T., the <expN1> and <expN2> are given in pixel, if .F. in row/col, otherwise the current SET PIXEL status is used.

<**expL6**> is a logical value specifying character alignment. If .T., the cursor shape is placed so that it fits to a character, whereby the currently used font is considered. If .F., no alignment is used. If not given or is NIL, and either <expN1> or <expN2> was specified, the alignment is OFF (.F.) when pixel coordinates are used, and ON otherwise. If not given or is NIL and also <expN1> and <expN2> are not specified, the previous setting is used.

Returns:

<**retA**> is an array of seven elements, reporting the status at the time of entering this function. You may use the first six elements as parameters for SetGuiCursor() to restore previous settings.

retA[1]	(N)	row in either coordinates or pixel, accord.to retA[5]
retA[2]	(N)	col in either coordinates or pixel, accord.to retA[5]
retA[3]	(N)	cursor shape type, same as <expN3>
retA[4]	(L)	blinking (.T.) or solid (.F.) cursor shape
retA[5]	(L)	if .T., the retA[1] and retA[2] values are in pixel
retA[6]	(L)	alignment, same as <expN6>
retA[7]	(L)	the text cursor is visible (.T.) or is disabled (.F.)

Description:

With SetGuiCursor(), you may display many different cursor shapes anywhere on the user area of your application screen, except on additional widgets (controls) created by e.g. @..GET, MemoEdit(), Tbrowse(), MessageBox() and so on.

SetGuiCursor() does not change or influence the current Row() and Col() position. When you wish to display specific cursor shape during the ?, ??, Qout(), Qqout() and @..SAY output at the current display position, you may use SET GUICURSOR ON and SET GUICURSOR TO <shape> commands, or the corresponding SET(_SET_GUICURSOR) and SET(_SET_GUICURSORTYPE) functions.

When SET GUICURSOR is ON, the default (or the SET GUICURSOR TO) cursor shape overrides the shape set by SetGuiCursor() at any output via ?, ??, Qout(), @..SAY etc. but it may be restored by SetGuiCursor(.T.) thereafter.

The WAIT command use own shape to signal user's entry. If the SetGuiCursor() display is enabled, your cursor shape will be restored automatically at the return from WAIT.

To set the shape of mouse cursor, use MsetCursor().

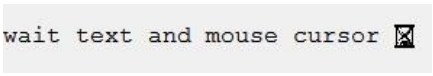
This function is designed mainly for GUI mode. In other i/o modes, only the standard values are returned but no action is taken. To set and display cursor at specific position in text mode, use SetPos() and control the visibility by SetCursor().

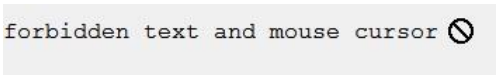
Example 1:

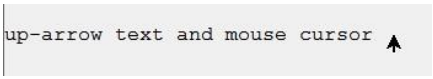
```
#include "mouse.fh"
? "hello world"
SetGuiCursor(10,25, CURSOR_WAIT, .T.)   // wait-cursor
?? "..text continued, any key..."
inkey(5)

SetGuiCursor(.F.)                       // disable it
? "cursor shape disabled now, any key..."
inkey(5)

SetGuiCursor(.T.)                       // enable it
? "cursor shape enabled now..."
inkey(5)

wait NOECHO TIMEOUT 5 ;
     "The WAIT command use own shape (any key...)"
? "but the cursor is restored thereafter if active"
inkey(5)
```

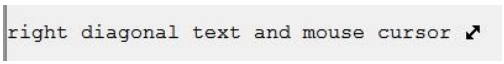
Example 2:

complete example is available in <FlagShip_dir>/examples/guicursor.prg, e.g.:









Classification:

screen oriented output in GUI mode

Compatibility:

Available in FlagShip only

Related:

SET GUICURSOR, SET CURSOR, SetCursor(), MsetCursor()

SETKEY ()

Syntax:

`retB = SETKEY (expN1, [expB2])`

Purpose:

Assigns an action block to a key.

Arguments:

<**expN1**> is the ASCII value of the key, including negative numbers for function keys (except 0), see INKEY() values. <expN1> can be NIL when also <expB2> is NIL, see below.

Options:

<**expB2**> specifies a code block that will be automatically executed whenever the specified key is pressed during a wait status.

If <expN1> is numeric and <expB2> is not given, the currently assigned code block for <expN1> key is returned.

If <expN1> is numeric and <expB2> is NIL, the corresponding key action of <expN1> is cleared, same as SET KEY <expN1> TO.

If both <expN1> and <expB2> are given and NIL, all SET KEY actions are cleared, same as SET KEY CLEAR

If both <expN1> and <expB2> are omitted (= SETKEY() w/o parameters), or <expN1> is neither numeric nor NIL, no action is taken and <retB> returns NIL.

Returns:

<**retB**> is the action block currently associated with the specified key, or NIL if the specified key is not currently associated with a block.

Description:

SETKEY() is similar function to SET KEY command. It sets or retrieves the automatic action associated with a particular key during a wait status. A wait status is any mode that extracts keys from the keyboard except for INKEY(), but including ACHOICE(), DBEDIT(), INKEYTRAP(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT. Any number of keys may be assigned at any one time.

SETKEY() is usually used to assign a UDP or UDF to a specific key. Upon starting up, the F1 key is automatically assigned to a UDF or UDP named HELP, if such exist.

When an assigned key is pressed during a wait state, the EVAL() function evaluates the associated action block, passing the parameters PROCNAME(), PROCLINE(), and READVAR().

For more information, refer to the SET KEY command.

Example:

Issue the "SET KEY" action also on INKEY():

```
LOCAL key, block
#include "inkey.fh"
SETKEY (K_F4, {|pnam, plin, rvar| MYUDF (pnam,plin,rvar)} )
key = INKEY(0)
block := SETKEY (K_F4)
IF block != NIL
   EVAL (block, PROCNAME(), PROCLINE(), READVAR())
ELSE
   ? "key", IF (key < 0, FKLABEL(key), CHR(key)), "pressed"
ENDIF
```

Classification:

programming

Compatibility:

Available in FS and C5 only. Refer to the current terminfo for the ability of particular function key. See also section SYS and the command SET KEY for more information.

Include:

Predefined function key constants are available in "inkey.fh"

Related:

SET KEY, SET FUNCTION, ON KEY, SetKeySave(), SetKeyRest(), PmKey(), Eval(), Inkey(), InkeyTrap(), GetKey(), IsFunction()

SETKEYREST ()

Syntax:

`retL = SETKEYREST ( )`

Purpose:

Restore previously saved SetKey's and OnKey's by SetKeySave() or by PUSH KEY.

Returns:

<**retL**> is .T. if the restore was successful, or is .F. if there was no previously saved SET KEYs by SetKeySave() or by PUSH KEY.

Description:

SetKeyRest() is similar to POP KEY ALL command. It restores the last SET KEY, ON KEY and ON KEY LABEL structure previously saved by PUSH KEY or by SetKeySave().

Example:

```
see SetKeySave()
```

Classification:

programming

Compatibility:

Available in VFS only.

Related:

SetKeySave(), SetKey(), SET KEY, ON KEY, PUSH KEY, POP KEY, FS2:GetSetKeys(), FS2:GetOnKeys(), FS2:GetFunKeys()

SETKEYSAVE ()

Syntax:

retL = SETKEYSAVE ([expL1])

Purpose:

Save SetKey's and OnKey's to be restored later by SetKeyRest() or by POP KEY ALL.

Options:

<**expL1**> if specified .T., all current SET KEYs and ON KEYs redirections are cleared. Default is .F. which does not clear them.

Returns:

<**retL**> is .T. if the store was successful, or is .F. if there was no SET KEYs nor ON KEYs redirections to be saved.

Description:

SetKeySave() is similar to PUSH KEY [CLEAR] command. It saves the current SET KEY, ON KEY and ON KEY LABEL definitions on internal stack for later restoring by SetKeyRest(). If the optional <expL1> is TRUE, it deletes all key assignments of SET KEY, ON KEY and ON KEY LABEL.

Example:

```
 SET KEY K_F2 to myFun2              // set F2 redirection
 SET KEY K_CTRL_X to myFunX          // set ^X redirection
 wait "press F2 or CTRL-X"           // both F2 and ^X are
redirected

 SetKeySave()                        // save current SET KEYs
 SET KEY K_F2 to                     // clear F2 redirection
 wait "press CTRL-X, F2 is disabled" // only ^X is redirected now
 SetKeyRest()                        // restore saved SET KEYs
 wait "press F2 or CTRL-X"           // both F2 and ^X are
redirected

 SetKeySave(.T.)                     // save & clear current SET
KEYs
 wait "both F2 and CTRL-X disabled"  // nothing is redirected now
 SetKeyRestore()                     // restore saved SET KEYs
 wait "press F2 or CTRL-X"           // both F2 and ^X are
redirected

 FUNCTION myFun2(p1,p2,p3)
 return alert("F2 pressed in " + procstack(1))
 FUNCTION myFunX(p1,p2,p3)
 return alert("^X pressed in " + procstack(1))
```

Classification: programming
Compatibility: available in VFS only
Related: SetKeyRest(), SetKey(), SET KEY, ON KEY, PUSH KEY, POP KEY, FS2:GetSetKeys(), FS2:GetOnKeys(), FS2:GetFunKeys()

SETMODE ()

Syntax:

```
retL = SETMODE (expN1, expN2)
```

Purpose:

Set the size of application window (GUI) or console window (Terminal i/o).

Arguments:

<**expN1**>, <**expN2**> is the number of rows and columns in the desired display mode.

Returns:

<**retL**> is TRUE if the mode change was successful. On Unix, the function always returns FALSE.

Description:

This function is equivalent to oApplic:Resize() or ConsoleSize() and apply in GUI mode and Terminal i/o mode of MS-Windows.

Not applicable for Linux in Terminal i/o mode, where the xterm settings (terminal size and coordinates) can be controlled by newfswin script. The terminal characteristic of curses is controlled in Unix by definitions in the "terminfo" file and the associated TERM environment variable, hence the SETMODE() function is implemented for compatibility purposes only. To change the environment variable TERM and reinitialize the screen, use the FS_SET("setenvir") function.

Classification:

programming, for compatibility purposes

Source:

The user modifiable SETMODE() function is available in *<FlagShip_dir>/system/scrnmode.prg*

Related:

FS_SET("setenv"), FS_SET("term"), MAXROW(), MAXCOL()

SETPOS ()

Syntax:

NIL = SETPOS (expN1, expN2, [expL3])

Purpose:

Moves the cursor to a new position.

Arguments:

<**expN1**> is the new screen row in the range 0 to MAXROW().

<**expN2**> is the new screen column in the range 0 to MAXCOL().

<**expL3**> is a pixel specification. If .T., the coordinates are assumed in pixel, if .F. the coordinates are row/col, otherwise the current SET PIXEL status is used. If <expL3> or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

Returns:

The function always returns NIL.

Description:

SETPOS() is an environment function that moves the cursor to a new position on the screen, as the `@ expN1, expN2 SAY ""` command, but may also be used in code blocks or expressions.

After the cursor is positioned, ROW() and COL() are updated accordingly. To control the visibility of the cursor, use the SETCURSOR() function or the SET CURSOR command.

The function is always executed on the screen, independent of the SET DEVICE status. To set the cursor or print head in dependence of the SET DEVICE status, use **DEVPOS()** instead, which is also used for the @..SAY command.

In GUI mode, any output is pixel oriented. For your convenience and to achieve cross compatibility to textual based applications, FlagShip supports also coordinates in common row/column values. It then internally re-calculates the given rows by using Row2pixel() and columns by using Col2pixel() function. The line and character spacing is affected by the currently used default font. For minimal porting effort, best to use fixed fonts (SET FONT "Courier", 12).

For additional hints how to manage proportional fonts, see further details in LNG.5.3, Col(), Row(), Col2pixel(), Row2pixel(), SET ROWALIGN, SET ROWADAPT, SET FONT.

Example:

```
bl := {|par| SETPOS (10,10), QQOUT (par)}
EVAL (bl, "Hello world")
SET DEVICE TO PRINT
SET PRINTER TO ("my.file")
SETPOS (11,10)
?? "printed to screen")
```

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND()

Compatibility:

Available in FS and C5 only. The 3rd parameter is new in FS5

Related:

DEVPOS(), DEVOUT(), @..SAY, ?, ??, COL(), ROW(), Col2Pixel(), Row2pixel(), SET FONT

SETPRC ()

Syntax:

NIL = SETPRC (expN1, expN2)

Purpose:

Sets the printer row and column to the specified values.

Arguments:

<**expN1**> is the new printer row.

<**expN2**> is the new printer column.

Returns:

The function always returns NIL.

Description:

Using SETPRC(), the printer row and column can be changed without moving the printer head. It can also be used to suppress page ejects or to correct the column position when control codes are sent to printer.

When a output is set to printer using SET PRINTER or SET DEVICE, the PROW() and PCOL() values are updated with the current print head position, regardless of the default FlagShip redirection of the print output to a spool file.

Sending a printer control code to the output, will desynchronize the visual printout and PCOL() - although control codes are not printed out, PCOL() is affected as if they were. In this case, correct the PCOL() value using SETPRC().

Example:

```
#define ITALIC_ON  CHR(27) + "I"
#define ITALIC_OFF CHR(27) + "E"
#define MAXLINE 65
USE address
SET DEVICE TO PRINTER
DO WHILE ! EOF()
   @ PROW(), 0 SAY ITALIC_ON + TRIM(name) + ITALIC_OFF
   SETPRC (PROW(), LEN(name) +1)        // make corrections
   @ PROW(), PCOL() SAY address
   SKIP
   IF PROW() > MAXLINE ; EJECT ; ENDIF
ENDDO
```

Classification:

programming

Related:

PCOL(), PROW(), COL(), ROW(), SET DEVICE, SET PRINTER, EJECT

SETVAREMPTY ()

Syntax:

`SetVarEmpty (@var)`

Purpose:

Set variable <var> to an empty value, considering the variable type

Arguments:

<**@var**> is a variable of any type, passed by reference. If you omit the @ prefix, it is automatically appended by the preprocessor. You may not use this function for database fields, nor for constants.

