FlagShip

Object Oriented Database Development System

Cross-Compatible to Unix, Linux and MS-Windows

MULTISOFT

Release 8.1

Section CMD

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 CMD

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

Trademarks

Headquarter Address

multisoft Datentechnik
Schönaustr. 7
84036 Landshut
Germany

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

Phone: (+49) 0871-3300237

Web: http://www.fship.com

CMD: FlagShip Commands

FlagShip Commands

Notation Used

The syntax of the FlagShip commands is the same as in other xBase languages, such as dBASE or Clipper. The following notation is used throughout this manual:

COMMAND [arguments] [KEYWORD [arguments]]

COMMAND

One or more special keywords (or symbols) at the beginning of a source line (leading spaces and tabs are not significant) define the commands, such as RUN, ? APPEND etc. The command keywords are case insensitive and may be shortened to 4 characters, so APPEND, APPEN and APPE represent the same command keyword, but APPEX will produce a compile-time error.

KEYWORD

The keyword (or clause) modifies the command to perform and satisfy additional special actions and requirements. The keywords are also case insensitive and may be shortened to 4 characters.

<argument>

Some commands and keywords require additional specification (arguments). The syntax used for the arguments is always "exp?" where "?" is the type of the expression e.g. "expC" for character, "expN" for numeric and so on. This means, that the argument may be entered as a constant, variable or any expression of the required type. If the type is not given, any type is allowed. The usual syntax is KEYWORD constant or KEYWORD "constant" or KEYWORD &macro. or KEYWORD (expression), see details in each command syntax. Note that the parenthesis ( ) does not specify here the priority of the evaluation like a mathematical parentheses, but tells the compiler: "use/calculate an expression instead of constant". So the arguments "abc.efg" and (xyz + ".efg") are valid (constant vs. expression), but (xyz)+".efg" is an invalid argument syntax, although it is a valid expression in all other context.

<item>

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

item1|item2

If more than one kind of syntax is allowed, the different syntax keywords or options are separated with the **|** sign. The items are mutually exclusive, you may use only one of them. Do not type the **|** sign.

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.

(item)

The parentheses are part of the syntax and must be entered.

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).

varS

Variable of type screen (see LNG.2.6).

expList, argList, fieldList

List of expressions (or arguments, fields etc.) in the syntax exp1 [,exp2 [, exp3 ...]]. If two or more expressions (or arguments, fields etc.) are specified, a comma is used as a separator between each of the single expressions <exp>; see also LNG.2.8.

on|OFF|(<expL>)

The ON or OFF switch (flag) activates or deactivates the command and is specified as a literal (meaning the letters "on" or "off"). Alternatively, the parenthesized <expL> (logical expression or constant) can be used, whereby logically TRUE is the same as ON. The default switch is given in capital letters.

<scope>

In some database commands, partial execution can be specified. The valid <scope> arguments are: ALL (all database records), NEXT <expN> (next n records), REST (from the current record to the end of the database), RECORD <expN> (the given record number). Additional filters are available using FOR and WHILE clauses.

...FOR <condition> ...WHILE <condition>

In some database commands, the FOR clause specifies that the command will be repeatedly executed for all records meeting the logical expression given as <condition>. The WHILE clause stops the repetition of the command when the first record which does not meet the condition is reached. The <scope> option, if given, restricts the FOR and WHILE clause.

...TO PRINTER

This clause echoes the output of the console command (per default ADDITIVE) to a printer file or to the device set by the SET PRINTER TO command. The ..TO PRINTER clause is equivalent to automatically echoing output to a printer file or device, already activated by the SET PRINTER ON command. If the SET PRINTER TO <file> (or device) was not specified, the output is redirected to the FlagShip's standard spooler file, see LNG.3.4 and LNG.5.1.6.

...TO FILE <file>

This clause echoes the output of the console command to the specified ASCII file. If the file extension is not specified, .txt is assumed. If the additional ADDITIVE option is given, an addition in made to the output instead of the <file> being overwritten. The TO FILE.. ..ADDITIVE clause is equivalent to automatically echoing output to a SET EXTRA file or device which has been already opened and activated by ON. Additional redirections of the sequential (console) output are available using the SET PRINTER ON and SET ALTERNATE ON/TO commands.

Syntax:

The required syntax, keywords and arguments of the command.

Arguments/Options:

Explanation of the required or optional command modifiers or entries.

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 command usage possibilities, in a program context.

Classification:

Classification of the command, e.g. input, output, database etc..

Compatibility:

The commands, keywords and arguments have the same syntax as in other xBASE dialects, like 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.

Translation:

Most commands will be translated by the FlagShip preprocessor to equivalent functions, according to the file *<FlagShip_dir>/include/std.fh*. The actual translation may differ, and is given for your orientation only. The std.fh file and the internal, undocumented functions (where the name starts with an underscore) may be changed without prior notice.

Related:

Equivalent, related or similar commands and functions.

PROCEDURE example

Typography used for program examples or command usage.

$ input

Typography used for user input from the Unix or Windows shell.

<FlagShip_dir>

The *<FlagShip_dir>* is usually the directory */usr/local/FlagShip8* in Unix and Linux, or *C:\Program Files\FlagShip8* in MS-Windows, but may differ according to your setup choice and MS-Windows defaults. The real path is displayed by "**FlagShip -v**" or "**FlagShip -h**".

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

The FlagShip preprocessor translates standard commands to their equivalent functions according to the definitions in the **std.fh** include file (see translation above). FlagShip also supports user-defined-commands (UDC), which are translated via the #command or #xcommand preprocessor directive to other functions or commands. See more in section PRE.

The commands that follow are listed in alphabetical order and may be used as the language reference. For a summary of the commands, see sections QRF and LNG.

! | RUN

Syntax:

```
! [WAIT|NOWAIT]
        [MESSAGE <expC1>]
        <Unix command|Windows command>|(<expC2>)
```

or:

```
RUN [WAIT|NOWAIT]
        [MESSAGE <expC1>]
        <Unix command|Windows command>|(<expC2>)
```

Purpose:

Executes a Unix or MS-Windows command, program or script within the actual application. This enables harnessing the power of Unix or Windows commands.

Arguments:

<**Unix command**> or <**Windows command**> may be any executable program or shell/batch script file, optionally with path. All character expressions must be enclosed in parentheses. Macro expressions can also be used and will be expanded before submitting the command to the shell.

Options:

WAIT or **NOWAIT**: optional modifier. With WAIT (default), the application will wait until the command will finish. NOWAIT will trigger the command to background and continue execution of the application. NOWAIT is similar to Unix command "shell_call &". Do not use WAIT/NOWAIT clause together with the "&" postfix.

MESSAGE <**expC1**> is an optional, user defined message to be printed on the screen, when the executed Unix command is finished. Note, no FlagShip output mapping is active when the MESSAGE is printed; it works as does the "echo <expC1>" from the Unix shell would. Before <expC1> is printed, a NEW LINE is executed (similar to the WAIT command).

Note that both options, if any given, needs to precede the command.

Return code:

The return code may be checked via DosError() function. Note: this return code is system dependant and correspond to the return value of system function system() or of errno if system() returns -1. On some oper. systems, you will get the true exit code by calculating nRet := int(DosError() / 256). You may display the clean error msg by Doserror2str()

Description:

At RUN command, FlagShip invokes a new shell and passes it the Unix or Windows command to be executed. The required command must be available in the current path or else given with an absolute path.

When the <Unix/Windows command> ends (or when the background process is started by "&" postfix or by NOWAIT clause), the control returns back to the application, executing the next FlagShip statement.

In MS-Windows, the ! or RUN command works by the same way as in Unix. See further details in CMD.RUN description.

To enable the inspection of the output from the called program, print a prompt (using e.g. the MESSAGE clause or the equivalent statement "; echo...") and stop the further execution using INKEY(0) after the RUN command; see example on the RUN command.

Shell access: You may run a shell by specifying the argument "sh" (or "csh", "ksh" respectively) to the RUN command. To exit the shell, type "exit". In MS-Windows, invoke "CMD" or COMMAND for that reason.

Background processing: the executable or script called may run in background, if the RUN command specification ends with an ampersand (&) character or by using the NOWAIT clause. The current application will not wait for the called executable to finish, but will carry on with its own execution immediately. The program called becomes a child of the calling executable and will terminate latest when the current application terminates. Applicable in Unix/Linux only. Note that any input to, or output from the background program may cause the called application to hang.

User break: when the called program is a FlagShip application, both programs will receive the break and debug signals (^K and ^O).

Screen output: In Terminal i/o, output from the called application goes to the application screen, and may garbage it. In GUI mode, the output goes to stdout or stderr, which is usually assigned to the console (or console window), and hence does not affect the current screen. See more in (CMD) RUN.

Compatibility note: since the Unix and MS-Windows commands usually differs from each other, you may use

```
#ifdef FS_WIN32
   RUN Windows-Command...
#else
   RUN Unix-Command...
#endif
```

Example 1:

This example shows how to use RUN in combination with MEMOREAD() and MEMOWRIT() to create a user-defined function that calls the editor with the current memo field:

```
PUBLIC FlagShip, Clipper
editor = if (FlagShip, "vi", "edlin")
success = MemoEditor (editor, "Notes")

FUNCTION MemoEditor (editor, memofld)
IF MEMOWRIT ("myedit.txt", &memofld)
   RUN (editor + " myedit.txt")
```

```
    REPLACE &memofld WITH MEMOREAD ("myedit.txt")
    RETURN 0                             // success
 ELSE
    RETURN -1                            // error
 ENDIF
```

Example 2:

Start MS-Word (Winword) in Windows as sub-process, continue processing of the application. Note the notification of path and/or file name including spaces: the executable (with path) and/or the file name needs to be passed to Windows enclosed in double quotas. When the command uses variables, enclose it in parentheses.

```
 ? "Invoking MS-Word as separate process..."
 RUN NOWAIT '"C:\Programs\Microsoft Office\Office\Winword.exe" /w'
 WAIT "press any key to continue this application..."

 // or:
 cDocFile := '"D:\Documens and Settings\Default User\' + ;
             'My Documents\myfile.doc"'
 cCommand := '"C:\Program Files (x86)\Microsoft Office\root\' + ;
             'Office16\WINWORD.EXE"'
RUN NOWAIT (cCommand + " " + cDocFile)
```

Example 3:

See additional examples in the RUN command.

Classification:

system call

Compatibility:

As opposed to the equivalent DOS execution, there are practically no limits to the use of RUN on Unix or Windows. If the available RAM space is insufficient, the additional swap disk area (Linux) or pagefile (Windows) will be used automatically.