Description:

The function is comparable to an assignment

```
var := ""          (for character variable)
var := ctod("")    (for date variable)
var := 0           (for numeric variable)
var := .F.         (for logical variable)
var := {|| NIL}    (for code block variable)
var := array(0)    (for array variable)
```

or to initialization of typed variables.

Example:

```
var := { 1, 2, {3, 4}}
? valtype(var), len(var), empty(var)   // "A"  3  .F.
SetVarEmpty(@var)
? valtype(var), len(var), empty(var)   // "A"  0  .T.

var := {|x| qout("?") }
? valtype(var), empty(var)             // "B"  .F.
SetVarEmpty(@var)
? valtype(var), empty(var)             // "B"  .T.

var := 1.23
? valtype(var), empty(var)             // "N"  .F.
SetVarEmpty(@var)
? valtype(var), empty(var)             // "N"  .T.

var := "abcd"
? valtype(var), len(var), empty(var)   // "C"  4  .F.
SetVarEmpty(var)
? valtype(var), len(var), empty(var)   // "C"  0  .T.
```

Classification:

programming

Compatibility:

New in FS5

Related:

assignment, typing, Empty(), FS2:Blank()

SLEEP ()

Syntax:

`NIL = Sleep ([expN1])`

Purpose:

Sleep for the given period of seconds.

Options:

<**expN1**> is the number of seconds the application should sleep for. The minimal value is 1, which is also the default setting when no parameter is given.

Description:

SLEEP(n) is very similar to INKEY(n), both suspends the current process for the required time slice. Whilst INKEY(n) will be interrupted by a key press or ^K, it cannot be used w/o IOCTL initialization in CursesInit() and is therefore not usable for background or CGI processing. On the other hand, SLEEP(n) is not interruptible, but may freely be used also with disabled Curses and disabled IOCTL input.

An alternative is SLEEPMS() which uses milliseconds instead of full seconds in SLEEP().

Example:

```
? "Please read this message ..."
sleep(5)
?? " continuing process"
```

Compatibility:

Available in FS only. See also "man sleep"

Related:

INKEY(), SLEEPMS(), SYS.2.7

SLEEPMS ()

Syntax:

`NIL = SleepMs ([expN1])`

Purpose:

Similar to Sleep() but use Milliseconds as parameter

Arguments:

<**expN1**> is the number of Milliseconds the application should sleep for, the default value is 1000 = 1 sec.

Description:

SleepMs() can be used instead of Sleep(num) when fraction of seconds are required.

On older Unix systems without usleep() system function, i.e. w/o milli/microsecond support, Sleep(expN1/1000) is used instead.

Classification:

programming

Compatibility:

New in FS5

Related:

Sleep(), Inkey()

SOUNDEX ()

Syntax:

retC = SOUNDEX (expC)

Purpose:

Converts character strings to a soundex code suitable for indexing and searching.

Arguments:

<**expC**> is the character string to convert.

Returns:

<**retC**> is a character string converted to a soundex code in the form A999.

Description:

SOUNDEX() provides a phonetic match or sound-alike code to find a match when the exact spelling is not known. The following algorithm is used in SOUNDEX():

a. The first character of <expC> is taken in uppercase to <retC>. If ISALPHA(expC) is FALSE, return 0000.
b. The remaining characters are translated to digits:
   1 = B F P V W T
   2 = C G J K Q S X Z ß
   3 = D
   4 = L
   5 = M N
   6 = R
   All other characters, e.g. the upper / lower A E H I O U Y, special characters and spaces, are skipped
c. If two or more adjacent letters have the same digit code, all but the first letter are dropped
d. Stop if three digits are filled in <retC>. If the <expC> is shorter, fill zeros to total of three digits

Example:

```
USE magazine
INDEX ON SOUNDEX(title) TO magazine
SEEK SOUNDEX("Byte")
? FOUND(), title, RECNO()              && .T. Byte 195
SEEK SOUNDEX("bitte")
? FOUND(), title, RECNO()              && .T. Byte 195
```

Classification:

programming

Compatibility:

FlagShip uses an extended xBASE and Clipper algorithm, applicable also for other human languages. Therefore, the SOUNDEX() results are not fully identical with the DOS dialects. The Unix like SOUNDEX() used in release 4.0- is available in source (soundex2.c) and can replace the default library function.

Source:

<FlagShip_dir>/system/soundex.c and soundex2.c

Related:

INDEX, LOCATE, SEEK, SET SOFTSEEK

SPACE ()

Syntax:

retC = SPACE (expN)

Purpose:

Returns a string consisting of the indicated number of spaces.

Arguments:

<**expN**> is the number of spaces to return.

Returns:

<**retC**> is the created character string. If the numeric argument is zero or negative, null string "" is returned.

Description:

SPACE() can be used to initialize a string variable before input, or to right/left justify strings within a column. It is similar to the REPLICATE(" ", expN) function.

Note, the similar PADC(), PADL(), and PADR() functions may perform justification more effectively.

Example:

```
LOCAL name := "myself" + SPACE(24)
@ 10,10 SAY "Enter your name:" GET name
READ
? SUBSTR(name + SPACE(40), 1, 40)       // =PADR (name, 40)
```

Classification:

programming

Related:

REPLICATE(), PADL(), PADR(), PADC()

SQRT ()

Syntax:

retN = SQRT (expN)

Purpose:

Returns the square root of a positive number.

Arguments:

<**expN**> is the positive numeric value to calculate.

Returns:

<**retN**> is a numeric value, representing the square root of the argument. In case of a negative argument, zero is returned. The format of the displayed return value depends on the state of SET DECIMALS.

Description:

SQRT() is often needed in statistical calculations.

Example:

```
? SQRT(2)                                   && 1.41
? SQRT(4)                                   && 2.00
? SQRT(4) ** 2                              && 4.00
SET DECIMALS TO 5
? SQRT(2)                                   && 1.41421
```

Classification:

programming

Related:

SET DECIMALS, SET FIXED

STATBARMSG ()

Syntax:

`retC = StatBarMsg ([expC1], [expC2], [expO3])`

Purpose:

Display message <expC1> in status bar or clear status bar

Arguments:

<**expC1**> If the argument is not specified or is NIL, <retC> returns the currently displayed message. If <expC1> is "", the current status bar message is cleared.

<**expC2**> is optional color (or GUI color) for the message <expC1>

<**expO3>** is optional font object for the message <expC1>. Apply for GUI mode only, ignored otherwise.

Returns:

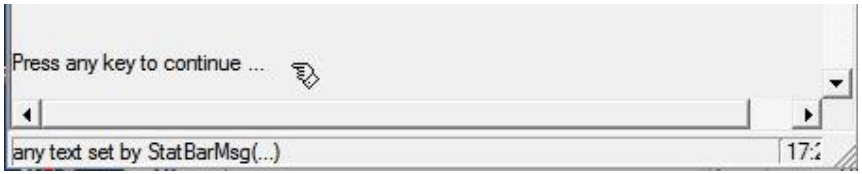
If status bar (or the default message item) is not available, <retC> returns NIL

Description:

StatBarMsg() is a subset of the generalized StatusMessage()

Example:

```
StatBarMsg("any text …")    // display message in StatusBar
Wait
StatBarMsg("")              // clear StatusBar
```



Classification:

programming

Compatibility:

New in FS5

Related:

StatusMessage(), StatusBar class

STATUSMESSAGE ()

Syntax:

retA = StatusMessage ([expCBAL1], [expC2], [expC3], [expO4])

Purpose:

Display message <expC1> in status bar or in SET MENU TO line

Arguments:

<**expC1**> is the text message. If <expC1> is "", the current status bar or SET MESSAGE TO line message is cleared.

<**expB1**> is an alternative syntax, specifying a code block to be evaluated and the returned string displayed either in SET MESSAGE line or in status bar

<**expA1**> is an alternative syntax, StatusMessage() will restore the previous screen content from array returned by previous invocation of retA := StatusMessage(...). Applicable for Terminal i/o only.

<**expL1**> is an alternative syntax. If .T. or NIL, StatusMessage() will restore the previous screen content from internal buffer. Applicable for Terminal i/o only.

<**expC2**> is the COLOR specification, same as SET COLOR. Apply in Terminal i/o mode. Considered also in GUI mode when <expC3> is not given and SET GUICOLOR ON is active.

<**expC3**> is the GUICOLOR specification, apply only in GUI mode and is ignored otherwise.

<**expO4>** is optional font object for the message <expCB1>. Apply for GUI mode only, ignored otherwise.

Returns:

Saved screen array if at least <expC1> or <expB1> was specified, or NIL otherwise

Description:

StatusMessage() is a generalized function to display a message in SET MENU TO line (Terminal i/o only) or in status bar (default in GUI or when set in Terminal i/o). Saves and restores the previous screen content occupied by the message in SET MENU TO line.

Classification:

programming

Compatibility:

New in FS5

Related:

StatBarMsg(), StatusBar class

STOD ()

Syntax:

retD = STOD (expC)

Purpose:

Converts a character string in ANSI format to a date value.

Arguments:

<**expC**> is an 8-byte character string, representing the date value. The form is equivalent to the returned value of DTOS(), in the "YYYYMMDD" format. Null date is represented by SPACE(8).

Returns:

<**retD**> is the resulting date value. If the <expC> does not contain 8 bytes, and on a wrong date, null-date is returned.

Description:

STOD() is a counterpart of the DTOS() function. It converts an ANSI string into a date value. The function is not affected by the current SET DATE, SET DATEFORMAT and SET CENTURY status.

Example:

```
? DATE()                                    //  08/20/95
? cDate := DTOS(DATE())                     // "19950820"
? STOD(cDate)                               //  08/20/95
? STOD("19831231")                          //  12/31/83
? STOD("19831131")                          //  00/00/00
? STOD("831231")                            //  00/00/00
? STOD("19831231text")                      //  00/00/00
```

Classification:

programming

Compatibility:

Available in FS and VO only, not available in C5. Embedded zero bytes are not supported.

Related:

DTOC()

STR ()

Syntax:

retC = STR (expN1 [, expN2 [, expN3]])

Purpose:

Converts a numeric expression to a character string.

Arguments:

<**expN1**> is the numeric expression to be converted.

<**expN2**> is the length of the destination character string; decimal digits, decimal point and sign included. The valid range is 1 to 500.

<**expN3**> is the number of the decimal places to return.

Returns:

<**retC**> is a formatted character string.

If <expN2> is less than the number of whole number digits in <expN1>, asterisks are returned instead of the number.

If the number of decimal places to be returned is smaller than the number of decimals in <expN1>, STR() rounds the number to fit the available number of decimal places.

If <expN3> is omitted, and <expN2> exists, the return value is rounded to an integer.

If both <expN2> and <expN3> are omitted, STR() formats the numeric value <expN1> according to the following:

<expN1> expression	<retC> length
Field Variable	Field length plus decimals
Expressions, Constants	Minimum of 10 digits plus decimals
VAL()	Minimum of 3 digits
MONTH()/DAY()	3 digits
YEAR()	5 digits
RECNO()	7 digits

Description:

STR() converts numeric values to character strings.

It can be used to concatenate numeric values to character strings, to build a mixed index key, or to display values in formats other than the SET DECIMALS format.

STR() is like TRANSFORM() which formats numeric values as character strings using a PICTURE mask instead of length and decimal specifications. The inverse operation of STR() is VAL().

Example 1:

```
LOCAL value := 34.999
? value                                 &&        34.99
? STR(value, 6)                         &&     34
? STR(value, 6, 2)                      &&  34.99
? STR(value, 6, 3)                      && 34.999
? STR(value, 6, 4)                      && ******
? STR(value, 0)                         && 0
```

Example 2:

```
USE orders
INDEX ON STR(ordnum, 6) + SUBSTR(article, 10) TO order
```

Classification:

programming

Related:

SUBSTR(), STRZERO(), VAL()

STRLEN ()

Syntax:

retN = STRLEN (expC, [expO2])

Purpose:

Reports the length of a character string, or the length of a character database field.

Arguments:

<**expC**> is the character expression for which the length should be returned.

<**expO2**> is optional Font object. If not given, the currently used font is considered.

Returns:

<**retN**> is a numeric value representing the length of a character expression. For a null-string "", zero is returned. With Unicode font, the glyphs are counted instead of their chr() representation, hence the STRLEN() may be shorter than LEN()

Description:

With a character string <expC>, each byte counts as one, including an embedded null byte CHR(0), if FS_SET("zero", .T.) is enabled. Otherwise, only the leftmost bytes up to (but not including) the zero byte are counted.

Similar but not equivalent to the function LEN() which work independent on the FS_SET("zero") setting. For strings without embedded zero bytes, both functions return the same value.

In GUI mode with Unicode font, i.e. when the given or current font is set by oFont:CharSet(FONT_UNICODE) or SET GUICHARSET FONT_UNICODE or m->oApplic:Font:CharSet(FONT_UNICODE), each glyph is counted as 1 instead or their CHR() representation which may be up to 4 chars. In such a case, STRLEN() return value is mostly shorter than LEN().

Example:

```
LOCAL  str1 := "abc"
LOCAL  str2 := str1 + chr(0) + "efg"
? STRLEN(str1), LEN(str1)                    &&  3  3
? STRLEN(str2), LEN(str2)                    &&  3  7
FS_SET("zero", .T.)
? STRLEN(str1), LEN(str1)                    &&  3  3
? STRLEN(str2), LEN(str2)                    &&  7  7
```

Classification:

programming

Compatibility:

Available in FS only. Embedded zero bytes are only supported, if FS_SET("zero") is set. Unicode is considered since VFS7.