Keep in mind the differences in system command names on DOS and Unix (ls instead of DIR etc.) and the different DOS vs. Unix screen handling. For portability, #ifdef FlagShip... #else...#endif or the PUBLIC FLAGSHIP variable can be used to compile platform specific code selectively.

The MESSAGE clause is new in FS4, WAIT/NOWAIT in FS6 and both are not available in Clipper.

Translation:

__RUN (expC)

Related:

RUN, REFRESH

* && // /*...*/ NOTE

Syntax:

```
NOTE [<text>]
```

or:

```
* [<text>]
```

or:

```
[<command>] && [<text>]
```

or:

```
[<command>] // [<text>]
```

or:

```
[<command>] /* [<text>] */ [<expression>]
```

Purpose:

Various kind of program comments: full-line, in-line and special comments.

Arguments:

<text> is a character string ending with a new line.

Description:

NOTE and * at beginning of the source line (leading spaces and TABs are not significant) marks the whole line as a (full-line) comment.

A double ampersand (&&) or double slashes (//) can be placed after the command, if there is one on the same line, the text followed && or // is a user comment, not evaluated by the compiler. Slash + star (/*) marks all following text as comment until star + slash (*/) is detected. This comment can continue over new lines and is accepted within an expression.

If you need to continue command on next line, place the semicolon before the // or && mark.

Example:

```
*************
*  Comment  *
*************
a = b                                  && Inline comment,
a = b + ;                              && usable also for
    c + d                              && continued statement
NOTE That is an comment line,
NOTE   same as these
*      or these line.
*      The &macro will be not evaluated
//     and commands (e.g. @ 5,1 CLEAR) not executed.

REPLACE name WITH var_name,    ;       // Inline-
        zip  WITH VAL(zip_var)         // comment
USE address     /* here starts a
                   special comment, continued
                   on several lines */
USE /* means open a database address.dbf: */ address

/*  The SELECT command will be executed: */   SELECT 5
//  The SELECT command will not be executed:  SELECT 5
&&  The SELECT command will not be executed:  SELECT 5
*   The SELECT command will not be executed:  SELECT 5

/* this comment
   is continued
   over several lines
*/
value =  am /* amount */ + tx /* plus tax */
```

Classification:

programming

Related:

#comment, #nocomment

? | ??

Syntax:

```
? [<expList>]
?? [<expList>]
```

Syntax:

```
? [SPLIT | COLUMN [<expN5>,<expN6>]]
        [COLOR <expC1>]
        [GUICOLOR <expC2>]
        [PRINTCOLOR <expC3>]
        [FONT <expO4> | FontNew(...)]
        <expList>
```

Syntax:

```
?? [SPLIT | COLUMN [<expN5>,<expN6>]]
        [COLOR <expC1>]
        [GUICOLOR <expC2>]
        [PRINTCOLOR <expC3>]
        [FONT <expO4> | FontNew(...)]
        <expList>
```

Purpose:

Evaluates and displays the results of one or more expressions to the console or to GUI printer.

Arguments:

<**expList**> is a list of values or expressions to be evaluated and displayed. If there are more than one expression, the expressions must be separated by commas. The expressions can be of any data type, including memos.

If no <expList> argument is specified in the ? command, a NEW LINE code is sent to the console. If the ?? command is used without <expList>, nothing happens.

Options:

SPLIT will split long string into two or more lines. The available size is calculated from current Col() position up to MaxCol() for current line and MaxCol() -1 for subsequent lines. If PrintGui(.T.) is active or SET GUIPRINT is ON, oPrinter:GuiMaxCol() is used instead. The string is splitted at the left next space or tab or dash if any. You may add conditional split position (separators) by chr(1) or chr(247), which are then interpreted as dash at line end and ignored otherwise.

COLUMN <expN5>,<expN6> or **SPLIT <expN5>,<expN6>** is similar to SPLIT, but instead of full line, it will split the large text column-wise, from column <expN5> to <expN6> (in row/cols). Note that <expList> may contain only single character string or expression. See example in <FlagShip_dir>/examples/printergui.prg

COLOR <expC1> specifies the color for displaying the <expList> data. Only the first color pair (standard) is significant. If this clause is not given, the current color setting is used. In GUI mode, first the GUICOLOR clause is checked. If not set, the COLOR <expC1> 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 COLOR and SET GUICOLOR setting.

GUICOLOR <expC2> specifies the color for displaying the <expList> data considered in GUI mode. Only the first color pair (standard) is significant. Instead of string, you also may use RGB triplets (or stringified triplets), see SET COLOR for details, and example below. If GUICOLOR is set, this color is used in GUI mode regardless the current SET GUICOLOR on/off. If omitted and SET GUICOLOR is ON, either the COLOR <expC1> is used if given, or the current SetColor() is used. The GUICOLOR clause apply for GUI mode only, and is ignored otherwise.

PRINTCOLOR <expC3> specifies the color for printing. If not given, GUICOLOR is used also for printer. Considered only in GUI mode when SET GUIPRINT is ON or with PrintGui(.T.), and ignored otherwise.

FONT <expO4> is a font specification, considered only for screen and/or SET GUIPRINT output in GUI mode and ignored otherwise. The <expO3> is already instantiated font object, which allows you to set the font/family name, size and additional attributes like bold, underscore, italic and so on, independent on the current SET FONT setting. Alternatively, instead of <expO4>, you may instantiate font directly, by specifying e.g. FONT FontNew("courier",12,"BI"). Note that the Col() is adapted automatically to a larger/smaller font size but the Row() only when SET ROWADAPT is ON (default is OFF). You may force the adaption manually by invoking RowAdapt().

Description:

The displayed results of the expressions are separated by a space character. The **? command** outputs a linefeed (the NEW LINE code) before displaying the expressions.

The **?? command** omits the linefeed and thus allows you to display multiple expressions on one line continuing the previous output at the current screen or printhead position.

FlagShip supports echoing of console commands (see LNG.5.1.1) to four different devices/files at a time: to the default SCREEN device, and additionally to the PRINTER, ALTERNATE, and EXTRA text files or devices. Each of these SET commands can be enabled/disabled using the ON/OFF switch; the PRINTER, ALTERNATE and EXTRA output can be redirected to any file or device using the SET...TO option. In GUI mode, alternative printer output is available by PrintGui(.T.) or SET GUIPRINTER ON. The SET CONSOLE OFF can be used to suppress displaying to the screen without affecting output to the echoed device or text file.

After completing the ? / ?? command, the cursor or printhead is located one position to the right of the last character displayed. ROW() and COL() are updated to reflect the new cursor position. With SET PRINTER ON, PROW() and PCOL() are also updated with the new printhead position. When a different than the standard FONT is used, you may force the ROW() setting to correspond to the used font in the output either by the global switch SET ROWADAPT ON, or by invoking RowAdapt() thereafter. To align output using different fonts on the same base line, use SET ROWALIGN BASELINE.

To format any of the specified expressions, TRANSFORM() or a user- defined function can be used. If you need to pad a variable length value for column alignment, you can use any of the PAD() functions to left-justify, right-justify, or center the value.

Terminal i/o mode: If the output from ? or ?? command reaches the edge of the screen as reported by MAXCOL(), it wraps to the next line. If the output reaches the bottom of the screen as reported by MAXROW(), normally the screen scrolls up one line.

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.

Note: to display array elements, either specify the element (e.g. ? myarray[5,3]), or use separate Aeval() or _DisplArrStd() function. To display object properties, either specify ? myObj:objInstance or ? myObj:objMethod(), or use the _DisplObjStd() function.

Unicode:

In GUI mode, FlagShip supports also Unicode (UTF-8 and UTF-16). If the <expO4> font is set to Unicode by expO4:CharSet(FONT_UNICODE) or globally by oApplic:Font:CharSet(FONT_UNICODE) or SET GUICHARSET FONT_UNICODE, Unicode glyphs displays e.g. for Asian languages. Predefined strings needs to be stored in UTF-8 encoding, or transformed from UTF-16 by Utf16_Utf8(). In Linux, you may need to set Unicode font, e.g. SET FONT "mincho" as well. Since glyphs usually uses multiple bytes chr(128..255), it is recommended not to use SET GUITRANSLATE TEXT ON to draw semigraphics; you may convert PC-8 ASCII characters to Unicode by Cp437_utf8(). See also example in <FlagShip_dir>/examples/unicode.prg

Tuning:

In GUI mode, when scrollbars are enabled or set to auto (default), vertical and/or horizontal scrollbar displays when current screen output exceeds the visible window size. Max scrollbar area is set to 2000 pixels by default (approx. 100 lines * 200 chars, depends on current font), but may be changed by assigning e.g.

```
  _aGlobSetting[GSET_G_N_MAXSIZE_ROW] := row2pixel(500)
  _aGlobSetting[GSET_G_N_MAXSIZE_COL] := col2pixel(300)
```

The minimal scroll area size is 100 pixel, maximal 32100 pixel (it is automatically fixed). See also example 3 below.

In Terminal i/o mode, there are sometimes special characters like arrows and boxes/lines not displayed correctly. You may emulate arrows chr(24,25,26,27) and/or chr(16,17,30,31) by setting

```
  _aGlobSetting[GSET_T_N_EMUL_ARROWS] := num  // default = 0
```

where <num>	= 0	uses default display by font/mapping
	= 1	emulates arrows chr(24,25,26,27) by ASCII ^ v > <
	= 2	emulates arrows chr(24,25,26,27) by curses
	+ 16	emulates also chr(16,17,30,31) by (1) or (2)

To emulate boxes and lines chr(179 to 218) in Terminal i/o, use

```
_aGlobSetting[GSET_T_N_EMUL_BOXES] := num  // default = 0
```

where <num>	= 0	uses default display by font/mapping
	= 1	emulates boxes chr(179..218) by ASCII + - |
	= 2	emulates boxes chr(179..218) by curses

If the current line position exceeds the vertical scroll area size, Scroll() function is invoked automatically for the scrollbar area (see above), same as in Terminal i/o mode. On special needs, you may avoid the scrolling in GUI mode by

```
#include "applic.fh"
oApplic:Attrib := APP_SCROLL_OFF  // disable auto scroll()
oApplic:Attrib := APP_SCROLL_ON   // enable auto scroll(), def
```

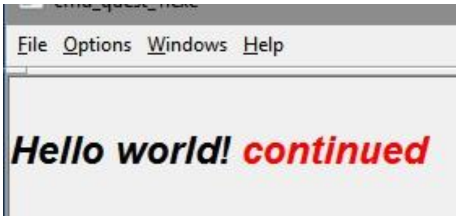
The functionality of Scroll() function is not affected. Apply for screen output in GUI mode only, ignored otherwise.

When printing to GDI printer via PrintGui(.T.), the printer's font size may slightly differ from screen size. This would cause smaller or larger spacing when continuing the ?? text output. FlagShip therefore uses by default the printer's font size for positioning the cursor on the screen. You may disable this feature by assigning

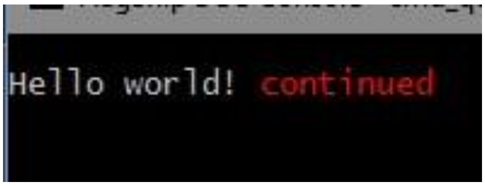
```
_aGlobSetting[GSET_G_L_PRINTER_SIZE_PREF] := .F.  // def = .T.
```

Example 1:

```
SET GUICOLOR ON       // same colors in GUI and Terminal i/o
? "Hello world! " FONT FontNew("Arial",18,"BI")
?? "continued "   FONT FontNew("Arial",18,"BI") COLOR "R+"
inkey(0)              // wait for keypress
```



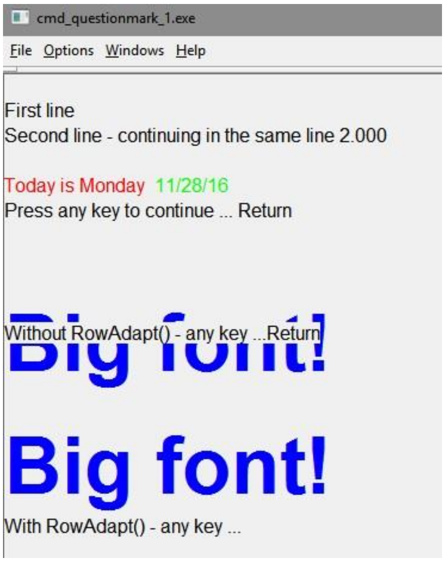
Hello world! continued


Hello world! continued

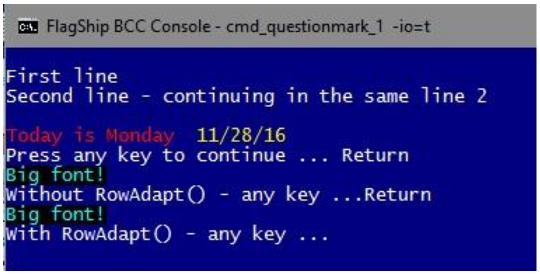
Example 2:

```
SET FONT "Arial", 12         // set default font (GUI only)
oApplic:Resize(25,80,,.T.)   // resize according to font (GUI only)
set color to "W+/B"
cls
// This will be displayed by default color
? "First line"
? "Second line"
?? " - continuing", "in the same line " + ltrim(row())
?  // empty line

// This will be displayed on the same line with different colors
? "Today is", CDOW(DATE()), " " COLOR "R+/B" GUICOLOR "R+"
?? DATE() COLOR "RG+/B" GUICOLOR "G+"
wait
```

```
oFont := Font{"Arial",50} // spacing now by this font!
oFont:Bold := .T.
? "Big font!" FONT oFont COLOR "BG+" GUICOLOR "B+" // GUI only
wait "Without RowAdapt() - any key ..."

? "Big font!" FONT oFont COLOR "BG+" GUICOLOR "B+" // GUI only
RowAdapt()      // adapt current Row() setting to larger font
wait "With RowAdapt() - any key ..."
```



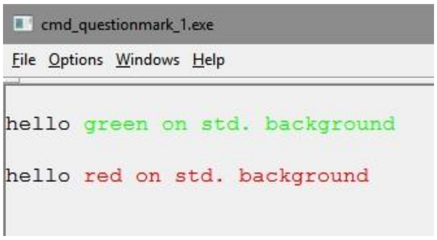
```
First line
Second line - continuing in the same line 2.000

Today is Monday 11/28/16
Press any key to continue ... Return
Without RowAdapt() - any key ...Return
Big font!
Big font!
With RowAdapt() - any key ...
```



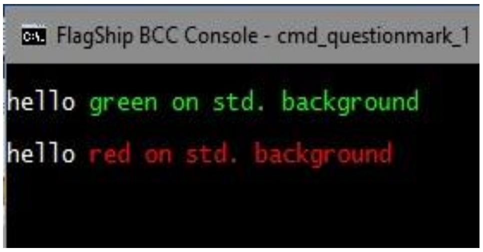
```
First line
Second line - continuing in the same line 2

Today is Monday  11/28/16
Press any key to continue ... Return
Big font!
Without RowAdapt() - any key ...Return
Big font!
With RowAdapt() - any key ...
```

Example 3: Display text by specific color

```
SET FONT "Courier", 12          // set default font (GUI only)
oApplic:Resize(25,80,,.T.)      // resize screen
set color to "W+/N"
cls
#include "color.fh"
? "hello "
?? "green on std. background " ;
   COLOR "G+/N" GUICOLOR "#00FF00"  // Terminal or GUI mode
?
? "hello "
?? "red on std. background" ;
   COLOR ("R+") GUICOLOR ("R+")
```



```
hello green on std. background

hello red on std. background
```



```
hello green on std. background

hello red on std. background
```

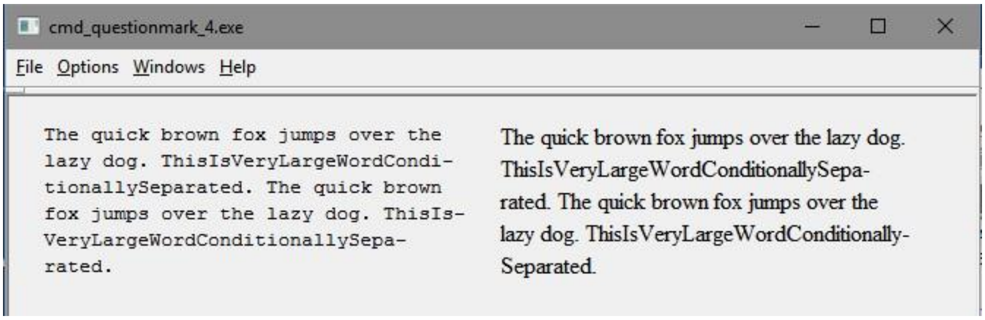
Example 4*:* Increase the GUI scroll area to 500 lines

```
_aGlobSetting[GSET_G_N_MAXSIZE_ROW] := row2pixel(500)
maxrow(.F., 500)
@ maxrow(),0 say "." ; clear       // increase instantly (faster)
for i := 1 to 495
   ? "line " + ltrim(i)            // does not scroll
next
wait "1..495, check by scrollbar ..."
rr := row()
for i := rr+1 to rr+10
   ? "line " + ltrim(i)            // will scroll
next
wait "8..506, check by scrollbar ..."
```

Example 5:

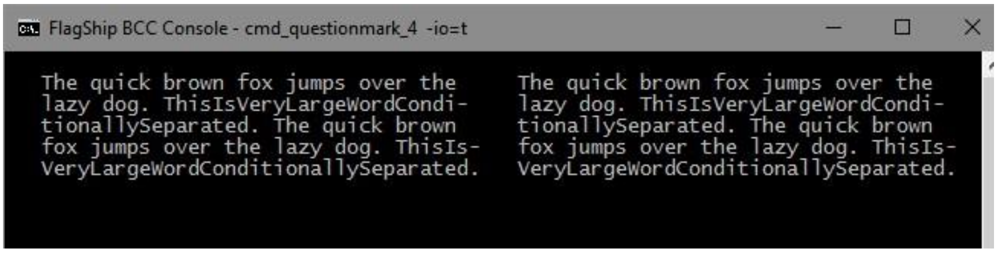
```
SET FONT "Courier", 10         // set default font (GUI only)
oApplic:Resize(25,80,,.T.)     // resize window

cText := replicate("The quick brown fox jumps over the lazy " + ;
                   "dog. This" + chr(1) + "Is" + chr(1) + ;
                   "Very" +chr(247) + "Large" +chr(247) + "Word" +;
                   chr(1) + "Con÷di÷tion÷ally÷Sep÷a÷rated. ", 2)

? space(3) + cText SPLIT 3,38
@ 1,42 say cText SPLIT 42,77  FONT Font{"Times", 12}

wait
```

Output:



```
The quick brown fox jumps over the
lazy dog. ThisIsVeryLargeWordCondi-
tionallySeparated. The quick brown
fox jumps over the lazy dog. ThisIs-
VeryLargeWordConditionallySepa-
rated.
```

The quick brown fox jumps over the lazy dog.
ThisIsVeryLargeWordConditionallySepa-
rated. The quick brown fox jumps over the
lazy dog. ThisIsVeryLargeWordConditionally-
Separated.



```
The quick brown fox jumps over the      The quick brown fox jumps over the
lazy dog. ThisIsVeryLargeWordCondi-     lazy dog. ThisIsVeryLargeWordCondi-
tionallySeparated. The quick brown      tionallySeparated. The quick brown
fox jumps over the lazy dog. ThisIs-    fox jumps over the lazy dog. ThisIs-
VeryLargeWordConditionallySeparated.    VeryLargeWordConditionallySeparated.
```

Example 6:

```
See several examples in <FlagShip_dir>/examples/*.prg and the
printergui.prg there for output to GUI/GDI printer.
```

Classification:

sequential screen output (SET CONSOLE ON) sequential printer output (SET PRINTER ON) sequential file output (SET EXTRA | ALTERNATE ON)

Compatibility:

FS4 and later supports embedded zero bytes by default. The COLOR and GUICOLOR clause is available in FS5 and later, SPLIT and PRINTCOLOR since VFS7. Unicode is available in VFS7 and later.

Translation: *see also std.fh file*

```
?    =>  QOUT   (exp1 [, exp2 ...] )
??   =>  QQOUT  (exp1 [, exp2 ...] )
?   COLOR/GUICOLOR/PRINTCOLOR/FONT
   => QOUT6 (col, guiCol, font, prCol,,, exp1 [,exp2 ...])
?? COLOR/GUICOLOR/PRINTCOLOR/FONT
   => QQOUT6(col, guiCol, font, prCol,,, exp1 [,exp2 ...])
?? SPLIT/COLOR/GUICOLOR/PRINTCOLOR/FONT
   => QsplitText(exp, c1, c2,, font, col, guiCol, prCol,,.T.)
```

Related:

@...SAY, @..DRAW, TEXT, COL(), ROW(), SET CONSOLE, SET ALTERNATE, SET EXTRA, SET HTMLTEXT, SET ROWADAPT, SET ROWALIGN, SET PRINTER, PrintGui()

?# | ??# | ??##

Syntax:

```
?# [<expList>]
```

Syntax:

```
??# [<expList>]
```

Syntax:

```
??## [<expList>]
```

Purpose:

Evaluates and displays the results of one or more expressions to the standard error device (stderr, usually console).