Related:

LEN(), FIELDLEN(), FS_SET()

STRLEN2COL ()

Syntax:

```
retN = StrLen2col (expC1, [expO2])
```

Purpose:

Return the string width in fractions of columns in dependence of the passed or the currently set font.

Arguments:

<**expC1**> is the string which size is to be calculated

<**expO2**> is optional Font object. If not given, the currently used font is considered.

Returns:

Number of columns (or a fraction of) occupied by the string. With Unicode font in GUI mode, the size of glyphs are considered instead of their chr() representation.

Description:

In Terminal i/o, StrLen2col() is equivalent to Len(expC1) or StrLen(expC1).

In GUI mode, the string width depends on the used font. With fixed fonts (like Courier), the StrLen2col() return value equivalent to Len(expC1) or StrLen(expC1). With proportional fonts (like Helvetica or Times) the displayed string width may differ significantly from the string size returned by Len(expC1) or StrLen(expC1).

In GUI mode with Unicode font, i.e. when the given or current font is set by oFont:CharSet(FONT_UNICODE) or SET GUICHARSET FONT_UNICODE or m->oApplic:Font:CharSet(FONT_UNICODE), the glyph width is considered instead or their CHR() representation.

Classification:

programming

Compatibility:

New in FS5. Unicode is considered since VFS7.

Related:

Len(), StrLen(), StrLen2pix(), Col2pixel(), SET FONT

STRLEN2PIX ()

Syntax:

```
retN = StrLen2pix (expC1, [expO2])
```

Purpose:

Return the string width in pixels in dependence of the passed or the currently set font.

Arguments:

<**expC1**> is the string which size is to be calculated

<**expO2**> is optional Font object. If not given, the currently used font is considered.

Returns:

String width, i.e. number of pixels occupied by the string. With Unicode font in GUI mode, the size of glyphs are considered instead of their chr() representation.

Description:

In Terminal i/o, StrLen2pix() is equivalent to Len(expC1) or StrLen(expC1).

In GUI mode, the string width depends on the used font. The displayed string width may differ significantly from the string size returned by Len(expC1) or StrLen(expC1).

When the string starts with "<html>" or "<HTML>", it is assumed to be displayed as RichText (GUI only) and all HTML formatting <...> is removed before calculating the string size, to get same results as with ?, ??, Qout(), @..SAY, InfoBox() etc. If the text contains more than one output line (by using
 or <p>), the size of the largest line is returned.

In GUI mode with Unicode font, i.e. when the given or current font is set by oFont:CharSet(FONT_UNICODE) or SET GUICHARSET FONT_UNICODE or m->oApplic:Font:CharSet(FONT_UNICODE), the glyph width is considered instead or their CHR() representation.

Classification:

programming

Compatibility:

New in FS5. Unicode is considered since VFS7.

Related:

Len(), StrLen(), StrLen2col(), Col2pixel(), Pixel2col(), SET FONT

STRLEN2SPACE ()

Syntax:

retN = StrLen2space (expC1, [expO2])

Purpose:

Return the number of spaces required to fill the given text in fully.

Arguments:

<**expC1**> is the string which size is to be calculated in spaces

<**expO2**> is optional Font object. If not given, the currently used font is considered. With Unicode font in GUI mode, the size of glyphs are considered instead of their chr() representation.

Returns:

Number of spaces required to overwrite the <expC1> string, rounded up to integer.

Description:

In Terminal i/o, StrLen2space() is equivalent to Len(expC1) or StrLen(expC1).

In GUI mode, the string width depends on the used font. With fixed fonts (like Courier), the StrLen2col() return value equivalent to Len(expC1) or StrLen(expC1). With proportional fonts (like Helvetica or Times) the displayed string width may differ significantly from the string size returned by Len(expC1) or StrLen(expC1). Also the width of space " " character differs significantly from e.g. the "A" character.

In GUI mode with Unicode font, i.e. when the given or current font is set by oFont:CharSet(FONT_UNICODE) or SET GUICHARSET FONT_UNICODE or m->oApplic:Font:CharSet(FONT_UNICODE), the glyph width is considered instead or their CHR() representation.

You may use StrLen2space() to overwrite/clear specific text on the screen, independent on the used i/o mode. Otherwise you would in GUI need to determine the string width (and the width of spaces) by StrLen2col() or StrLen2pix() and/or use pixel coordinates.

Example:

```
LOCAL myStr := "Hello world"
@ 10,5 say myStr
:
@ 10,5 say Space(StrLen2space(myStr)) // instd.of Space(Len(myStr))
```

Classification:

programming

Compatibility:

New in FS5. Unicode is considered in VFS7.

Related:

Len(), StrLen(), StrLen2col(), StrLen2pix(), Space(), SET FONT

STRPEEK ()

Syntax:

`retN = STRPEEK (expC1, expN2)`

Purpose:

Retrieves (get) a single character from a string.

Arguments:

<**expC1**> is the target character string to examine. Passing the string by reference will speed-up the operation.

<**expN2**> is the position of the character to be returned. The valid range is between 1 and LEN(expC1).

Returns:

<**retN**> is the ASCII value of the requested character, the valid range being 0/1 to 255. If <expN2> is out of range, zero is returned.

Description:

STRPEEK() is a character function which retrieves a single character in a given string or in a copy of a memo variable. This function is much faster than the corresponding extraction method by means of SUBSTR().

Example:

```
LOCAL mystr := "abcd"
? STRPEEK (mystr, 3)                          // 99
? STRPOKE (@mystr, 3, ASC("X"))               // abXd
```

Classification:

programming

Compatibility:

Available in FS only. Embedded zero bytes are supported in FS by default. For compatibility to other xBASE dialects, use:

```
#ifndef FlagShip
   FUNCTION strpeek (str, pos)
   RETURN ASC (SUBSTR (str, pos, 1))
#endif
```

Related:

STRPOKE(), SUBSTR(), ASC()

STRPOKE ()

Syntax:

retC = STRPOKE (expC1, expN2, expN3)

Purpose:

Pokes (replaces) a single character within a string with another.

Arguments:

<**expC1**> is the target character string to change. Passing the string by reference will speed-up the replacement on the target string.

<**expN2**> is the position of the character to be replaced. The valid range is between 1 and LEN(expC1).

<**expN3**> is the ASCII value of the new character, the valid range being 0/1 to 255.

Returns:

<**retC**> is the new, changed string <expC1>. If either <expN2> or <expN3> is out of range, the original string <expC1> is returned unchanged.

Description:

STRPOKE() is a character function which enables replacing a single character in a given string or in a copy of a memo variable. This function is much faster than the addition and extraction method used in SUBSTR().

Example:

```
LOCAL mystr := "abcd"
? STRPOKE (mystr, 3, ASC("X"))                 // abXd
? mystr + STRPOKE( mystr,3,88) + mystr     // abcdabXdabcd
? mystr + STRPOKE(@mystr,3,88) + mystr     // abXdabXdabXd
```

Classification:

programming

Compatibility:

Available in FS only. Embedded zero bytes are only supported, when FS_SET("zero") is set. For compatibility to other xBASE dialects, use:

```
#ifndef FlagShip
  FUNCTION strpoke (str, pos, char)
  RETURN STUFF (str, pos, 1, CHR(char))     // Clipper 5
  ** RETURN SUBSTR(str, 1, pos-1) + ;
  **    CHR(char) + SUBSTR(str, pos+1)      // others
#endif
```

Related:

STRPEEK(), SUBSTR(), STUFF(), ASC()

STRTRAN ()

Syntax:

retC = STRTRAN (expC1, expC2, [expC3], [expN4], [expN5], [expL6])

Purpose:

Searches and replaces characters within a character string or memo field.

Arguments:

<**expC1**> is the character string to search through.

<**expC2**> is the string to locate within <expC1>.

Options:

<**expC3**> is the string to replace occurrences of <expC2> with. If this argument is not specified, occurrences of <expC1> are replaced with a null string "".

<**expN4**> is the n-th occurrence of <expC2> in <expC1> from which the replacement should begin. If this argument is not specified, the default is 1.

<**expN5**> is the number of occurrences to replace. If this argument is not specified, the default is all (or exactly: the default limit is 100000 to abort accidental endless loops). You may set <expN5> value from 1 to 2000000000 (two billions), but of course only found occurences are replaced if this <expN5> value is higher.

<**expL6**> is a logical value. If set true (.T.), the search is case insensitive. If not specified or is .F., the search is case sensitive (the default).

Returns:

<**retC**> is a character string, containing <expC1>, with specified occurrences of <expC2> replaced by <expC3>.

Description:

STRTRAN() searches and replaces within the specified character string for the defined number of occurrences.

Example 1:

```
? STRTRAN ("abcabcabc", "bc", "X", 1, 2)   // aXaXabc (2 replaced)
? STRTRAN ("abcabcabc", "a")               // bcbcbc  ('a' removed)
? STRTRAN ("abcabcabc", "a", "", 2, 1)     // abcbcabc (1a removed)

? STRTRAN ("abcabcabc", "BC", "X")          // abcabcabc (n.found)
? STRTRAN ("abcabcabc", "BC", "X", , , .T.) // aXaXaX    (found)
```

Example 2:

```
str := "abc" + chr(0) + "def" + chr(0) + "ghi"
? str, SET(_SET_ZEROBYTEOUT)          // abc?def?ghi  ?
? STRTRAN(str, chr(0), "#")           // abc?def?ghi (not replaced)
lSet := FS_SET("zero", .T.)
? STRTRAN(str, chr(0), "#")           // abc#def#ghi  (replaced)
? STRTRAN(str, chr(0), "\0", 2)       // abc?def\0ghi (2nd replac)
FS_SET("zero", lSet)                  // restore previous status
```

Classification:

programming

Compatibility:

Embedded zero bytes in <expC1>, <expC2> and <expC3> are supported when FS_SET("zero",.T.) is set but ignored otherwise. Parameters <expN5> and <expL6> are available in FS5 only.

Related:

$, SUBSTR(), STUFF(), MEMOTRAN(), HARDCR(), AT(), RAT(), STRPEEK(), STRPOKE(), FS_SET(), FS2:ReplAll(), FS2:RemAll(), FS2:RangeRem(), FS2:RangeRepl(), FS2:PosDel(), FS2:PosRepl()

STRZERO ()

Syntax:

retC = STRZERO (expN1, [expN2], [expN3])

Purpose:

Converts a numeric expression to a character string where leading spaces are replaced with 0.

Arguments:

<**expN1**> is the numeric expression to be converted.

<**expN2**> is the length of the destination character string, decimal digits, decimal point and sign included.

<**expN3**> is the number of the decimal places to return. If not given and <expN2> exists, the output is rounded to integer.

Returns:

<**retC**> is a formatted character string.

If <expN2> is less than the number of whole number digits in <expN1>, asterisks are returned instead of the number.

If the number of decimal places to return is smaller than the number of decimals in <expN1>, STR() rounds the number to the available number of decimal places.

Description:

STRZERO() can be used instead of STR() to display or print results of numeric expressions included in character text.

Example:

```
? STRZERO(34.999, 7, 3)              && "034.999"
? STR    (34.999, 7, 3)              && " 34.999"
? STRZERO(34.999, 7)                 && "0000035"
? STR    (34.999, 7)                 && "     35"
? STRZERO(34.999, 4)                 && "****"
```

Classification:

programming

Compatibility:

This function is included but not documented in Clipper. Embedded zero bytes are not supported.

Related:

STR(), SUBSTR()

STUFF ()

Syntax:

retC = STUFF (expC1, expN2, expN3, expC4)

Purpose:

Performs delete, insert and replace operations on the specified character string.

Arguments:

<**expC1**> is the target character string.

<**expN2**> is the starting position in the target string, where the operation begins.

<**expN3**> is the number of characters to replace in <expC1>.

<**expC4**> is the replacement string.

Returns:

<**retC**> is a character string, consisting of <expC1>, where <expN3> characters, beginning at <expN2> character, are replaced with <expC4>.

Description:

With different relations between <expN2>, <expN3> and the length of <expC4>, different combinations of insert, replace and delete operations can be performed.

- **Insert:** If <expN3> is zero, no characters are removed from <expC1>. The <expC4> string is inserted at the <expN2> position.
- **Replace:** If <expC4> has the same length as <expN3>, the <expC4> string replaces the original characters starting at <expN2>.
- **Delete:** If <expC4> is a null string "", the number of characters specified by <expN3> are removed, starting at the <expN2> position.
- **Replace and insert:** If <expC4> is longer than <expN3>, all <expN3> characters starting at <expN2> are removed and the string <expC4> is inserted; the resulting string is longer.
- **Replace and delete:** If <expC4> is shorter than <expN3>, all <expN3> characters starting at <expN2> are removed and the string <expC4> is inserted; the resulting string is shorter.
- **Replace and delete rest:** If <expN3> is greater than or equal to the number of characters remaining in <expC1> beginning with <expN2>, all remaining characters are removed before the string <expC4> is inserted.

Example:

```
* Insert
? STUFF("John", 4, 0, "NSO")           && JohNSOn
* Delete
? STUFF("Johnson",4, 3, "")            && John
* Replace
? STUFF("Johnson",2, 1, "a")           && Jahnson
* Replace and delete
? STUFF("Johnson",2, 3, "a")           && Jason
* Replace and insert
? STUFF("Jason",2,1,"ohn")             && Johnson
? STUFF("Müller", 2, 1, "ue")          && Mueller
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

STRTRAN(), SUBSTR(), AT(), LEFT(), RAT(), RIGHT(), PADx()

SUBSTR ()

Syntax:

retC = SUBSTR (expC1, expN2 [,expN3])

Purpose:

Extracts the specified part of the given string.

Arguments:

<**expC1**> is the source character string.

<**expN2**> is the position to start extracting at. If positive, it is counted from the beginning of the source string. If negative, it is counted from the end of the string.

<**expN3**> is the number of bytes to extract from the source string, starting at <expN2>. If it exceeds the length of the source string, the extra is ignored. If this argument is omitted, the substring extracted is taken to the end of the source string.

Returns:

<**retC**> is the extracted string. If <expN2> is greater than the length of <expC1>, or <expN3> is zero, null string "" is returned.

Description:

SUBSTR() can be used in string manipulation where parts of strings need to be used. SUBSTR() is related to the LEFT() and RIGHT() functions which extract substrings beginning with the leftmost and the rightmost characters in <expC1>, respectively.

Example:

```
LOCAL name := "Smith, Peter G."
? SUBSTR (name, 1, 5)                          // Smith
? SUBSTR (name, 1, AT(",")-1)                  // Smith
? SUBSTR (name, 1, AT("xx"))                   // ""
? x := LTRIM(SUBSTR (name, AT(",")+1))         // Peter G.
? SUBSTR (x, AT(" ")+1)                        // G.
? SUBSTR (name, RAT(" ")+1), 1)                // G
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default.

Related:

AT(), RAT(), LEFT(), RIGHT(), FS2:ReplLeft(), FS2:ReplRight(), FS2:ReplAll(), FS2:RemLeft(), FS2:RemRight(), FS2:RemAll(), FS2:RangeRepl(), FS2:Range-Rem(), FS2:PosRepl()

TBCOLUMNNEW ()

Syntax:

```
retO = TbColumnNew (expC1, expB2)
retO = TbColumn {expC1, expB2}
```

Purpose:

Create/instantiate new TbColumn object, holding the column information for Tbrowse.

Arguments:

See TbColumn{} in section OBJ.

Returns:

<**retO**> is newly instantiated TbColumn object.

Description:

Generally speaking, TbColumn is simply a helper for Tbrowse.

Classification:

programming

Related:

Tbrowse and TbColumn classes in section OBJ

TBROWSENEW ()
TBROWSEARR ()
TBROWSEDB ()

Syntax 1:

```
retO = TbrowseNew ([expN1], [expN2], [expN3],
           [expN4], [expL5], [expL6], [expC7],
           [expO8], [expN9])

retO = Tbrowse { [expN1], [expN2], [expN3],
           [expN4], [expL5], [expL6], [expC7],
           [expO8], [expN9] }
```

Syntax 2:

```
retO = TbrowseArr ([expN1], [expN2], [expN3],
           [expN4], [expL5], [expL6], [expC7],
           [expO8], [expN9], [expA10])
```

Syntax 3:

```
retO = TbrowseDb ([expN1], [expN2], [expN3],
           [expN4], [expL5], [expL6], [expC7],
           [expO8], [expN9])
```

Purpose:

Create/instantiate new Tbrowse object.

Purpose:

Creates a new TBROWSE object with predefined array movement blocks, optionally initialized by the arguments supplied.

Options:

<**expN1**> is the top screen row where TBROWSE is displayed. This argument is equivalent to assigning the obj:NTOP with the same value. The valid range is 0...MAXROW(). The default value is zero.

<**expN2**> is the leftmost screen column where TBROWSE is displayed. This argument is equivalent to assigning the obj:NLEFT with the same value, the valid range is 0...MAXCOL(). The default value is zero.

<**expN3**> is the bottom screen row where TBROWSE is displayed. This argument is equivalent to assigning the obj:NBOTTOM with the same value. The valid range is <expN1>...MAXROW(). The default value is MAXROW().

<**expN4**> is the rightmost screen column TBROWSE is displayed. This argument is equivalent to assigning the obj:NRIGHT with the same value. The valid range is <expN2>...MAXCOL(). The default value is MAXCOL().

<**expL5**> is the pixel specification. If .T., the coordinates given are assumed in pixel. If .F., the coordinates are in row/column. If not given and SET PIXEL or SET COORD is not set, the GUI coordinates are calculated from current SET FONT (or oApplic:Font)

<**expL6**> specifies whether the Tbrowse widget is resizable by user or not. Default is .F. which means the Tbrowse widget is fix. Apply for GUI mode only, ignored otherwise.

<**expC7**> is a ToolTip string. Apply for GUI mode only, ignored otherwise.

<**expO8**> is a Font object. If not specified, the oApplic:Font is used. Apply for GUI mode, ignored otherwise.

<**expN9**> specify the row height in pixel, see also tb:RowHeight. If not specified, the size of one row is used. Apply for GUI mode, ignored otherwise.

<**expA10**> is the array to browse. If nor specified, the array need to be assigned by tb:UserArray

Returns:

<**obj**> is the newly allocated TBROWSE object, usually assigned to a regular FlagShip variable.

Description:

Creates flexible table to display any data in rows and columns. The used columns needs to be prepared by `TbColumNew()` or `TbColumn{}` and assigned to the Tbrowse object via `:AddColumn()` method, see examples in OBJ.Tbrowse.

Syntax 1 is fully equivalent to instantiation of `Tbrowse{}`. The required skip blocks (i.e. handling the movement in rows) needs to be explicitly assigned.

Syntax 2 is similar to syntax 1, but the skip blocks are created automatically supporting display from an array. The array itself may be passed as 10th parameter.

Syntax 3 is similar to syntax 1, but the skip blocks are created automatically supporting display from a database.

If the optional arguments are supplied, the corresponding instance variables are filled with these values.

Prior to using the TBROWSE object, at least one or more TBCOLUMNs (see `tb:ADDCOLUMN()`) must be specified. See additional description in section OBJ.Tbrowse and OBJ.TbColumn

Tuning:

See OBJ.TbrowseNew()

Example 1:

Browse thru a database. See also examples/tbrowse_db.prg

```
dbfName := "personal"
USE (dbfName) SHARED NEW                 // open database
oTbr := TbrowseDb(3,0, 24,maxcol()-1 )   // create browser
for ii := 1 to Fcount()                  // add columns
   oTbr:AddColumn( TbColumnNew(FieldName(ii), ;
```

```
                                    FieldBlock(FieldName(ii)) ) )
next
oTbr:Exec()                                    // execute browser
```

Example 2:

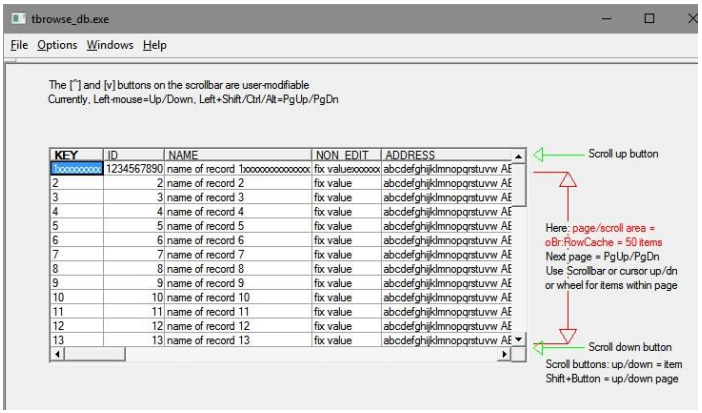
Browse thru multi-dimensional array 'myArray'. See more in section OBJ.Tbrowse and OBJ.TbrowseArr() and examples/tbrowse_ar.prg

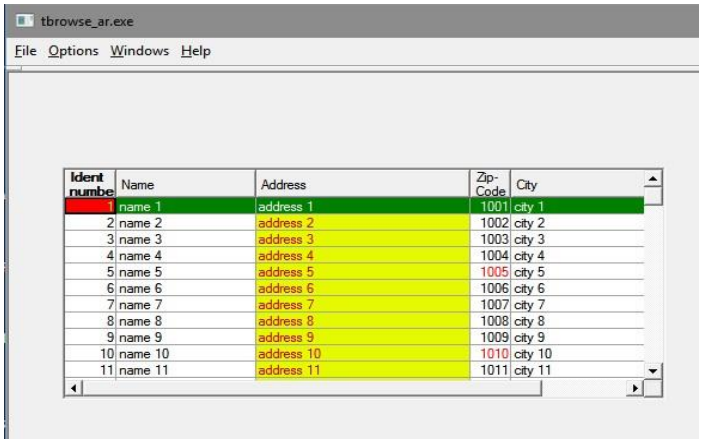
```
myArray := {{"Maier","John","M",.T.,ctod("15-03-74")}, ;
            {"Smith","Ann", "F",.F.,ctod("11-09-01")}, ;
            {"Scott","Paul","M",.T.,ctod("09-12-82")} }
aHeader := {"Name","First","Gender","Married","Born"}

oBr := TbrowseArr(6,5, 20,60, NIL, NIL, "My Browse")
oBr:UserArray := myArray                    // assign array
for ii := 1 to len(myArray[1])
   oTbcol := TbColumnNew(aHeader[ii], .T.) // use def. array block
   oBr:AddColumn(oTbcol)
next
oBr:Exec()                    // calls UDF assigned by :Handler
```

Example 3:

Additional examples are in *<FlagShip_dir>/examples/tbrowse_db.prg*, …/examples/*tbrowse_arr.prg*, …/examples/*unicode.prg* and in section OBJ.Tbrowse

GLYPHS	TEXT	NUM
text1: ターボット CO 4OZ(テスト)	text1	1
text2: 保税ブラックツナ4OZ CO	text2	2
text3: サーモン 5OZ COBF	text3	3
text4: 保税牛肉 SS(5*) 130G ｶｯﾄ	text4	4
text5: 保税牛肉 120G ｶｯﾄ	text5	5
text6: 保税ﾘﾌﾞｱｲ12ｵﾝｽ08942	text6	6
text7: 保税牛肉ｻｰﾛｲﾝ10ｵﾝｽ 08948	text7	7
text8: 保	text8	8
Hello Japan : 日本こんにちは	Hello Japan	9
Hello Chinese: 你好中國	Hello Chinese	10
Hello Korea : 한국 안녕	Hello Korea	11
Hello Arabic : مرحبا العربية	Hello Arabic	12
Hello Thai : สวัสดีไทย	Hello Thai	13
Hello Hindi : नमस्कार हिन्दी	Hello Hindi	14
Hello Hebrew : שלום עברית	Hello Hebrew	15
empty text:	empty text	16
text1: ターボット CO 4OZ(テスト)	text1	17
text2: 保税ブラックツナ4OZ CO	text2	18
text3: サーモン 5OZ COBF	text3	19
text4: 保税牛肉 SS(5*) 130G ｶｯﾄ	text4	20
text5: 保税牛肉 120G ｶｯﾄ	text5	21
text6: 保税ﾘﾌﾞｱｲ12ｵﾝｽ08942	text6	22





Classification:

programming

Class:

uses TBROWSE class, prototyped in <FlagShip_dir>/include/ tbrclass.fh

Compatibility:

TbrowseArr() is new in FS5 The source code for TbrowseArr() and TbrowseDb() is available in the <FlagShip_dir>/system directory.

Related:

Tbrowse and TbColumn classes in section OBJ

TBMOUSE ()

Syntax:

retN = TbMouse (expO1, expN2, expN3, [expL4])

Purpose:

For compatibility purposes to undocumented Clipper function of the same name only.

Arguments:

<expO1> is Tbrowse object <expN2> is the current mouse row coordinate <expN3> is the current mouse column coordinate <expL4> is pixel specification or NIL

Returns:

<**retN**> is TBR_CONTINUE or TBR_EXCEPTION, see buttons.fh

Description:

The function is a wrapper to equivalent functionality of the Tbrowse class method expO1:GoMousePos(expN2, expN3, expL4)

Classification:

programming

Compatibility:

New in FS5, for compatibility to CL53 only

Related:

Tbrowse class in section SYS

TEMPFILENAME ()

Syntax:

`retC = TEMPFILENAME ([expC1], [expC2], [expC3])`

Purpose:

Returns a unique file name for the current or specified directory.

Options:

<**expC1**> is the directory path in which a unique file name should be created. If omitted or specified null-string or NIL, the current directory ("./") is used. Note, the environment variable TMPDIR will override this parameter.

<**expC2**> is a favorite file name initial letter(s). If not given, or specified null-string or NIL, a fully random name is generated.

<**expC3**> is an optional extension, with or without period. If not given, or specified null-string or NIL a file name without extension is generated.

Returns:

<**retC**> is the generated, random and unique file name, with or without extension. On error, null-string "" is returned.

Description:

This function generates file names (but not the file self) that can be safely used for temporary files. It avoids accidentally overwritten files (by/from others), usually named e.g. "tempor.doc", "tmp.dbf" etc. The algorithm and returned string is similar (but not equivalent) to the system function tempnam(), i.e.

```
[<path>]/[<prefix>]<PID><postfix>[.<ext>]
```

where the

<path>	is either <expC1> or ./ or /tmp or TMPDIR envir.var
<prefix>	are up to the first 5 characters of <expC2>
<PID>	is the process ID number, last 5 digits
<postfix>	are three random characters
<ext>	is <expC3> if specified

If the file would not have access to the given directory, either TMPDIR or /tmp directory is used instead. If you wish to create the temp file self (not only it name), use FS2:TEMPFILE() instead.

In MS-Windows, TempFileName() achieves cross source compatibility to Unix and Linux:

- when <expC1> is not given or is empty, current directory is used
- when <expC1> is "./" or ".\", the current directory is used, same as when passing CurDir() value
- when <expC1> is "/tmp", the environment variable %TEMP% or %TMP% or %TMPDIR% (in that order) is used if such available, otherwise the current directory
- the "/" path delimiters are accepted and translated to "\"

Example:

```
cTmp := TempFileName()                    //  ./00647aaa
cTmp := TempFileName("/tmp", "my")        //  /tmp/my00647du5
dbcreate (cTmp + ".dbf", myDbfStru)       //  /tmp/my00647du5.dbf

cTmp := TempFileName("./", "my", "dbf")   //  /tmp/my00647baa.dbf
if !empty(cTmp)
   dbcreate (cTmp, myDbfStru)             //  /tmp/my00647baa.dbf
else
   ? "cannot create temp file in current nor in /tmp directory"
endif
```

Classification:

programming, system

Compatibility:

Available in FlagShip only. The 3rd parameter is available in FS5 only.