Arguments:

<**expList**> is a list of values or expressions to be evaluated and displayed. If there are more than one expression, the expressions must be separated by commas. The expressions can be of any data type, including memos.

If no argument is specified and the ?# command is used, a NEWLINE code is sent to stderr.

Description:

This command is often used for debugging purposes, where

?# ...	is similar to ? or Qout() and prints NewLine + text to stderr, same as the C statement fprintf(stderr,"\n...")
??# ...	is similar to ?? or Qqout() and prints text to stderr, same as the C statement fprintf(stderr,"...")
??##...	is similar to ?# but print text + NewLine to stderr, same as the usual C statement fprintf(stderr,"...\n")

The commands SET CONSOLE, SET ALTERNATE, SET FILE, SET PRINTER are not affected here and are also not considered.

Redirection: you may redirect this stderr output to a file (here named 'myfile') at the time of invoking your application 'myapp' (with optional command-line arguments):

● in MS-Windows, and in Unix/Linux using sh, bash, ksh, bash shell:

```
myapp [cmd-line arguments] 2>myfile          #overwrites myfile
myapp [cmd-line arguments] 2>>myfile         #appends to myfile
```

● in Unix/Linux using csh, tcsh script:

```
( myapp [cmd-line arguments] >/dev/tty ) >& myfile   #overwrites
( myapp [cmd-line arguments] >/dev/tty ) >>& myfile  #appends
```

● in Unix/Linux using newfswin script:

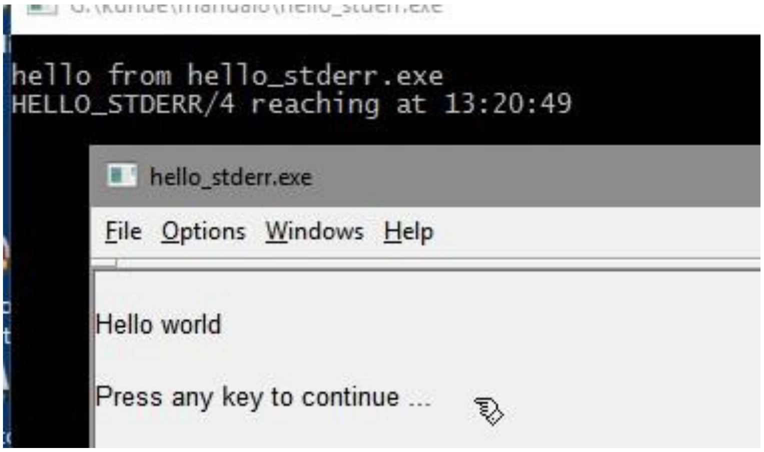
```
newfswin myapp [cmd-line arguments] 2\>myfile   #overwrites
newfswin myapp [cmd-line arguments] 2\>\>myfile #appends
```

In Unix/Linux, the 'myfile' may also be any device of your choice, e.g. /dev/lpt0 or /dev/pts/12. In Windows, you may redirect it to printer by specifying e.g. PRN: or LPT2: for 'myfile'.

If command-line redirection was not specified, the `?[?#]#` output appears in GUI mode on the console screen, in terminal and basic i/o mode intermixed with the standard ?, ?? and @... output.

Example:

```
? "Hello world"
?# "hello from " + execname()
?# procstack(), "reaching at", time()
wait
```



```
hello from hello_stderr.exe
HELLO_STDERR/4 reaching at 13:20:49
```

```
Hello world

Press any key to continue ...
```

Compatibility:

New in VFS5.

Related:

?, ??, OutErr(), OutStd(), Qout(), Qqout()

@...

Syntax:

```
@ <expN1>, <expN2>
           [PIXEL|NOPIXEL]
           [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
```

Purpose:

Clears to the end of line.

Arguments:

<**expN1**> and <**expN2**> are the starting row and column coordinates to clear. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

PIXEL : the <expN1>, <expN2> are values in pixel

NOPIXEL : the <expN1>, <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN4> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set(_SET_COORD_-UNIT,num). Apply for GUI mode only, ignored otherwise.

Description:

This command is used to clear the rest of the line <expN1> beginning at column <expN2>.

In GUI mode, if there is a (part of) widget in the cleared area, the widget is cleared as well, see also LNG.5.3.

After executing the command, the cursor (and ROW(), COL()) is set to <expN1>, <expN2>.

Example:

```
@ 10,15                                 && clear from 10,15 to eol
@ 11,0                                  && clears whole line 11
```

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND()

Translation:

SCROLL (expN1, expN2, expN1) ; SETPOS (expN1, expN2)

Related:

@...CLEAR, @...CLEAR TO, CLEAR, LNG.5.3

@...BOX

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
     BOX [<expC5>]
        [COLOR <expC6>]
        [GUICOLOR <expC7>] [PRINTCOLOR <expC8>]
        [SUNKEN|RAISED|PLAIN]
        [FRAMEONLY]
        [LINEWIDTH <expN9>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN10>)]
```

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
     GUI BOX [<expC5>]
        [COLOR <expC6>]
        [GUICOLOR <expC7>] [PRINTCOLOR <expC8>]
        [SUNKEN|RAISED|PLAIN]
        [FRAMEONLY]
        [LINEWIDTH <expN9>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN10>)]
```

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
     TERM BOX [<expC5>]
        [COLOR <expC6>]
```

Purpose:

Draws a customized box on the screen.

Arguments:

GUI BOX the box is driven only in GUI mode and ignored otherwise

TERM BOX the box is driven only in Terminal mode and ignored otherwise. If not specified, the box is drawn in GUI mode only when SET GUITRANSL BOX is ON, or any of GUICOLOR, SUNKEN, RAISED, PLAIN clause was given. This avoids unpleasant imagining of default widgets like Listbox, Tbrowse, Alert etc. by backward or Terminal i/o compatible sources.

<**expN1**...**expN4**> are the coordinates, upper, left, lower, and right respectively. The row coordinates can range from zero to 24, and the column coordinates can range from zero to 79 (or MAXROW() and MAXCOL() respectively, depending on the used terminfo description or the set screen size. In GUI mode, the coordinates specify mid of the character so the look-and-feel is comparable to Terminal i/o mode; to set the coordinate exactly at pixel value, use the PIXEL clause (or enable SET PIXEL ON). In

GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer if Terminal i/o mode is used. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

<**expC5**> is a character string containing eight border characters and optional one fill character. The first character is used for the upper left-hand corner, the next for the upper line, and so on, the clockwise. The box is filled with the ninth character, if any. If <expC5> is a single character, that character draws the whole border. You alternatively may use constants B_PLAIN, B_SINGLE, B_DOUBLE, B_SINGLE_DOUBLE, B_DOUBLE_SINGLE defined in box.fh specifying 8 chars of the border.

If <expC5> is not specified, the default value is taken from global variable _aGlobSetting[GSET_T_C_AT_TO_SINGLE] defined in initio.prg. If <expC5> is a variable named same as the significant part of BOX clauses, e.g. FRAM*, COLO*, LINE*, SUNK* etc, enclose the variable in parentheses to avoid confusions.

In Terminal i/o mode, the border and <expC5> is always considered. In GUI mode, <expC5> is ignored when SUNKEN, RAISED or PLAIN clause was specified. Special frames (except B_SINGLE and B_DOUBLE) and the fill character are drawn only w/o PIXEL or UNIT clause.

COLOR <**expC6**> is an optional color specification (according to SET COLOR). If not specified, the box is drawn using the current color setting. Only the first color pair is used. The frame is drawn by foreground/background, the box is filled by the background color. Apply for Terminal i/o. Apply also for GUI mode when SET GUICOLOR is ON, otherwise the GUICOLOR clause is used.

GUICOLOR <**expC7**> is an optional color specification (according to SET COLOR). If not specified, the box is drawn using the current color setting. Only the first color pair is used. The frame is drawn by foreground/background, the box is filled by the background color. Apply for GUI mode only and overrides the optional COLOR clause. If not specified, the default color is used to fill the box area, except the FRAMEONLY clause was given.

PRINTCOLOR <**expC8**> is an optional color specification (according to SET COLOR) for GUI/GDI printout by SET GUIPRINT ON. Only the first color pair (foreground or foreground/background) is considered. If not given, GUICOLOR is used also for printer, but with foreground only.

SUNKEN :	creates 3-dim panel with sunken effect
RAISED :	creates 3-dim panel with raised effect
PLAIN :	draws plain (2-dimensional) box frame

These three clauses apply for GUI mode and overrides <expC5>. When this clause is specified, the box is drawn in GUI mode regardless SET GUITRANSL BOX on/off. If FRAMEONLY clause is not specified, the box is filled by background color. Ignored in Terminal i/o mode.

FRAMEONLY : apply for GUI mode with SUNKEN, RAISED, PLAIN clause, ignored otherwise. It suppress filling the box by background color but draws the box frame only.

LINEWIDTH <**expN9**> is optional line width (in pixel) of the frame used in GUI mode with SUNKEN, RAISED or PLAIN clause. If the argument is 0, no frame is drawn, only background color is filled. When LINEWIDTH is not specified, default is _aGlobSetting [GSET_G_N_DRAWLINE] .

PIXEL : the <expN1> .. <expN4> are values in pixel

NOPIXEL : the <expN1> .. <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN10> specifies unit for <expN1> .. <expN4> coordinates. The <expN10> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set (_SET_COORD_UNIT,num). Apply for GUI mode only, ignored otherwise.

Description:

The command @...BOX is used for drawing boxes using a configurable border and filling it with a specified character. After @...BOX is executed, the cursor and ROW(), COL() are set into the boxed region at <expN1> +1, <expN2> +1.

In GUI mode, the box is drawn only when SET GUITRANSL BOX is ON, or when any of GUI, GUICOLOR, SUNKEN, RAISED, PLAIN clause was given. You may:

- use sunken, raised or plain box frame, when SUNKEN, RAISED, PLAIN clause is specified. The background color of GUICOLOR is considered, filling character of <expC5> is ignored with these clauses.
- use the semi-graphic characters in <expC5>, simulated via line drawing, when SET GUITRANSL BOX is ON or the GUI or GUICOLOR clause was specified, and no SUNKEN, RAISED, PLAIN clauses was given. This is the "old", backward compatible syntax. The background color and filling character in <expC5> is supported but it may cause unwanted results with proportional fonts.

In Terminal i/o mode, the box is always drawn by the <expC5> chars. The optional SUNKEN, RAISED, PLAIN, LINEWIDTH, GUICOLOR, FRAMEONLY and PIXEL clauses are ignored. The color set by COLOR clause, as well as the fill character of <expC5> are considered.

Note that @...BOX 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.

The @..BOX command is processed also for GUI/GDI printout (when SET GUIPRINT ON is active) and accepts only PRINTCOLOR, LINEWIDTH, PIXEL and UNIT= clauses, other (incl. <expC5>) are ignored.

An alternative to @..BOX in GUI mode is @..DRAW RECTANGLE which does not require GUITRANSL BOX ON and optionally draws rounded rectangle.

Unicode:

In GUI mode, when the default font is set to Unicode by oApplic:Font: CharSet(FONT_UNICODE) or global by SET GUICHARSET FONT_UNICODE, boxes are emulated by line drawing when <expC5> starts with B_SINGLE or B_DOUBLE string (see box.fh). For other options, use @..GUI BOX... instead.

Tuning:

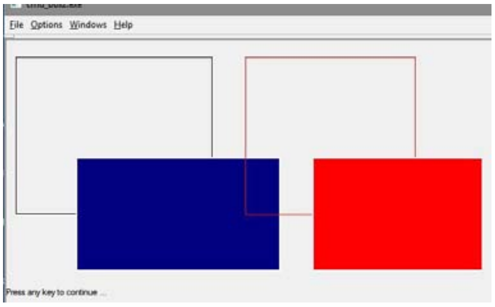
In GUI mode, you may set

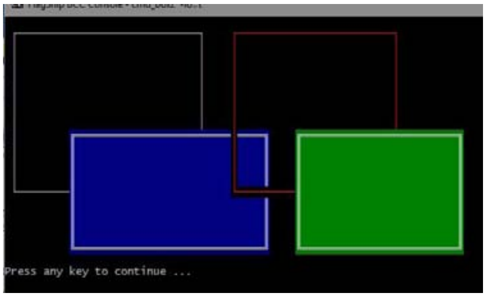
```
    _aGlobSetting[GSET_G_L_BOX_FRAME_BLACK] := .T.  // default = .F.
```

to draw the boxframe in black instead of the foreground color. Do not apply for SUNKEN or RAISED boxes.

Example 1:

```
// Draw the frame and box in both Terminal i/o and GUI modes
#include "box.fh"                 // for B_DOUBLE and B_SINGLE
SET GUICOLOR ON                   // accept COLOR clause in GUI
SET GUITRANSL BOX ON              // draw boxes in GUI mode
@  1, 1, 15,30 BOX B_SINGLE
@ 10,10, 20,40 BOX B_DOUBLE + " " COLOR "W+/B"

// GUI mode only
@ 1, 35, 15,60 GUI BOX PLAIN FRAMEONLY GUICOLOR "R+"
@ 10,45, 20,70 GUI BOX B_DOUBLE + " "  COLOR "W+/R+"

// Terminal mode only
@ 1, 35, 15,60 TERM BOX PLAIN COLOR "R+"
@ 10,45, 20,70 TERM BOX B_DOUBLE + " "  COLOR "W+/G+"

setpos(21,0)
wait
```

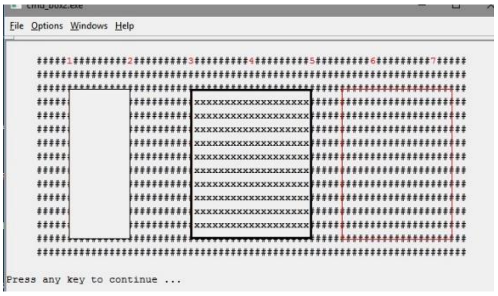




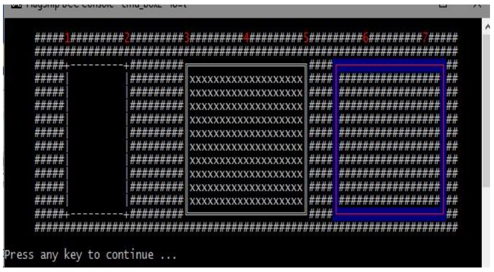
```
Press any key to continue ...
```

Example 2:

```
       * Draw box with standard or extended ASCII char set:
       *
       * ########################################################
       * ###+----------+#####╔═══════════╗########┌----------┐###
       * ###|          |#####║xxxxxxxxxxx║########|##########|###
       * ###|          |#####║xxxxxxxxxxx║########|##########|###
       * ###+----------+#####╚═══════════╝########└----------┘###
       * ########################################################
       *

#include "box.fh"          // for B_DOUBLE and B_SINGLE
SET GUICOLOR ON            // use COLOR in no GUICOLOR specified
SET GUITRANSL BOX ON       // always display boxes in GUI mode

filler1 = "+-+|+-+| "
filler2 = chr(201, 205, 187, 186, 188, 205, 200, 186) + "x"
filler3 = chr(218, 196, 191, 179, 217, 196, 192, 179)

@ 1, 5,15,75 BOX replicate("#", 9)           // Background
for ii := 1 to 7
   @ 1,ii*10 say ltrim(ii) color "R+"
next
@ 3,10,14,20 BOX filler1                     // Box 1
@ 3,30,14,50 BOX B_DOUBLE + "x"              // Box 2
@ 3,55,14,73 BOX filler3 COLOR "R+/B"        // Box 3
setpos(16,0)
wait
```



```
Press any key to continue ...
```

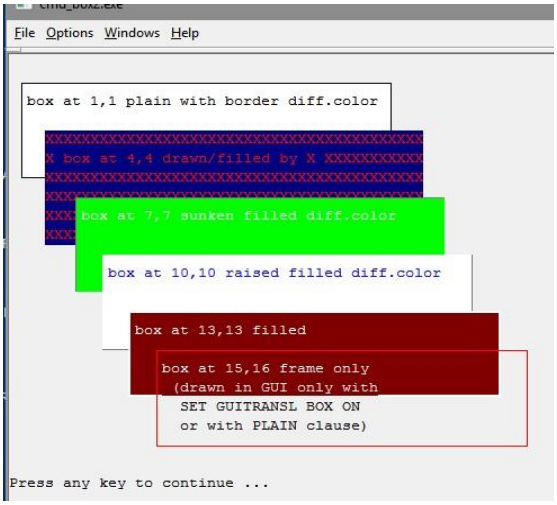


```
Press any key to continue ...
```

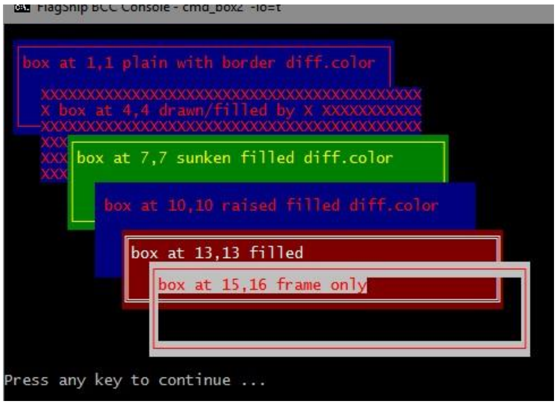
Example 3:

```
// draw boxes by different GUI and Terminal colors
#include "box.fh"          // for B_DOUBLE and B_SINGLE
SET FONT "courier",10      // use fixed font
SET GUICOLOR ON            // use COLOR in no GUICOLOR specified
SET GUITRANSL BOX ON       // always display boxes in GUI mode

@ 1,1, 6,42 BOX B_SINGLE + " " PLAIN COLOR "R+/B" GUICOLOR "N/W+"
?? "box at 1,1 plain with border diff.color" ;
        COLOR "R+/B" GUICOLOR "N/W+"

@ 4,4, 9,45 BOX repli("X",9) COLOR "R+/B"
?? " box at 4,4 drawn/filled by X " COLOR "R+/B"

@ 7,7, 12,48 BOX B_SINGLE + " " SUNKEN ;
```

```
         COLOR "GR+/G" GUICOLOR "W+/G+"

?? "box at 7,7 sunken filled diff.color" ;
         COLOR "GR+/G" GUICOLOR "W+/G+"

@ 10,10, 15,51 BOX space(9) RAISED COLOR "R+/B" GUICOLOR "B+/W+"
?? "box at 10,10 raised filled diff.color" ;
         COLOR "R+/B+" GUICOLOR "B+/W+"

@ 13,13, 17.4,54 BOX B_DOUBLE + " " COLOR "W+/R"
?? "box at 13,13 filled" COLOR "W+/R"

// SET GUITRANSL BOX OFF     // don't display boxes in GUI mode
@ 15,16, 20,57 BOX B_SINGLE COLOR "R+/W"
?? "box at 15,16 frame only" COLOR "R+/W" GUICOLOR "W+/R"
if AppIoMode() == "G"
   @ 17,17 say " (drawn in GUI only with" COLOR "W+/R"
   @ 18,17 say "  SET GUITRANSL BOX ON"   COLOR "N"
   @ 19,17 say "  or with PLAIN clause)"  COLOR "N"
endif
```



```
box at 1,1 plain with border diff.color
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X box at 4,4 drawn/filled by X XXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXX box at 7,7 sunken filled diff.color
    box at 10,10 raised filled diff.color
        box at 13,13 filled
            box at 15,16 frame only
             (drawn in GUI only with
              SET GUITRANSL BOX ON
              or with PLAIN clause)

Press any key to continue ...
```

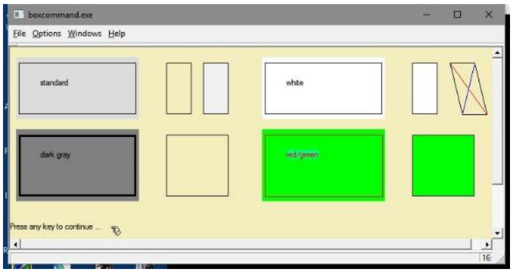


```
box at 1,1 plain with border diff.color
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  X box at 4,4 drawn/filled by X XXXXXXXXXXX
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXX box at 7,7 sunken filled diff.color
      box at 10,10 raised filled diff.color
          box at 13,13 filled
              box at 15,16 frame only

Press any key to continue ...
```

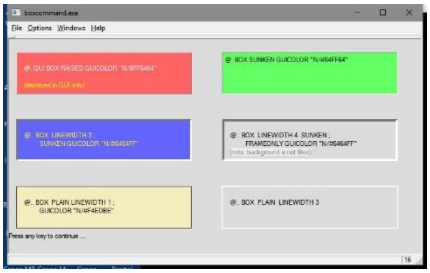
Example 4:

See <FlagShip_dir>/examples/boxcommand.prg for additional examples

Output in GUI mode:

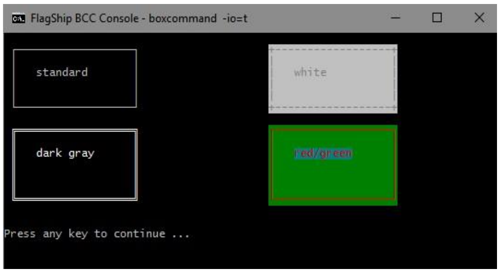