Related:

FS2:TEMPFILE(), SYS(3) in <FlagShip_dir>/system/foxpro_api.prg

TEXTBOX ()

Syntax:

`retN = TextBox (expC1, [expC2], [expN3])`

Purpose:

Display an text message box dialog, similar to Alert()

Arguments:

<**expC1**> is a string containing the displayed text, the lines are separated by semicolon ";". See additional details in Alert() description.

<**expC2**> is an optional header text.

<**expN3**> is optional time-out in seconds. Zero or NIL specify to wait until user press key or mouse selection. When the time-out expires without user action, the box is closed and <retN> is set to 0.

Description:

The TextBox() function is available for your convenience. It is equivalent to TextBox{NIL,cHeader,cMessage}:Show() described in section OBJ. In GUI mode, plain box without icon is displayed. In Terminal i/o mode, the color is specified in global variable

```
_aGlobSetting[GSET_T_C_TEXTBOXCOLOR] := "W+/B,B/W"   // default
```

and can be re-assigned according to your needs. You also may set border via e.g.

```
_aGlobSetting[GSET_T_C_TEXTBOX] := B_SINGLE // default = B_PLAIN
_aGlobSetting[GSET_T_C_TEXTBOX] := B_DOUBLE // default = B_PLAIN
```

Example:

```
_aGlobSetting[GSET_T_C_TEXTBOXCOLOR] := "W+/B,B/W"  // default
_aGlobSetting[GSET_T_C_TEXTBOX     ] := B_SINGLE    // terminal i/o

TextBox ("This is any text displayed by TextBox()", "Info", 10)
```

Compatibility:

New in VFS5. The expN3 is supported since VFS7

Related:

Alert(), InfoBox(), ErrBox(), WarnBox(), MessageBox{}, InfoBox{}, ErrorBox{}, WarnBox{}, TextBox{}

TEXTBOXNEW ()

Syntax:

```
retO = TextBoxNew ([oOwner], [cTitle], [cText])
```

Description:

This is alternative syntax for creator of TextBox class, see details in section OBJ

Related:

TextBox Class, TextBox()

TIME ()

Syntax:

`retC = TIME ([expN1])`

Purpose:

Reports the system time.

Options:

<**expN1**> is an optional modifier for the returned string:

NIL or 0:	Return the system time as "hh:mm:ss" (default)
1:	Return the system time as "hh:mm:ss.uuuu"
2:	Return the system time as "DD-MM-YYYY hh:mm:ss.uuuu"
3:	Return the system time as "YYYY-MM-DD hh:mm:ss.uuuu"
4:	Return the system time as "MM/DD/YYYY hh:mm:ss.uuuu"
5:	Return the system time as "hh:mm:ss AM" or "hh:mm:ss PM"
6:	Return the system time as "hh:mm:ss a.m." or ".... p.m."
other:	same as TIME() or TIME(0)

where "h" represents hour, "m" minute, "s" second, "u" millisecond, "D" day (01..31), "M" month (01..12) and "Y" 4 digits of year.

Returns:

<**retC**> is a character string, representing the system time "hh:mm:ss" (hours, minutes, seconds) in 24-hour format (00:00:00 to 23:59:59). The returned value may be modified by the optional parameter.

The 12-hour format with TIME(5) starts at "12:00:00 AM" (midnight), goes thru "12.59.59 AM" .. "01:00:00 AM" to "11:59:59 AM", switches to "12:00:00 PM" (noon) and continues to "11:59:59 PM". The same is valid for TIME(6).

Description:

TIME() is used to display the system time. TIME() is related to SECONDS() which returns the number of seconds since midnight. The alternative SECONDSCPU() function determines number of seconds (and its millisecond fragments) since the program start.

Example:

```
? TIME()                                  // 15:28:12
? minute := val(substr(TIME(), 4, 2)) // 28
? TIME(0)                                 // 15:28:12
? TIME(1)                                 // 15:28:12.5342
? TIME(2)                                 // 27-02-2013 15:28:12.5345
? TIME(3)                                 // 2013-02-27 15:28:12.5348
? TIME(4)                                 // 02/27/2013 15:28:12.5352
? TIME(5)                                 // 03:28:12 PM
? TIME(6)                                 // 03:28:12 p.m.
SET DATE GERMAN
SET CENTURY OFF
? DATE(), left(TIME(),5)                  // 27.02.13 15:28
```

```
SET DATE BRITISH
SET CENTURY ON
? DATE(), TIME(5)                    // 27/02/2013 03:28:12 PM
? DATE(), upper(TIME(6))             // 27/02/2013 03:28:12 P.M.
? left(TIME(2), 16)                  // 27-02-2013 15:28
? CMONTH() + " " + ltrim(DAY()) + ", " + ltrim(YEAR()) + ;
  " at " + left(TIME(6),5) + substr(TIME(6),9)
                                  // February 27, 2013 at 03:28 p.m.
```

Classification:

programming, system

Compatibility:

The correct setting of the environment variable TZ is significant for the proper calculation of system time.

The optional parameter is new in VFS and not available in Clipper, LEFT(TIME(1),11) corresponds to FoxPro's TIME(n).

In earlier VFS versions prior 7.1.20 the year was 2 digits only.

Related:

DATE(), SECONDS(), SECONDSCPU() FS2:TimeValid(), FS2:TimeToSec(), FS2:SecToTime(), FS2:SetTime()

TONE ()

Syntax:

NIL = TONE ([expN1, expN2])

Purpose:

Produces a beep in FlagShip. On the DOS system, TONE() produces a tone using the frequency generator.

Arguments:

<**expN1**> is the tone frequency (DOS only).

<**expN2**> is the tone duration in 1/18 seconds (DOS only).

Returns:

The function always returns NIL.

Description:

This function is included for compatibility purposes only. You may also use ?? chr(7) instead.

Example:

```
Warns the user in case the wrong number of parameters is passed

PROCEDURE myproc
PARAMETERS First, Second, Third
IF PCOUNT() < 3
   ? "Usage: ", PROCNAME()
   ?? " <first>, <second>, <third>"
   TONE (300,5)                        // beep in FlagShip
   RETURN
ENDIF
```

Classification:

sequential output

Compatibility:

FlagShip produces a single beep. The arguments, if any, are ignored.

Related:

SET BELL, ?? chr(7)

TRANSFORM ()

Syntax:

retC = TRANSFORM (exp1, [expC2])

Purpose:

Forms the results of an expression of the "C", "N", "D", and "L" type to a formatted string - or any other type to the default string similar to QQOUT() printout.

Arguments:

<**exp1**> is the character, numeric, date or logical expression to format. If not given, the standard formatting to string, similar to Qqout() is used.

<**expC2**> is the format PICTURE of the returned string.

Returns:

<**retC**> is the resulting formatted string.

Description:

TRANSFORM() is a conversion function that formats a given expression in the same way the PICTURE clause of @..SAY does, e.g. for display purposes.

For more information, refer to the @..SAY command, PICTURE clause.

Example:

```
? TRANSFORM("Smith","@!")                    && SMITH
? TRANSFORM(123.456,"999.9")                 && 123.4
? TRANSFORM(123.456,"@R 9 9 9.9")            && 1 2 3.4
```

Classification:

programming

Compatibility:

Embedded zero bytes are not supported.

Related:

@...SAY...PICTURE, STR(), LOWER(), UPPER(), PADx()

TRIM ()
RTRIM ()

Syntax:

retC = TRIM (expC1|expN1)

or:

retC = RTRIM (expC1|expN1)

Purpose:

Removes all trailing spaces from a string.

Arguments:

<**expC1**> is the character string to be trimmed.

<**expN1**> is a numeric value which is converted to string first and then trimmed.

Returns:

<**retC**> is a character string, with all trailing blanks removed. In case of a null string, TRIM() returns a null string "".

Description:

TRIM() and RTRIM() are alternative names for the same function. It can be used to remove trailing blanks when several strings are used to form an expression like an address or title.

Refer also to the description of the RTRIM() function. The complementary operation of TRIM() is LTRIM() or PADR().

Example:

```
Name = "John        "
Last = "Smith       "
? TRIM(Name)+" "+Last                  && "John Smith      "
? LEN(TRIM(SPACE(20))                  && 0
```

Classification:

programming

Compatibility:

FS supports embedded zero bytes by default. Support of numeric value is new in FS44

Related:

RTRIM(), ALLTRIM(), LTRIM(), SUBSTR(), PADx()

TRUEPATH ()

Syntax:

`retC = TRUEPATH (expC1)`

Purpose:

Determines the true path or the full file name including a path.

Arguments:

<**expC1**> is either a relative path to be converted to an absolute one, or a file name of which the full path is to be determined the same way as the FILE() function searches for it. If only a path is specified, the string has to be terminated by the "\" or "/" character.

Returns:

<**retC**> is the expanded absolute path, optionally including the file name. If the file is not found, a null string "" is returned. If only a path is specified, no check for its existence is performed.

Description:

TRUEPATH() is an environment function determining the absolute Unix/Windows path. If a path only is specified, the FS_SET() translations and the drive substitution (using the environment variable x_FSDRIVE) is considered. If a file name is given as well, the current SET PATH is also considered.

Example:

```
? CURDIR  ()                    && /usr/john/tmp
? TRUEPATH ("./")               && /usr/john/tmp/
? TRUEPATH ("..\..\data\")      && /usr/data/
? FILE (TRUEPATH("..\..\data\"))&& .T.   (/usr/data exists)
? TRUEPATH ("myfile.txt")       && /usr/john/tmp/myfile.txt
? TRUEPATH ("../adr/adr.dbf")   && /usr/john/adr/adr.dbf
SET PATH TO ../../src,../adr
? TRUEPATH ("adr.dbf")          && /usr/john/adr/adr.dbf
FS_SET ("lower, .T.)
? TRUEPATH ("Adr.DBF")          && /usr/john/adr/adr.dbf
? GETENV   ("D_FSDRIVE")        && ../../data/
? TRUEPATH (GETENV("D_FSDRIVE"))&& /usr/data/
? TRUEPATH ("D:Mydata.X")       && /usr/data/mydata.x
? TRUEPATH ("Mydata.X")         && ""
? TRUEPATH ("Myprog.prg")       && /usr/src/myprog.prg
FS_SET ("trans", "ntx", "idx")
? FILE ("ADR.NTX")              && .T.
? TRUEPATH ("ADR.NTX")          && /usr/john/adr/adr.idx
```

Classification: programming

Compatibility:

Available in FS only.

Related:

FindExeFile(), FS_SET(), FILE(), SET PATH, x_FSDRIVE

TYPE ()

Syntax:

`retC = TYPE (expC)`

Purpose:

Retrieves the type of the contents of a character expression.

Arguments:

<**expC**> is an expression of character type, for which the type of the result is to be determined. The expression can be a field, with or without the alias, a PRIVATE or PUBLIC variable, or an expression of any type.

Returns:

<**retC**> is one of the following characters:

retC		Variable/Field/Expression	
A		Array, Sub-array	
B		Code Block	
C		Character	
D		Date	
E	*	short string, usually displayed as C	
F		Float num, when FS_SET("intv",.T.) is set	
I	*	integer numeric, when FS_SET("intvar") is set	
2	*	binary short int = +/-32767	(.dbf field)
4	*	binary long int = +/-2147483647	(.dbf field)
8	*	binary double float	(.dbf field)
L		Logical	
M		Memo	(.dbt field)
N		Numeric	
O		Object	
S		Screen variable	
U		Undefined, NIL, Local, Static	
UE		Syntax error	
UF		Undefined function error	
UI		Other errors	
VC	*	variable character size	(.dbv field)
VCZ	*	compressed var character size	(.dbv field)
VB	*	variable binary/blob data	(.dbv field)
VBZ	*	compressed var binary/blob data	(.dbv field)

(*) FlagShip specific

Description:

TYPE() determines the type of a PRIVATE, PUBLIC or FIELD variable, public UDF and standard function by evaluating the argument using the macro evaluator. Therefore, LOCAL, STATIC and TYPED variables, as well as STATIC FUNCTIONs cannot be supported and will lead to "U" or "UF" being returned.

To determine the type of any visible variable, including STATIC and LOCAL, use VALTYPE() instead. To check the existence of any public UDF or standard function without executing it, use ISFUNCTION().

References to PRIVATE and PUBLIC arrays and sub-arrays return "A." References to array elements return the type of the element.

TYPE() can only test the validity of parameters received using the PARAMETERS statement. Testing a parameter declared as part of a FUNCTION or PROCEDURE declaration always returns "U". For these formal parameters, use PCOUNT() and/or VALTYPE() instead.

Example:

```
DECLARE info[5]
LOCAL var1 := {|| NIL}
info[1] = .T.
info[2] = 12.01
info[3] = CTOD("06/07/93")
info[4] = "Jones"
info[5] = {1, 2, 3}
? TYPE ("info")                          &&   A
? TYPE ("info[1]")                       &&   L
? TYPE ("info[2]")                       &&   N
? TYPE ("info[3]")                       &&   D
? TYPE ("info[4]")                       &&   C
? TYPE ("info[5]")                       &&   A
? TYPE ("info[5,2]")                     &&   N
? TYPE ("SUBSTR(info[4],1,2)")           &&   C  (std function)
? TYPE ("SOUNDEX('')")                   &&   UF (std, not linked)
? TYPE ("myudf(1, 2)")                   &&   L  (returns logical)
? TYPE ("help()")                        &&   U  (UDP, returns NIL)
? TYPE ("unknownudf()")                  &&   UF (not available)
? TYPE ("var1")                          &&   U

? VALTYPE (var1)                         &&   B
? ISFUNCTION ("help")                    &&  .T.
? ISFUNCTION ("SUBSTR")                  &&  .T.
? ISFUNCTION ("SOUNDEX")                 &&  .F. (not linked)
? ISFUNCTION ("myudf")                   &&  .T.

PROCEDURE help (p1, p2, p3)
RETURN
FUNCTION myudf (p1, p2, p3)
? PCOUNT(), VALTYPE(p2)                  &&   2 N
? PROCNAME(1), PROCLINE(1)               &&   TEST  17  (=TYPE(..))
RETURN .T.
```

Classification:

programming

Compatibility:

FlagShip will also evaluate (the linked) standard functions, UDPs and UDFs properly. To check the ability of a PROCEDURE, use the UDF syntax; see example above.

Related:

VALTYPE(), ISFUNCTION(), PCOUNT(), ISOBJPROP()

UPDATED ()

Syntax:

`retL = UPDATED ()`

Purpose:

Determines if there was a change in any of the processed GETs during the last or the current READ. If you wish to change the flag manually, use ReadUpdated().

Returns:

<**retL**> is logically TRUE if there were changes, or FALSE if there were none.

Description:

UPDATED() determines whether characters were successfully entered into a GET from the keyboard during the last or the current READ.

Each time READ executes, UPDATED() is set to FALSE. Then, any change to a GET entered from the keyboard or pushed to the keyboard buffer by KEYBOARD sets UPDATED() to TRUE. This happens after the GET exits or before a VALID clause or the SET KEY procedure is executed. Changes of the get:BUFFER in a post-valid function will be not recognized by UPDATED().

Once UPDATED() is set to TRUE, it retains this value until the next READ is executed or manually reset by ReadUpdated(.F.).

Example:

```
LOCAL how_much
USE salary
how_much := salary->how_much
SET KEY -2 TO hundred
@ 10,0  SAY "Enter amount:" GET how_much PICT "999999.99"
READ
IF UPDATED()
   REPLACE salary->how_much WITH salary->how_much + how_much
ENDIF
PROCEDURE hundred                       // when F3 is pressed:
KEYBOARD "100"                          // write 100 to buffer
RETURN
```

Classification:

programming

Compatibility:

Unlike Clipper, FlagShip returns .T. only if the changes were really made, not by retyping the value with the same characters.

Source:

The function is available in <FlagShip_dir>/system/getsys.prg

Related:

@...GET, READ, getsys.prg, ReadUpdated()

UPPER ()

Syntax:

retC = UPPER (expC)

Purpose:

Converts all alphabetic characters of a string to uppercase.

Arguments:

<**expC**> is a character string to be converted to uppercase.

Returns:

<**retC**> is a character string, with all lowercase alphabetic characters converted to uppercase according to their ASCII values. If the FS_SET ("loadlang") language table is used, other conversions may be specified.

Description:

UPPER() is generally used to format character strings for display purposes, or to normalize strings for case-independent comparison or indexing.

For proper Upper() conversion of extended character set > chr(127), the FSsortab.def - or other table assigned by FS_SET("Setl"/"Load") is used.

The inverse operation of UPPER() is LOWER().

Example:

```
? UPPER("abcde")                          && ABCDE
? UPPER("1266 Maple St.")                 && 1266 MAPLE ST.
USE address
INDEX ON UPPER(name) TO name              && normalize
SEEK "SMITH"
```

Classification:

programming

Compatibility:

User defined tables for upper/lower case translation are loaded by FS_SET("load") and FS_SET("setlang"). They are available in FlagShip only. Embedded zero bytes are supported in FS by default.

Related:

ISUPPER(), LOWER(), ISLOWER(), ISALPHA(), FS_SET()

USED ()

Syntax:

retL = USED ()

Purpose:

Determines whether the working area selected has a database file in use.

Returns:

<**retL**> is TRUE if there is a database file in use, or FALSE if there is none.

Description:

USED() is a database function that determines whether there is a database file in USE in a particular working area. Also other database functions, like DBF(), ALIAS(), FCOUNT() etc. will determine if a database is in USE.

To apply the function in a different working area from the current one, alias->(USED()) may be used.

Example:

```
SELECT 1
USE Article
? USED()                                  &&  .T.
SELECT 2
? USED()                                  &&  .F.
```

Classification:

database

Related:

SELECT, USE, SELECT(), oRdd:Used

USERSACTIVE ()

Syntax:

`retN = USERSACTIVE ()`

Purpose:

Determines the number of currently active users.

Returns:

<**retN**> is a numeric value representing the number of users executing the same application, including the caller. The valid range is 1 to USERSMAX().

Description:

USERSACTIV() can be used for developer purposes, e.g. to check the number of users of the current application.

Example:

```
#define MY_HIDDEN_NUMBER  10

IF USERSACTIV() > MY_HIDDEN_NUMBER
   ? "Sorry, your purchased software supports " + ;
     LTRIM(STR(MY_HIDDEN_NUMBER)) + " users only."
   ? "Try later..."
   QUIT
ENDIF
```

Technical note:

You should never kill a process (this is very similar to plug the power cable off <g>) since the application CANNOT quit properly and restore the status. See also sect. FSC.3.2.

FlagShip updates the user's count when the application terminates regularly. When some process irregularly dies or was hard killed, the user information does not (cannot) get updated correctly. The user count is stored in shared memory, so it is cleared automatically when rebooting the system.

In Linux, you (or your sysadmin) may fix it manually:

1) Let all users log off from this application
2) Invoke as root or su or sudo

   ```
   ipcs -m
   ```

   You will receive a list of descriptors. Look for your/the user name in the 'owner' column and note <id1>, <id2>,... in the 'shmid' column,usually occupying 24 bytes.
3) Invoke

   ```
   ipcrm shm <id1>  [<id2> ... ]
   ```

   or

   ```
   ipcrm -m <id1> [-m <id2> ...]
   ```

   which removes the shared segment(s) according to the above ipcs output.

You also can remove the user-own segments without being su/root. But be **very** careful what you are doing. Removing wrong shared memory segments may hang your or other application or produce unpredictable results.

Classification:

programming

Compatibility:

Available in FS only.

Related:

USERSDBF(), USERSMAX()

USERSDBF ()

Syntax:

retN = USERSDBF ([expC1], [expL2])

Purpose:

Determines the number of users simultaneously using the same database.

Options:

<**expC1**> is a name of the database, optionally including path and extension to check before opening it with USE. The SET PATH and SET DEFAULT is considered. If no extension is specified, .dbf is used. If the argument is omitted, the database in the current working area is retrieved.

<**expL2**> is a logical value specifying if the <expC1> database, when already open in the current application, should be counted as well. The default is .F. which means that database <expC1> only open by others is counted; this behavior is compatible to previous FlagShip 4.x versions.

Returns:

<**retN**> is a numeric value, representing the count of users simultaneously using the same database. The valid range is 0/1 to USERSMAX(), or:

-1 = Error. When <expC1> is given: could not open this file. When <expC1> is not specified: no database in use in the current work area
-2 = only with <expC1>: database is locked exclusively by others
-3 = only with <expC1>: database is locked exclusively by this executable

Description:

USERSDBF() can be used for system developer purposes, for example, to check the total usage level of the current database.

If the <expC1> argument is not given, the database in the current working area is counted in <retN>. Id <expC1> is given, an already open same named database in the current application is counted in dependence on <expL2>, i.e. not per default.

If the database is open simultaneously in different working areas in this application, it is counted only once. To determine the concurrent use by the same application, use IsDbMultipleOpen()

Example 1:

```
? USERSDBF ("address")                    // 0  (not in USE)
USE address NEW
? USERSDBF ()                             // 1  (that's me)
? USERSDBF ("address")                    // 0  (nobody else)
? USERSDBF ("address", .T.)               // 1  (only self)
USE otherbase NEW
? USERSDBF ()                             // 1  (only self)
? address->(USERSDBF())                   // 1  (only self)
```

Example 2:

```
if USERSDBF("address") == -2
   ? "Cannot open address.dbf, used exclusively by other applic"
   return
endif
USE address SHARED NEW
IF NETERR()
   ? "Cannot open address.dbf"
   RETURN
ENDIF

IF USERSDBF() == 1                      // that's my self
   ? "Will perform the balance now"
   FLOCK()
   do_balance()
   UNLOCK
ELSE
   ? "Cannot perform the balance now, since", ;
     USERSDBF() -1, "other users need the data. Try later"
ENDIF
```

Example 3:

```
#define MY_HIDDEN_NUMBER  10
LOCAL dbf_users
dbf_users := USERSDBF ("address")
DO WHILE .T.
   IF dbf_users >= MY_HIDDEN_NUMBER
      ? "Sorry, cannot open additional database, since"
      ? "your purchased software supports " + ;
        LTRIM(STR(MY_HIDDEN_NUMBER)) + " .dbf users only."
      ? "Try later..."
   ELSEIF dbf_users >= USERSMAX()           // avoid RTE
      ? "Cannot open address.dbf because current", ;
        dbf_users, "already is using the database."
      ? "Since my FlagShip is limited to", ;
        USERSMAX(), "users, at least",
        dbf_users - USERSMAX() + 1, ;
        "OTHER applications use MY database."
      ? "Terminate the OTHER program or try later."
   ELSE
      EXIT
   ENDIF
   WAIT
   IF LASTKEY() = 27
      QUIT
   ENDIF
ENDDO
USE address
```

Classification:

programming, database

Compatibility: Available in FS only

Related:

USERSDBF(), USERSMAX(), ISDBMULTIPLE()

USERSMAX ()

Syntax:

```
retN = USERSMAX ()
```

Purpose:

Determines the maximum supported number of users for the current FlagShip license.

Returns:

<**retN**> is a numeric value, representing the number of users allowed to use the same application or database simultaneously. The returned value is 1 for a single-user license, 4 for a four- user, etc. or 1024 for an unlimited license.

Description:

USERSMAX() can be used for system developer purposes.

Refer to the section GEN.2 and GEN.3.3.7 for more information.

Example:

see examples in USERSACTIVE() and USERSDBF().

Classification:

programming

Compatibility:

Available in FS only.

Related:

USERSACTIVE(), USERSDBF()

UTF16_UTF8 ()

Syntax:

`retC = UTF16_UTF8 ( expN )`

Purpose:

Converts Unicode UTF-16 number to UTF-8 string

Arguments:

<**expN**> is the Unicode representation in UTF-16 encoding.

Returns:

<**retC**> is the resulting string in UTF-8 encoding, with one to six characters. On failure, empty string "" is returned.

Description:

UTF16_UTF8() converts Unicode representation of UTF-16 encoding (e.g. used in MS-Windows) to UTF-8 string, often resulting in multi-bytes output.

Refer to the section LNG.5.4.5 for more information about Unicode.

Example 1:

```
str := utf16_utf8(26412) // 0x672C to chr(230,156,172) = E6 9C AC
str += utf16_utf8(12371) + utf16_utf8(12435)  // addit.glyphs
oApplic:Font:CharSet(FONT_UNICODE)
? str
```

Example 2:

see example in <FlagShip_dir>/examples/unicode.prg

Classification:

programming, input/output of Asian and other languages in GUI mode

Compatibility:

Available in VFS7 and later only.

Related:

CP437_UTF8(), LNG.5.4.5, @..SAY, ?, ??, SET GUICHARSET, OBJ.Font

VAL ()

Syntax:

`retN = VAL ( expC )`

Purpose:

Converts a numeric string to a numeric value.

Arguments:

<**expC**> is the character string to convert. The string may contain leading spaces, sign, digits and deci point. All other trailing characters are discarded. Accepts also exponential values "nn.nnEnn", "nn.nnE+nn", "nn.nnE-nn" in the syntax [spaces][sign]<digits>[.deci][+/-]e<exp>

Returns:

<**retN**> is the resulting numeric value including decimal digits, if any. If no valid digits are encountered, zero is returned.

Description:

VAL() is a conversion function that converts a character string containing numeric digits to a numeric value.

When VAL() encounters a non-numeric character, a second decimal point, the end of the string, or embedded zero byte, it stops the evaluation. Leading spaces are ignored.

When SET FIXED is ON, VAL() returns the number of decimal places specified by SET DECIMALS, rounding the <retN> value, if <expC> contains more digits than the current DECIMALS value. When SET FIXED is OFF, VAL() returns the number of decimal places specified in <expC> and the length of the returned number is calculated from the leading spaces, sign and significant digits including decimal point. The size of exponential entry is variable to fit the result.

The inverse operations of VAL() are STR() or TRANSFORM().

Example:

```
? VAL (" 01.1")                         && 1.1
? VAL ("No. 1")                         && 0
? VAL (" ")                             && 0
? VAL ("1 23.9")                        && 1

SET FIXED ON
SET DECIMAL TO 3
? VAL ("123.4")                         && 123.400
? VAL ("123.4569")                      && 123.457
```

Classification:

programming

Related:

STR(), TRANSFORM(), STRZERO(), SUBSTR(), SET DECIMALS, SET FIXED

VALTYPE ()

Syntax:

`retC = VALTYPE (exp)`

Purpose:

Determines the data type of the given expression.

Arguments:

<**exp**> is an expression whose data type is to be determined. The expression can be a field, with or without the alias, a PRIVATE, PUBLIC, LOCAL or STATIC variable, or an expression of any type.