```
standard
white
dark gray
red/green
Press any key to continue ...
```

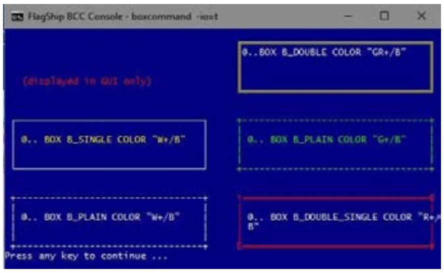


```
@..GUI BOX RAISED GUICOLOR "N/#FF6464"
(displayed in GUI only)
@ BOX SUNKEN GUICOLOR "N/#64FF64"
@.. BOX LINEWIDTH 3 ;
    SUNKEN GUICOLOR "N/#6464FF"
@.. BOX LINEWIDTH 4 SUNKEN ;
    FRAMEONLY GUICOLOR "N/#6464FF"
(note: background is not filled)
@.. BOX PLAIN LINEWIDTH 1 ;
    GUICOLOR "N/#F4EDBE"
@.. BOX PLAIN LINEWIDTH 3
Press any key to continue ...
```

Output in terminal i/o mode:



```
standard                                  white

dark gray                                 red/green

Press any key to continue ...
```



```
                                   @..BOX B_DOUBLE COLOR "GR+/B"
 (displayed in GUI only)

 @.. BOX B_SINGLE COLOR "W+/B"     @.. BOX B_PLAIN COLOR "G+/B"

 @.. BOX B_PLAIN COLOR "W+/B"      @.. BOX B_DOUBLE_SINGLE COLOR "R+/
                                   B"
Press any key to continue ...
```

Include:

The #include file "box.fh" contains predefined PC-8 border character combinations.

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND() in terminal i/o, as well as GUI printout

Compatibility:

In Terminal i/o mode for Linux, the physical output on the screen depends on the terminal description selected (environment variable TERM), the ability of the terminal to output mapping applied via FSchrmap.def. See also LNG.5.1.4, section SYS, and FS_SET ("outmap")

GUI printout (by PrintGui() or SET GUIPRINT ON) is available in GUI mode only.

Translation:

```
DISPBOX (expN1, expN2, expN3, expN4, expC5, [color], [lPixel],
        [lGUI], [GuiColor], [nPlainMode], [nLineWidth], [lFrame],
        [PrintColor] )
```

Related:

@..DRAW RECTANGLE, @...CLEAR, @...TO, LNG.5.3

@...CLEAR

Syntax:

```
@ <expN1>,<expN2> CLEAR
        [TO <expN3>,<expN4>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
```

Purpose:

Clears a screen region.

Arguments:

<**expN1**> and <**expN2**> are the row and column coordinates of the upper left corner. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer if Terminal i/o mode is used. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

Options:

TO <**expN3**> and <**expN4**> are the row and column coordinates of the lower right corner. If this option is not specified, the screen is cleared from the specified upper left corner to 24,79 (or MAXCOL() and MAXROW() respectively), as specified in the terminfo description used.

PIXEL : the <expN1> .. <expN4> are values in pixel

NOPIXEL : the <expN1> .. <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN4> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set (_SET_COORD_UNIT,num). Apply for GUI mode only, ignored otherwise.

Description:

This command can be used to clear a rectangular region of the screen by filling it with space characters of the current color setting.

After @...CLEAR erases the designated region, the cursor is positioned in the upper left corner of the cleared region at <expN1> and <expN2>. In GUI mode, if there is a (part of) widget in the cleared area, the widget is erased as well. ROW() and COL() coordinates are updated to reflect the new cursor position.

In Terminal i/o mode, the screen background corresponds to the standard color pair, set by SetColor() or SET COLOR TO command.

In GUI mode, the background color (assigned by SET COLOR) is set only when SET GUICOLOR is ON (default is OFF - according to GUI design specs). You may set the background also explicitly by invoking SetColorBackground(cColor) followed by CLS, CLEAR SCREEN, Scroll() or @ ... CLEAR [TO..]

Example:

```
LOCAL scr
scr = SAVESCREEN (10,10,20,60)
@ 10,10 CLEAR TO 20,60
@ 10,10 TO 20,60 DOUBLE
*
* additional output in the window
*
RESTSCREEN (10,10,20,60,scr)
```

Classification:

screen oriented output, in terminal i/o mode buffered via DISPBEGIN()..DISPEND()

Compatibility:

[PIXEL|NOPIXEL] clause is new in FS5

Translation:

```
SCROLL (expN1, expN2 [, expN3, expN4] )
SETPOS (expN1, expN2)
```

Related:

@...BOX, @...TO, CLEAR, RESTSCREEN(), SAVESCREEN(), LNG.5.3

@...DRAW ARC

Syntax 1:

```
      @ <expN1>,<expN2> [GUI]
           DRAW ARC RADIUS <expN3> ANGLE <expN5>,<expN6>
              [COLOR <color>]
              [GUICOLOR <color>]
              [PRINTCOLOR <expC7>]
              [LINEWIDTH <expN8>]
              [PIXEL|NOPIXEL]
              [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)]
```

#***Syntax 2:***

```
      @ <expN1>,<expN2>, <expN3>,<expN4> [GUI]
           DRAW ARC ANGLE <expN5>,<expN6>
              [COLOR <color>]
              [GUICOLOR <color>]
              [PRINTCOLOR <expC7>]
              [LINEWIDTH <expN8>]
              [PIXEL|NOPIXEL]
              [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)]
```

Purpose:

Draws circle or ellipse part in GUI mode, specified by radius or bounding rectangle.

Arguments Syntax 1:

<expN1>, <expN2> are row/col or y/x coordinates of the circle center in specified or default <units>. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

RADIUS <expN3> is the circle radius in specified or default <units>. With row/cols unit, <expN3> is assumed as (fractional) number of columns.

ANGLE <expN5>,<expN6> are the start angle and the arc length in positive or negative degrees (-360..0..360). Positive values mean counter-clockwise while negative values mean the clockwise direction. Zero degree of <expN5> is at the 3 o'clock position, the 12 o'clock position is either 90 or -270. The arc length <expN6> is the drawn part of circle or ellipse in positive or negative degrees starting at <expN5>. The direction is clockwise when both <expN5> and <expN6> are positive or negative, or counter-clockwise otherwise, see also example below.

Arguments Syntax 2:

<expN1>, <expN2> are top left row/col or y/x coordinates of the bounding rectangle in specified or default <units>.

<expN3>, <expN4> are bottom right row/col or y/x coordinates of the bounding rectangle in specified or default <units>. If the bounding rectangle (calculated in pixels) is quadratic, circle arc is drawn.

ANGLE <expN5>,<expN6> are the start angle and the arc length in positive or negative degrees (or degree fractions), same as in Syntax 1 above.

Options:

COLOR <color> or **GUICOLOR <color>** is optional color specification. The circle or ellipse arc is drawn by foreground color in width of <expN8> pixel, background color is ignored.

PRINTCOLOR <expC7> specifies the color for printing by SET GUIPRINT ON, or with PrintGui(.T.). The circle or ellipse arc is drawn by foreground color. If PRINTCOLOR is not given, GUICOLOR is used also for printer.

LINEWIDTH <expN8> is the line width of the circle or ellipse arc in pixels. If not given, line width of 1 pixel is used.

PIXEL : the <expN1>..<expN4> are values in pixel

NOPIXEL : the <expN1>..<expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN9> specifies unit for <expN1> .. <expN4> coordinates. The <expN9> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set (_SET_COORD_UNIT,num). Apply for GUI mode only, ignored otherwise.

Description:

@...DRAW ARC draws parts of ellipse or circle specified by rounding rectangle or circle radius on screen and/or printer in GUI mode. It is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active).

The row() and col() values are set accordingly to draw end.

To draw full circle or ellipse, @..DRAW CIRCLE or @..DRAW ELLIPSE may be used instead which supports also filling color. The @..DRAW PIE command is another alternative to @..DRAW ARC.

Tuning:

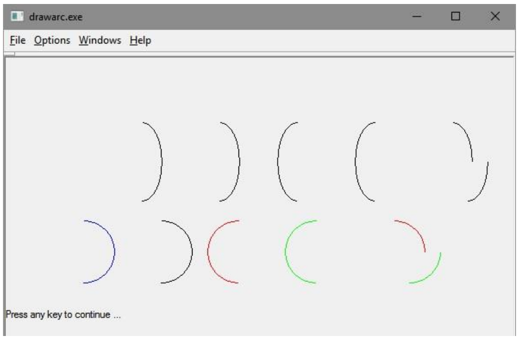
In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
_aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

Example:

```
@ 5,15, 11,20 DRAW ARC ANGLE -90, 180 LINEW 2          // ")"
@ 5,25, 11,30 DRAW ARC ANGLE  90,-180 LINEW 2          // ")"
@ 5,35, 11,40 DRAW ARC ANGLE  90, 180 LINEW 2          // "("
@ 5,45, 11,50 DRAW ARC ANGLE -90,-180 LINEW 2          // "("
@ 5,55, 11,60 DRAW ARC ANGLE   0,  90 LINEW 2          // ")" top
@ 5,57, 11,62 DRAW ARC ANGLE   0, -90 LINEW 2          // ")" bott

@ 15,10 DRAW ARC RADIUS 4 ANGLE -90, 180 GUICOLOR "B+"  // ")"
@ 15,20 DRAW ARC RADIUS 4 ANGLE  90,-180 GUICOLOR "N"   // ")"
@ 15,30 DRAW ARC RADIUS 4 ANGLE  90, 180 GUICOLOR "R+"  // "("
@ 15,40 DRAW ARC RADIUS 4 ANGLE -90,-180 GUICOLOR "G+"  // "("
@ 15,50 DRAW ARC RADIUS 4 ANGLE   0,  90 GUICOLOR "R+"  // ")" top
@ 15,52 DRAW ARC RADIUS 4 ANGLE   0, -90 GUICOLOR "G+"  // ")" bott
```

Output:

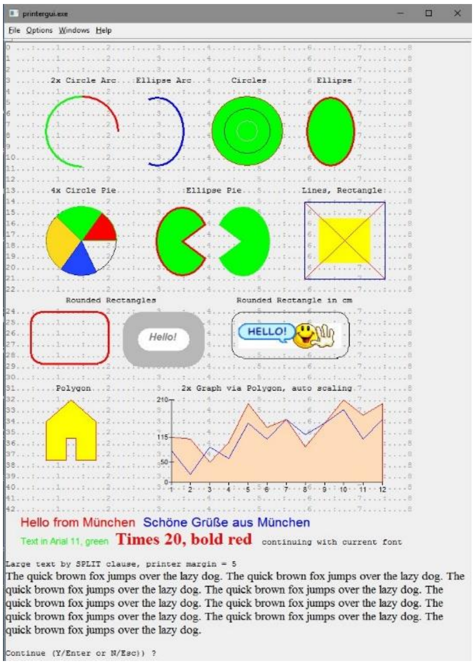