Returns:

<**retC**> is one of the following characters:

retC		Variable/Field/Expression	
A		Array, Sub-array	
B		Code Block	
C		Character	
D		Date	
E	*	short string, usually displayed as C	
F		Float num, when FS_SET("intv",.T.) is set	
I	*	Integer numeric, when FS_SET("intv") is set	
2	*	binary short int = +/-32767	(.dbf field)
4	*	binary long int = +/-2147483647	(.dbf field)
8	*	binary double float	(.dbf field)
L		Logical	
M		Memo	(.dbt field)
N		Numeric	
O		Object	
S		Screen variable	
U		Undefined, NIL, Local, Static	
UE		Syntax error	
UF		Undefined function error	
UI		Other errors	
VC	*	variable character size	(.dbv field)
VCZ	*	compressed var character size	(.dbv field)
VB	*	variable binary/blob data	(.dbv field)
VBZ	*	compressed var binary/blob data	(.dbv field)

(*) FlagShip specific

Description:

VALTYPE() determines the type of any variable, including PRIVATE, PUBLIC, FIELD, LOCAL, STATIC and TYPED variables. Also the result of a function execution may be evaluated.

Unlike TYPE(), which is evaluated by the expansion, the argument of VALTYPE() specifies directly the variable or function. If the argument specifies an unknown function, a link error (undefined external) will occur.

To check the existence of any public UDF or standard function without executing it, use ISFUNCTION().

References to arrays and sub-arrays return "A". References to array elements return the type of the element.

Example:

```
DECLARE info[5]
LOCAL var1 := {|| NIL}, var2
STATIC_INT var3
info[1] = .T.
info[2] = 12.01
info[3] = CTOD("06/07/93")
info[4] = "Jones"
info[5] = {1, 2, 3}

? TYPE    ("info")                  && A
? VALTYPE (info)                    && A
? VALTYPE (info[1])                 && L
? VALTYPE (info[2])                 && N
? VALTYPE (info[3])                 && D
? VALTYPE (info[4])                 && C
? VALTYPE (info[5])                 && A
? VALTYPE (info[5,2])               && N
? VALTYPE (SUBSTR(info[4],1,2))     && C (std function)
? VALTYPE (SOUNDEX(''))             && C (will be linked)
? VALTYPE (myudf(1, 2))             && L (returns logical)
? VALTYPE (help())                  && U (returns NIL)
? VALTYPE (var1)                    && B
? VALTYPE (var2)                    && U
? VALTYPE (var3)                    && N

? ISFUNCTION ("help")               && .T.
? ISFUNCTION ("SUBSTR")             && .T.
? ISFUNCTION ("MAXCOL")             && .F. (not linked)
? ISFUNCTION ("myudf")              && .T.

PROCEDURE help (p1, p2, p3)
RETURN
FUNCTION myudf (p1, p2, p3)
RETURN .T.
```

Classification: programming

Compatibility:

To check whether a PROCEDURE is available, use the alternative UDF syntax as in the example above. To check the UDF or UDP existence only, use ISFUNCTION() instead. The INTVAR type is available in FS only.

Related:

TYPE(), LOCAL..AS, FS_SET("intvar"), INT2NUM(), NUM2INT(), ISFUNCTION(), PCOUNT(), ISOBJPROPERTY()

VERSION ()

Syntax:

`retC = VERSION ( [expL] )`

Purpose:

Returns the FlagShip release used.

Arguments:

<**expL**> is optional logical value representing the required output. If <expL> is .T., the returned string includes the serial number of FlagShip. If the FS2 Toolbox is available, also the release and serial number of FS2 Tollbox library is returned.

Returns:

<**retC**> is the current release of the linked FlagShip library. The returned string will consist of these parts:

- `FlagShip` or another tool name
- `8.nn.` the main FlagShip release
  `nn` the current subrelease
- `[oper.system]` operating system
- addit. info (additional info, e.g. Eval expiration)
- `(xx users)` license used
- `s/n …xxxxxxxx` Serial number (only with <expL> = .T.)

Description:

VERSION() is a function that returns the version of the used FlagShip library (and FS2 Toolbox, if such is available). The alternative to display version of FS2 Toolbox (w/o serial number) is function ToolVer(), see manual section FS2 and example 3.

For other system information, refer also to OS(), NETNAME(), FS_SET(), SET().

The returned string is similar to the start-up information given by the command-line argument

```
$ a.out -FSversion          in Linux/Unix, or
C:> myexe.exe -FSversion    in MS-Windows.
```

Example 1:

```
? VERSION()

/* output either:
   FlagShip 8.01.13 [Linux 64-bit] (1024 users)
   FlagShip 8.01.18 [Linux 32-bit] (Personal, 2 users)
   FlagShip 8.01.24 [Windows 64-bit] (1024 users)
   FlagShip 8.01.15 [Windows 32-bit] Eval/DEMO version valid until \
                    14-Sep-2017 (1024 users)
*/

? VERSION(.T.)
```

```
/* output e.g.:
   FlagShip 8.01.21 [Windows 64-bit] (1024 users) s/n ...abcdefgh
   FS2 Toolbox 8.01.21 [Windows 64-bit] (1024 users) s/n ...xyz123ab
or
   FlagShip 8.01.21 [Linux64-bit] Eval/DEMO version valid \
       until 14-Sep-2017 (1024 users) s/n ...abcdefgh
   FS2 Toolbox 8.01.21 [Linux64-bit] Eval/DEMO version valid \
       until 14-Sep-2017 (1024 users) s/n ...xyz123ab
*/
```

Example 2: Don't display Eval/DEMO version if any

```
version := VERSION(.T.)
release := SUBSTR(version, 10, AT("]", version) -9)
? "The FlagShip release is " + release + ;
  " " + SUBSTR(version, AT("s/n",version), 15)
if "FS2 Toolbox" $ version
   version := SUBSTR(version, AT("FS2 Toolbox", version))
   release := SUBSTR(version, 1, AT("]", version))
   ? "  with " + release + " " + ;
     SUBSTR(version, AT("s/n",version), 15)
endif
? "Oper.System: " + OS()
? "You are: " + NETNAME()


// The FlagShip release is 8.01.01 [Linux 64-bit] s/n ...khiyglto
//   with FS2 Toolbox 8.01.01 [Linux 64-bit] s/n ...fmhygiyg
// Oper.System: Linux 32bit & 64bit  Release 3.16.7-53-desktop \
//     #1 SMP PREEMPT Fri Dec 2 13:19:28 UTC 2017 (7b4a1f9)
// You are: hugo@HUGO_PC5
```

Example 3: similar to example 2, using FS2 function ToolVer()

```
version := VERSION()
release := SUBSTR(version, 10, AT("]", version) -9)
? "The current FlagShip release is " + release
#ifdef VFS_FS2TOOLS           /* is FS2 Toolbox available? */
  ? "and the additional ", ;
     SUBSTR(ToolVer(), 1, AT("]", ToolVer()))  // see manual FS2
#endif

// The current FlagShip release is 8.01.07 [Windows 64-bit]
// and the additional  FS2 Toolbox 8.01.07 [Windows 64-bit]
```

Classification: programming

Compatibility: The returned string differs from the Clipper's. The optional parameter is available in VFS8 and newer.

Related: OS(), NETNAME(), FS_SET(), SET()

WARNBOX ()

Syntax:

`retN = WarnBox (expC1, [expC2], [expN3])`

Purpose:

Display an warning message box dialog, similar to Alert()

Arguments:

<**expC1**> is a string containing the displayed text, the lines are separated by semicolon ";". See additional details in Alert() description.

<**expC2**> is an optional header text.

<**expN3**> is optional time-out in seconds. Zero or NIL specify to wait until user press key or mouse selection. When the time-out expires without user action, the box is closed and <retN> is set to 0.

Description:

The WarnBox() function is available for your convenience. It is equivalent to WarningBox{NIL,cHeader,cMessage}:Show() described in section OBJ. In GUI mode, warning icon (yellow triangle with exclamation mark) is displayed. In Terminal i/o mode, the color is specified in global variable

```
_aGlobSetting[GSET_T_C_WARBOXCOLOR] := "GR+/B,B/W"  // default
```

and can be re-assigned according to your needs. You also may set border via e.g.

```
_aGlobSetting[GSET_T_C_ALERTBOX] := B_SINGLE // default = B_PLAIN
_aGlobSetting[GSET_T_C_ALERTBOX] := B_DOUBLE // default = B_PLAIN
```

Example:

```
_aGlobSetting[GSET_T_C_WARBOXCOLOR] := "GR+/B,B/W" // default
_aGlobSetting[GSET_T_C_ALERTBOX   ] := B_SINGLE   // terminal i/o

WarnBox (";Warning: inproper input data", "Caution", 10)
```

Compatibility:

New in VFS5. The expN3 is supported since VFS7

Related:

Alert(), InfoBox(), ErrBox(), TextBox(), MessageBox{}, InfoBox{}, ErrorBox{}, WarnBox{}, TextBox{}

WARNINGBOX ()

Syntax:

retO = WarningBoxNew ([oOwner], [cTitle], [cText])

Syntax:

retO = WarningBox ([oOwner], [cTitle], [cText])

Description:

This is alternative syntax for creator of WarningBox class, see details in section OBJ

Related:

WarningBox Class, WarnBox()

WebDate ()

Syntax:

retC = WebDate ([expD1], [expCL2])

Purpose:

Forms the given or current date and time to a string common in e- mails and web pages.

Options:

<**expD1**> is the date value. If not given, current DATE() is used.

<**expCL2**> is the time value. If passed as non-empty string, the value is added to the resulting value. If passed as an empty-string or logical .T., the current TIME() is used. Otherwise ignored.

Returns:

<**retC**> is the resulting string in a form "01-Jan-2016" or "01-Jan- 2016 15:46:25" depending on <expCL2>.

Description:

WebDate() converts the date (and time) value to a normalized string, independent of the current SET DATE setting.

Example:

```
? WebDate()                      // 10-Aug-2011
? WebDate(, .T.)                 // 10-Aug-2011 15:14:56
? WebDate(date()-20, "hello")    // 21-Jul-2011 hello
```

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

DATE(), TIME()

WebErrorHandler ()

Syntax:

```
NIL = WebErrorHandler (expO1, expC2, [exp3],
           [expA4], [expB5])
```

Purpose:

Installs new error handler designed for use in CGI invoked applications on the Web server.

Arguments:

<**expO1**> is the error object, passed e.g. via ErrorBlock(), see example.

<**expC2**> is the name of error log file, including path if required, where the error messages are appended.

Options:

<**expC3**> optional string with an e-mail address where a notification about errors should be send. If not given, no error e-mails are sent.

<**expA4**> optional array with e-mail header data, see WebSendMail(). Applicable only if <expC3> is given.

<**expB5**> optional code block returning a string with additional data to be stored in the error log.

Returns:

There is no typical return value. If the handler detect a currently active BEGIN SEQUENCE, it performs a BREAK to jump into the RECOVER section of the SEQUENCE, passing the current <expO1> value. If no BEGIN SEQUENCE is active, NIL is returned after storing the error data and optionally sending the e-mail.

Description:

WebErrorHandler() avoids displaying of errors on the screen or in the produced HTML output. It stores the messages in an error file and optionally sends an e-mail to e.g. the Web administrator.

Example:

```
bOldErrBlock := ErrorBlock({|oErr| WebErrorHandler(oErr, ;
     "/cgi-bin/err.log", "webmaster@mydomain.com") })
```

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

ErrorBlock(), WebLogErr(), WebSendMail(), BEGIN SEQUENCE

WebGetEnvir ()

Syntax:

`retA = WebGetEnvir ()`

Purpose:

Fills the commonly used Web-server environment variables into an array.

Returns:

<**retA**> is a two-dimensional array containing each the name of the environment variable and its value.

Description:

The WebGetEnvir() function check the current system environment for variables (see the webutil.prg source for details) and if found, adds the name and its value to the resulting array.

Example:

```
aVars := WebGetEnvir()
aeval(aVars, {|x| qout(x[1],"= [" + x[2] + "]") } )
```

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

GetEnv()

WebGetFormData ()

Syntax:

retA = WebGetFormData ([expBC1], [expL2])

Purpose:

This is usually a central function for CGI invoked application, receiving the FORM data from the HTML page.

Options:

<**expBC1**> optional code block or a name of a UDF which handles errors. There are two parameters passed to the UDF: the current value of REQUEST_METHOD and the text message.

<**expL2**> if specified .T., it translates the first character of each field value to upper case.

Returns:

<**retA**> is a two-dimensional array containing each the form identifier (name) and the corresponding value.

Description:

WebGetFormData() determines the Form Request Method (GET or POST) and stores the form names + values in the resulting array.

Example:

Simple application invoked from HTML via CGI in the FORM section. It returns a page containing the FORM-Tag fields and the current environment.

```
ErrorBlock({|oErr| WebErrorHandler(oErr, ;
     "/cgi-bin/err.log", "webmaster@mydomain.com") })

private cWebHeaderTitle := "Back-response"
local   aEnv  := WebGetEnvir()
local   aData := WebGetFormData()

WebHtmlBegin(NIL,"FFFFFF")
? "<P>Received data:<BR>"
WebOutData(aData)
if len(aEnv) > 0
   ? "<P>Current environment:<BR>"
   aeval(aEnv, {|x| qout(x[1], " = ", x[2], "<BR>") })
   ? "<P>"
endif
WebHtmlEnd()
quit

FUNCTION CursesInit()       // see SYS.2.7
RETURN NIL
FUNCTION ProgramInit()   // see SYS.2.9
  CALL fgsUse4html
RETURN NIL
```

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

WebHtmlBegin(), WebHtmlEnd(), WebErrorHandler(), WebGetEnvir(), WebOutData(), FREADSTDIN(), SYS.2.7, SYS.2.9, fs4web.html in <FlagShip_dir>/tools/fs4web/

WebHtmlBegin ()

Syntax:

`NIL = WebHtmlBegin ([expBC1], [expC2])`

Purpose:

Prepare a header for HTML output.

Options:

<**expBC1**> optional code block (returning a string) or a string which should be printed at the begin of the HTML text, i.e. just after the <BODY> tag.

<**expC2**> optional background color in the syntax "#RRGGBB" or "RRGGBB", where "#FFFFFF" is white and "#000000" is black.

`cWebHeaderTitle` a PRIVATE or PUBLIC string variable describing the HTML <TITLE> tag. If not specified, a PUBLIC cWebHeaderTitle := "HTML Request" will be created automatically.

`lWebHeaderOut` a PRIVATE or PUBLIC logical variable containing .F. as a start value. If not specified, a PUBLIC `lWebHeaderOut` will be created automatically. At the first invocation of WebHtmlBegin(), this variable is set .T. so the next invocations of this function do not generate the header again.

Description:

Every HTML output needs a correct syntax containing the header tags, optional text and the bottom tags. The HTML page header tags are generated by this function.

Example:

see example in WebGetFormData()

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

WebHtmlEnd(), fs4web.html in <FlagShip_dir>/tools/fs4web/

WebHtmlEnd ()

Syntax:

NIL = WebHtmlEnd ([expBC1])

Purpose:

Prepare the bottom tags for a HTML output.

Options:

<**expBC1**> optional code block (returning a string) or a string which should be printed just before the bottom tags of the HTML page.

Description:

Every HTML output needs a correct syntax containing the header tags, optional text and the bottom tags. The HTML page bottom tags are generated by this function.

Example:

see example in WebGetFormData()

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

WebHtmlBegin(), fs4web.html in <FlagShip_dir>c/tools/fs4web/

WebLogErr ()

Syntax:

NIL = WebLogErr (expC1, expC2, [expC3], [expA4])

Purpose:

Writes any given message to the error log file.

Arguments:

<**expC1**> is the message text written to error-log file.

<**expC2**> the name of error log file, including path if required. It should correspond to <expC2> of WebErrorHandler() if such is used.

Options:

<**expC3**> optional string with an e-mail address where a notification about errors should be send. If not given, no error e-mails are sent. It may correspond to <expC3> of WebErrorHandler() if such is used.

<**expA4**> optional array with e-mail header data, see WebSendMail(). Applicable only if <expC3> is given. It may correspond to <expA4> of WebErrorHandler() if such is used.

Description:

This function is invoked by WebErrorHandler(), but may also be called explicitly, e.g. when an error is detected during the data validation etc.

Example:

see example in WebErrorHandler()

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

WebErrorHandler()

WebMailDomain ()

Syntax:

retC = WebMailDomain (expC1)

Purpose:

Checks and extracts the e-mail address from the given string.

Arguments:

<**expC1**> is the source string from which the e-mail address should be extracted.

Returns:

<**retC**> is the extracted e-mail address from <expC1> or one space " " == empty(retC) on error.

Description:

E-mail addresses needs to be normalized according to URL rules (name@[subhosts.]host.domain) for a correct addressing by the mail server. This function extracts the plain URL address (w/o prefix and postfix) from the given string. It checks the given syntax, but does not (and cannot) verify the availability of this domain/URL (you may try "ping" here), nor real availability of the addressee.

Example:

```
? '"aaa bbb" <xxx@yyy.zzz> qqq'   // xxx@yyy.zzz
? 'aaa bbb xxx@yyy.zzz'           // xxx@yyy.zzz
? 'aaa bbb xxx @ yyy.zzz'         // " " = error
```

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

WebSendMail()

WebOutData ()

Syntax:

NIL = WebOutData ([expA1])

Purpose:

This is a helper which prints the received form-tag fields to the HTML output.

Options:

<**expA1**> is an array containing the results from e.g. WebGetFormData() or WebGetEnvir(). If not given, the WebGetFormData() is invoked automatically.

Description:

You may use this function e.g. for testing your CGI based application. For a correct functionality, WebHtmlBegin() should already be invoked.

Example:

see example in WebGetFormData()

Source:

Available in the <FlagShip_dir>/tools/webtools/webutil.prg file.

Compatibility:

Available in FlagShip only.

Related:

WebGetFormData(), WebGetEnvir(), WebHtmlBegin(), fs4web.html in <FlagShip_dir>/tools/fs4web/

WebSendMail ()

Syntax:

```
retN = WebSendMail (expCA1, expC2, expC3, expC4,
                    [expC5], [expC6], [expC7])
```

Purpose:

Prepares and sends an e-mail (electronic mail) to the given addressee via system's "sendmail" daemon.

Arguments:

<**expCA1**> e-mail header: either a string with the sender e-mail address, or an array containing e-mail header data

```
{"sender address",
 "sender organization/company",
 "send reply to",
 "return path"}
```

where the not used array elements may be set NIL or may contain an empty string.

<**expC2**> a string for the e-mail subject.

<**expC3**> the e-mail body (text), sent "as given". At least one character is required. You may format the text by new-line LF = chr(10) or CRLF = chr(13)+chr(10) for the e-mail line width (usually 75-78 chars) but this is not strictly required.

<**expC4**> the e-mail addressee(s). Since the string is not validated here, it may be advised to use WebMailDomain() for a check, if the string contains a correct e-mail URL. A usual syntax for the e-mail address is e.g.:

```
any_name@my.domain.com
"any name" <myname@mydomain.com>
<name1@server1.com>, "other name" <name2@server2.com>
```

The URL address in case insensitive.

Options:

<**expC5**> optional string with CC (carbon copy) e-mail address(es).

<**expC6**> optional string with BCC (blind carbon copy, invisible for the recipient) e-mail address(es).

<**expC7**> optional string specifying the sendmail executable incl. path and optional switches, e.g. "/usr/lib/sendmail -t" or "C:\myProg\sendmail.exe" (see below). If not given, the default is "/usr/lib/sendmail -t" or "/usr/lib/sendmail -t -f xxx@yyy.zzz" where xxx@yyy.zzz is taken from expCA1[4] if any.

Returns:

<**retN**> is the success or error return code:

0	success
1..6	check for argument no. 1..6 failed
7	not enough arguments passed
10	command-line of the header is too long, check e.g. <expC4>

11 could not open (invoke) sendmail as sub-process. Check if sendmail is in /usr/lib, otherwise create a (symbolic) link or modify the #define and recompile webmail.c according to instruction given in the header of it, or use <expC7>.

Description:

This function can be used to send e-mails from any FlagShip application, either stand-alone or web based. It invokes the "sendmail" program or daemon on the computer where the application is running. The used sendmail is default for all Unix and Linux versions. For MS-Windows, you may need sendmail.exe available with Microsoft Mail (PC Networks Resource Kit), or the free package from http://glob.com.au/sendmail or it copy via http://www.fship.com/tools/sendmail.zip or via ftp://fship.com/pub/multisoft/FlagShip/tools/sendmail.zip or the commercial sendmail version from http://www.indigostar.com/sendmail.htm

Example 1:

```
#define CRLF chr(13,10)
cText := "Dear Jim," + CRLF + ;
         "this is my first try to send e-mail" + CRLF + ;
         "via FlagShip..." + CRLF + CRLF + "Paul" + CRLF
ok := WebSendMail('"Paul Smith" <pauls@mydomain.org>', ;
         "First e-mail to Jim", cText, "jimx@aol.com")
if (ok > 0)
   ? "mail not sent, error",ltrim(ok),"from sendmail"
endif
```

Example 2:

You may add additional MIME tags (like HTML formatting etc.) in the subject parameter, separated from it (and by each other) with chr(10):

```
cSubject := "A subject of this mail" +  ;
   chr(10) + 'Content-Type: text/html; charset="iso-8859-1"' + ;
   chr(10) + 'Content-Transfer-Encoding: quoted-printable'

ok := WebSendMail(aYourData, cSubject, cText, ...)
```

Source:

Available in the <FlagShip_dir>/tools/webtools/webmail.c file.

Compatibility:

Available in FlagShip only.

Related:

WebMailDomain()

WORD ()

Syntax:

Cint = WORD (expN1)

Purpose:

Converts the CALL command numeric argument from double to a C int value.

Arguments:

<**expN1**> is a FlagShip numeric value to convert to (long) int which usually has a range of +/- 2,147,483,647.

Returns:

The function returns a C integer. Used in the CALL command, the FlagShip compiler generates a temporary (int) variable which is passed by value to the CALLed function.

Description:

WORD() is a numeric conversion function that is designed for parameter conversion within a CALL command. Refer to the CALL command for more information.

Using the FlagShip Extend C System is the safer way to execute a C function because of parameter checking and passing.

Example:

```
LOCAL var1 := 5.56, var2 := 10
CALL myCudf WITH WORD(12345), var1, WORD(var2)
? var1                          // 987.65 changed by myCudf
? var2                          //  10.00 remains unchanged
```

Classification:

programming

Compatibility:

On DOS systems, the int value is two-byte long in the range of +/- 32,767. On most UNIX and 32/64bit Windows systems, the C int value is equivalent to a LONG int, stored in four bytes.

Related:

CALL

YEAR ()

Syntax:

`retN = YEAR ( [expD] )`

Purpose:

Extracts the year from a date value.

Arguments:

<**expD**> is the date value to convert. If not specified, the current system date() is used.

Returns:

<**retN**> is a numeric value representing the year including the century digits. The value returned is not affected by the current SET DATE or SET CENTURY format. Specifying an invalid or a null date CTOD("") returns zero.

Description:

YEAR() is a date conversion function that converts a date value to a numeric year value. It is used when the year is needed as a part of some calculations.

Other date extracting functions are DAY() and MONTH().

Example:

```
? DATE()                                    // 07/25/93
? YEAR(DATE())                              // 1993
? YEAR()                                    // 1993
d := DATE() +1
? CMONTH(d), LTRIM(STR(DAY(d))) + ",", ;
  LTRIM(STR(YEAR(d)))                       // July 26, 1993

SET DATE GERMAN
? DATE()                                    // 25.07.93
? YEAR(DATE())                              // 1993
? YEAR() + 12                               // 2005
```

Classification:

programming

Compatibility:

Clipper requires the <expD> value, it does not use current date() with empty parameter.

Related:

DATE(), DAY(), MONTH(), CTOD(), DOW(), CDOW(), CMONTH(), DTOC(), DTOS(), FS2:DoY(), FS2:BoY(), FS2:EoY(), FS2:IsLeap(), FS2:Week(), FS2:Quarter(), FS2:MDY()

_DISPLARR ()
_DISPLARRSTD ()
_DISPLARRERR ()

Syntax 1:

```
retA = _DisplArr (expA1, [expC2], [expN3], [expN4],
                  [expL5])
```

Syntax 2:

```
retA = _DisplArrStd (expA1, [expC2], [expN3],
                  [expN4], [expL5])
```

Syntax 3:

```
retA = _DisplArrErr (expA1, [expC2], [expN3],
                  [expN4], [expL5])
```

Purpose:

Formats an one- or multi-dimensional array for displaying

Arguments:

<**expA1**> is the array to be formatted.

<**expC2**> is an optional text prefix describing the output.

<**expN3**> start element of <expA1>, default is 1

<**expN4**> last element of <expA1>, default is LEN(expA1)

<**expL5**> display flag for object elements:

.T. display object header of sub-objects only,
.F. (default) display all sub-object properties

Returns:

<**retA**> is 1-dimensional array containing re-formatted array <expA1>. All elements of <retA> are converted to character.

Description:

The function is mainly used for testing and debugging purposes.

Syntax 1 return the formatted array

Syntax 2 display the formatted array to standard output, same as

```
AEVAL( _DisplArr(expA1,expC2), {|x| Qout(x)} )
```

Syntax 3 display the formatted array to stderr, same as

```
AEVAL( _DisplArr(expA1,expC2), {|x| outerr(x,chr(10)) } )
```

Classification:

programming

Compatibility:

New in FS5

Related:

Aeval(), _DisplObj()

_DISPLOBJ ()
_DISPLOBJSTD ()
_DISPLOBJERR ()

Syntax 1:

```
retA = _DisplObj (expO1, [expC2], [expN3], [expN4],
                  [expL5])
```

Syntax 2:

```
retA = _DisplObjStd (expO1, [expC2], [expN3],
                     [expN4], [expL5])
```

Syntax 3:

```
retA = _DisplObjErr (expO1, [expC2], [expN3],
                     [expN4], [expL5])
```

Purpose:

Returns or displays object properties

Arguments:

<**expO1**> is the object to be traced.

<**expC2**> is an optional text prefix describing the output.

<**expN3**> requested properties to output, add to each other:

0:	CLASS ... only
1:	exported instance
2:	usual/protect instance (with -d switch)
4:	access
8:	assign
16:	methods
255:	- all above, default for _DisplObjStd(), _DisplObjErr()
256:	list also properties of super class
511:	- all above
512:	use instance instead of access, if possible
1023:	- all above, default for _DisplObj()

<expN4> sort and display format flag:

0:	(default) sort, filter, transfer/format to 1-dimens array
1:	display multi-dimensional, no formatted, no filtered, no sorted
2:	display multi-dimensional, no formatted, no filtered, sorted
3:	display multi-dimensional, no formatted, filtered, sorted
>3:	same as 0
<0:	same as positive, but print also debug info

<**expL5**> display flag:

.T. display object header of sub-objects only,
.F. (default) display all sub-object properties

Returns:

<**retA**> is 1-dimensional array containing properties of <expO1> and re-formatted to string.

Description:

The function is mainly used for testing and debugging purposes.

Syntax 1 return the object properties in an array

Syntax 2 display the formatted object properties to standard output, same as AEVAL(_DisplObj(expA1,expC2), {|x| Qout(x)})

Syntax 3 display the formatted object properties to stderr, same as AEVAL(_DisplObj(expA1,expC2), {|x| outerr(x,chr(10)) })

Classification:

programming

Example:

see <FlagShip_dir>/examples/printer.prg

Compatibility:

New in FS5

Related:

Aeval(), _DisplArr()

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Notes

multisoft Datentechnik
Schönaustr. 7
D-84036 Landshut

http://www.fship.com
sales@multisoft.de
support@flagship.de