```
Press any key to continue ...
```

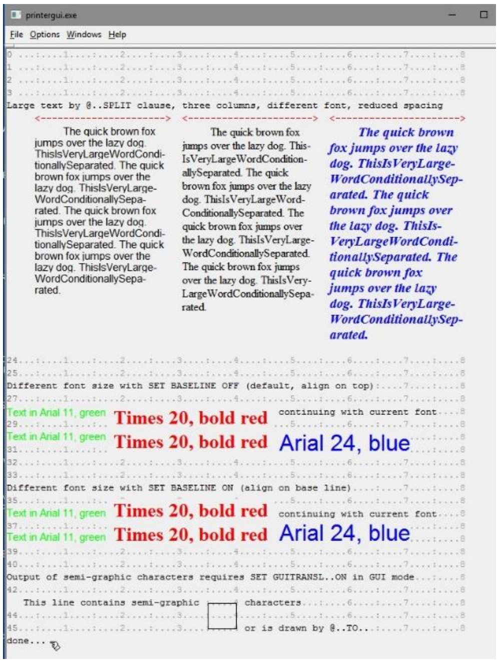
Example:

```
See complete example in <FlagShip_dir>/examples/printergui.prg
```

Output:



```
Hello from München  Schöne Grüße aus München
Text in Arial 11, green  Times 20, bold red  continuing with current font

Large text by SPLIT clause, printer margin = 5
The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog.

Continue (Y/Enter or N/Esc)) ?
```



```
Large text by @..SPLIT clause, three columns, different font, reduced spacing
```

The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated.

The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated.

The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated. The quick brown fox jumps over the lazy dog. ThisIsVeryLargeWordConditionallySeparated.

```
Different font size with SET BASELINE OFF (default, align on top)
Text in Arial 11, green  Times 20, bold red  continuing with current font
Text in Arial 11, green  Times 20, bold red  Arial 24, blue

Different font size with SET BASELINE ON (align on base line)
Text in Arial 11, green  Times 20, bold red  continuing with current font
Text in Arial 11, green  Times 20, bold red  Arial 24, blue

Output of semi-graphic characters requires SET GUITRANSL..ON in GUI mode
   This line contains semi-graphic       characters
                                         or is drawn by @..TO..
done...
```

Classification:

screen oriented output in GUI mode, GUI printout

Compatibility:

New in FS7, not available in Clipper nor in FoxPro.

Translation: *in std.fh*

```
GuiDrawArc(...)
```

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW PIE, @..DRAW LINES, @..DRAW IMAGE, @..DRAW POLYON, @..DRAW RECTANGLE, @...BOX, @...TO.., SET GUIPRINT, PrintGui()

@...DRAW CIRCLE

Syntax:

```
@ <expN1>,<expN2> [GUI]
     DRAW CIRCLE RADIUS <expN3>
        [COLOR <color>]
        [GUICOLOR <color>]
        [PRINTCOLOR <expC5>]
        [LINEWIDTH <expN6>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN7>)]
```

Purpose:

Draws a circle in GUI mode (ignored in Terminal i/o) of specified radius, color and line width, optional fill.

Arguments:

<expN1, expN2> are row/col or y/x coordinates of the circle center in specified or default <units>. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

RADIUS <expN3> is the circle radius in specified or default <units>. With row/cols unit, <expN3> is assumed as (fractional) number of columns.

Options:

COLOR <color> or **GUICOLOR <color>** is optional color specification. The circle is drawn by foreground color in width of <expN6> pixel, and filled by background color (if such given). To draw circle in mono color, use the same color for foreground and background.

PRINTCOLOR <expC5> specifies the color for printing by SET GUIPRINT ON, or with PrintGui(.T.). The circle is drawn by foreground and filled by background color. If PRINTCOLOR is not given, GUICOLOR is used also for printer.

LINEWIDTH <expN6> is the line width of the circle in pixels. If not given, line width of 1 pixel is used.

PIXEL : the <expN1>, <expN2> are values in pixel

NOPIXEL : the <expN1>, <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN7> specifies unit for <expN1> .. <expN2> coordinates. The <expN7> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

Description:

@...DRAW CIRCLE draws a circle of specified radius (the diameter is twice of the radius) on screen and/or printer in GUI mode. This command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active).

The row() and col() values are set accordingly to draw end.

To draw circle fragments, use @..DRAW ARC or @..DRAW PIE instead. To draw circle specified by bounding rectangle, use @..DRAW ELLIPSE.

Tuning:

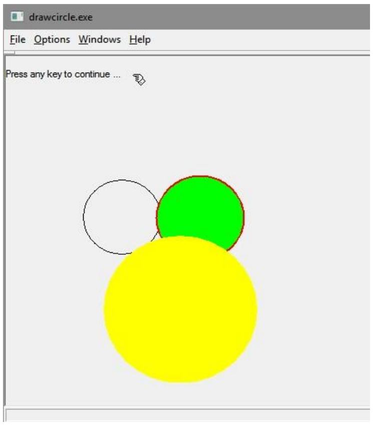
In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
  _aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

Example:

```
 @  13, 15 DRAW CIRCLE RADIUS 5           // diameter is 10 columns
 @  13, 25 DRAW CIRCLE RADIUS 5.6 GUICOLOR "R+/G+" LINEWIDTH 3
 @  8.3,5.5 DRAW CIRCLE RADIUS 2.4 GUICOLOR "RG+/RG+" UNIT=CM
```

Output:



Press any key to continue ...

Example:

```
 See complete example in <FlagShip_dir>/examples/printergui.prg
```

Classification:

screen oriented output in GUI mode, GUI printout

Compatibility:

New in FS7, not available in Clipper nor in FoxPro.

Translation: *in std.fh*

```
 GuiDrawCircle(...)
```

Related:

@..DRAW ELLIPSE, @..DRAW ARC, @..DRAW PIE, @..DRAW LINES, @..DRAW POLYON, @..DRAW IMAGE, @..DRAW RECTANGLE, @...BOX, @...TO.., SET GUIPRINT, PrintGui()

@...DRAW ELLIPSE

Syntax:

```
@ <expN1>,<expN2>, <expN3>,<expN4> [GUI]
     DRAW ELLIPSE
        [COLOR <color>]
        [GUICOLOR <color>]
        [PRINTCOLOR <expC5>]
        [LINEWIDTH <expN6>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN7>)]
```

Purpose:

Draws ellipse or circle (specified by bounding rectangle) in GUI mode, optional fill by specified color.

Arguments:

<expN1>, <expN2> are top left row/col or y/x coordinates of the bounding rectangle in specified or default <units>. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

<expN3>, <expN4> are bottom right row/col or y/x coordinates of the bounding rectangle in specified or default <units>.

Options:

COLOR <color> or **GUICOLOR <color>** is optional color specification. The ellipse (or circle) is drawn by foreground color in width of <expN6> pixel, and filled by background color (if such given). To draw it in mono color, use the same color for foreground and background.

PRINTCOLOR <expC5> specifies the color for printing by SET GUIPRINT ON, or with PrintGui(.T.). The ellipse or circle is drawn by foreground and filled by background color. If PRINTCOLOR is not given, GUICOLOR is used also for printer.

LINEWIDTH <expN6> is the line width of the ellipse or circle in pixels. If not given, line width of 1 pixel is used.

PIXEL : the <expN1>, <expN2> are values in pixel

NOPIXEL : the <expN1>, <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN7> specifies unit for <expN1> .. <expN2> coordinates. The <expN7> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

Description:

@...DRAW ELLIPSE draws ellipse on screen and/or printer in GUI mode. If the bounding rectangle (calculated in pixels) is quadratic, circle is drawn. This command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active).

The row() and col() values are set accordingly to draw end.

To draw ellipse or circle fragments, use @..DRAW ARC or @..DRAW PIE instead. To draw circle specified by it radius, use @..DRAW CIRCLE.

Tuning:

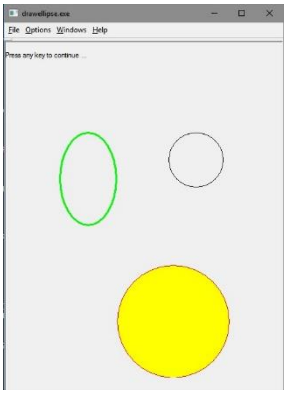
In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
   _aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

Example:

```
 @ 10,10,20,20 DRAW ELLIPSE GUICOLOR "G+" LINE 4         // ellipse
 rSize := pixel2row(col2pixel(10))       // = 10 columns into rows
 @ 10,30,10+rSize,40 DRAW ELLIPSE                        // circle
 @ 100,50,150,100 DRAW ELLIPSE GUICOLOR "R+/RG+" UNIT=MM // circle
```

Output:



```
Press any key to continue ...
```

Example:

```
 See complete example in <FlagShip_dir>/examples/printergui.prg
```

Classification:

screen oriented output in GUI mode, GUI printout

Compatibility:

New in FS7, not available in Clipper nor in FoxPro.

Translation: *in std.fh*

GuiDrawEllipse(...)

Related:

@..DRAW CIRCLE, @..DRAW ARC, @..DRAW PIE, @..DRAW LINES, @..DRAW POLYON, @..DRAW IMAGE, @..DRAW RECTANGLE, @..BOX, @...TO.., SET GUIPRINT, PrintGui()

@...DRAW IMAGE

Syntax 1:

```
@ <expN1>, <expN2>, [<expN3>], [<expN4>]
     DRAW IMAGE [FROM] FILE <expC6>
        [SCALE]
        [CLIP|NOSCALE]
        [IMGTYPE <expC7>]
        [BORDER|FRAME <expN8>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)]
```

Syntax 2:

```
@ <expN1>, <expN2>, [<expN3>], [<expN4>]
     DRAW IMAGE [USING] <expC5>
        [SCALE]
        [CLIP|NOSCALE]
        [IMGTYPE <expC7>]
        [BORDER|FRAME <expN8>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)]
```

Purpose:

Display bitmap image at specified screen position. Applicable in GUI mode only, ignored otherwise.

Syntax 1 reads the image from file,

Syntax 2 uses image data stored in database or character variable.

Arguments:

This command and it arguments is fully equivalent to @...SAY IMAGE, see detailed description there.

Description:

This command displays bitmap image at specified position in GUI mode, considered also by GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active). This is an alternative syntax for the @...SAY IMAGE command, see the full description there.

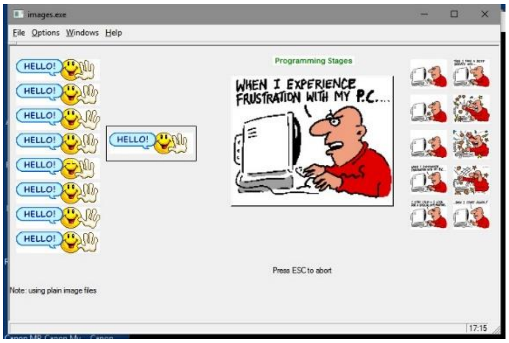
Example 1:

```
#include "box.fh"
@ 10,40 DRAW IMAGE file "myimg.gif"
cImgVar := "..\images\otherimage.bmp"
@ 15,50,18 DRAW IMAGE from file (cImgVar)
@ 350,500,480,600 SAY IMAGE file "myimg.jpg" PIXEL NOSCALE

local cImgData := memoread("../images/myimg.png")
@ 10,40,,60 DRAW IMAGE (cImgData) SCALE border BOX_SUNKEN
```

Example 2:

```
see also <FlagShip_dir>/examples/images.prg and printergui.prg for
additional examples
```

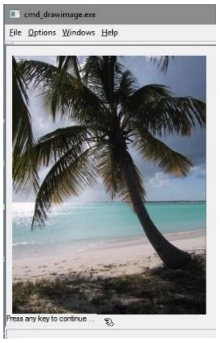
Output:


images.exe
File Options Windows Help
Programming Stages
HELLO!
WHEN I EXPERIENCE FRUSTRATION WITH MY P.C....

Press ESC to abort

Note: using plain image files

17:15



Press any key to continue ...

Classification:

screen oriented output in GUI mode as well as GUI printout

Compatibility:

New in FS7

Translation:

DispImageData() or DispImageFile()

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW ARC, @..DRAW LINES, @..DRAW PIE, @..DRAW POLYON, @..DRAW RECTANGLE, @...BOX, @...TO.., SET GUIPRINT, PrintGui(), MemoCode(), MemoDecode()

@...DRAW LINE

Syntax 1:

```
@ <expN1>,<expN2> [GUI]
      DRAW [LINES] [TO] <expN3>,<expN4>
         [COLOR <color> | GUICOLOR <color>]
         [PRINTCOLOR <expC5>]
         [PIXEL|NOPIXEL]
         [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN6>)]
         [WIDTH <wpix> | LINEWIDTH <wpix>]
```

Syntax 2:

```
@ [GUI] DRAW [LINES] [TO] <expN3>,<expN4>
         [COLOR <color> | GUICOLOR <color>]
         [PRINTCOLOR <expC5>]
         [PIXEL|NOPIXEL]
         [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN6>)]
         [WIDTH <wpix> | LINEWIDTH <wpix>]
```

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 With Syntax 2, the line drawing is continued from the current position. You may use numeric values with decimal fractions for row and column, or the PIXEL clause (or SET PIXEL ON) to set the pen exactly at specified pixel position relative to the user screen. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

Options:

COLOR <color> or **GUICOLOR <color>** is optional color specification. Only the foreground color is considered.

PRINTCOLOR <expC5> specifies the color for printing. If not given, GUICOLOR is used also for printer. Considered only in GUI mode when SET GUIPRINT is ON, or with PrintGui(.T.), and ignored otherwise.

PIXEL : the <expN1> .. <expN4> are values in pixel

NOPIXEL : the <expN1> .. <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN6> specifies unit for <expN1> .. <expN4> coordinates. The <expN6> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_CO-ORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

WIDTH <wpix> is the width of the drawn line in pixels

Description:

@...DRAW draws a line from the start to the end coordinate (syntax 1) or from the current position to the end coordinate (syntax 2). It applies for GUI mode only, ignored in other modes. The setting of SET GUITRANSL LINES is not relevant here.

You alternatively may use @..DRAW POLYGON to draw lines specified in array of coordinate pairs, and optionally fills the polygon area by background color.

The @..DRAW command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active) and accepts only PRINTCOLOR, WIDTH, PIXEL and UNIT=... clauses, other are ignored.

Tuning:

In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

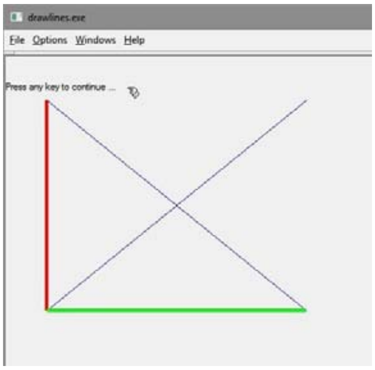
```
  _aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

Example:

Draw a large "X" and "L"

```
 @  3, 5 DRAW TO 20,40 COLOR "B+"
 @  3,40 DRAW TO 20,5  COLOR "B+"
 @  3, 5 DRAW TO 20,5  COLOR "R+" WIDTH 5
 @  DRAW TO 20,40      COLOR "G+" WIDTH 5  // continued
```

Output:



Classification: screen oriented output in GUI mode, GUI printout

Compatibility:

New in FS5, not available in Clipper. GUI printout is available since VFS7.

Translation:

GuiDrawLine(...)

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW ARC, @..DRAW PIE, @..DRAW POLYON, @..DRAW IMAGE, @..DRAW RECTANGLE, @...BOX, @...TO.., SET GUIPRINT, PrintGui()

@...DRAW PIE

Syntax 1:

```
@ <expN1>,<expN2> [GUI]
     DRAW PIE RADIUS <expN3> ANGLE <expN5>,<expN6>
        [COLOR <color> | GUICOLOR <color>]
        [PRINTCOLOR <expC7>]
        [LINEWIDTH <expN8>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)
           ]
```

#***Syntax 2:***

```
@ <expN1>,<expN2>, <expN3>,<expN4> [GUI]
     DRAW PIE ANGLE <expN5>,<expN6>
        [COLOR <color> | GUICOLOR <color>]
        [PRINTCOLOR <expC7>]
        [LINEWIDTH <expN8>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)
           ]
```

Purpose:

Draws circle or ellipse pie (i.e. closed and filled part of ellipse or circle) in GUI mode, specified by radius or bounding rectangle.

Arguments Syntax 1:

<expN1>, <expN2> are row/col or y/x coordinates of the circle center in specified or default <units>. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

RADIUS <expN3> is the circle radius in specified or default <units>. With row/cols unit, <expN3> is assumed as (fractional) number of columns.

ANGLE <expN5>,<expN6> are the start angle and the arc length in positive or negative degrees (-360..0..360). Positive values mean counter-clockwise while negative values mean the clockwise direction. Zero degree of <expN5> is at the 3 o'clock position, the 12 o'clock position is either 90 or -270. The arc length <expN6> is the drawn part of circle or ellipse in positive or negative degrees starting at <expN5>. The direction is clockwise when both <expN5> and <expN6> are positive or negative, or counter-clockwise otherwise, see also example below.

Arguments Syntax 2:

<expN1>, <expN2> are top left row/col or y/x coordinates of the bounding rectangle in specified or default <units>.

<expN3>, <expN4> are bottom right row/col or y/x coordinates of the bounding rectangle in specified or default <units>. If the bounding rectangle (calculated in pixels) is quadratic, circle pie is drawn.

ANGLE <expN5>,<expN6> are the start angle and the arc length in positive or negative degrees (or degree fractions), same as in Syntax 1 above.

Options:

COLOR <color> or **GUICOLOR <color>** is optional color specification. The circle or ellipse arc and closing lines are drawn by foreground color in width of<expN8> pixel, the pie area is filled by background. To draw and fill it in mono color, use the same color for foreground and background.

PRINTCOLOR <expC7> specifies the color for printing by SET GUIPRINT ON, or with PrintGui(.T.). The circle or ellipse arc is drawn by foreground color and the pie filled by background. If PRINTCOLOR is not given, GUICOLOR is used also for printer.

LINEWIDTH <expN8> is the line width of the circle or ellipse arc in pixels. If not given, line width of 1 pixel is used.

PIXEL : the <expN1>**..**<expN4> are values in pixel

NOPIXEL : the <expN1>..<expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN9> specifies unit for <expN1> .. <expN4> coordinates. The <expN9> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set(_SET_CO-ORD_UNIT,num). Apply for GUI mode only, ignored otherwise.

Description:

@...DRAW PIE draws parts of ellipse or circle specified by rounding rectangle or circle radius on screen and/or printer in GUI mode. It is often used to draw pie charts by using the same coordinates and different angles. This command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active).

The row() and col() values are set accordingly to draw end.

To draw full circle or ellipse, @..DRAW CIRCLE or @..DRAW ELLIPSE may be used instead. The @..DRAW ARC command is another alternative to draw part of circles or ellipses w/o filling the area.

Tuning:

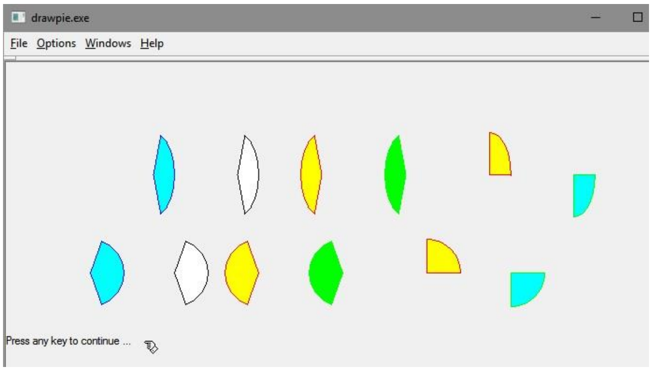
In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
_aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

Example:

```
@  5,15, 11,20 DRAW PIE ANGLE  -70, 140 GUICOLOR "B+/BG+" // <)
@  5,25, 11,30 DRAW PIE ANGLE   70,-140 GUICOLOR "N/W+"   // <)
@  5,35, 11,40 DRAW PIE ANGLE  110, 140 GUICOLOR "R+/RG+" // (>
@  5,45, 11,50 DRAW PIE ANGLE -110,-140 GUICOLOR "G+/G+"  // (>
@  5,55, 11,60 DRAW PIE ANGLE    0,  90 GUICOLOR "R+/GR+" // <) top
@  5,65, 11,70 DRAW PIE ANGLE  0, -90 GUICOLOR "G+/BG+" // <) bott
@ 15,10 DRAW PIE RADIUS 4 ANGLE  -70, 140 GUICOL "B+/BG+" // <)
@ 15,20 DRAW PIE RADIUS 4 ANGLE   70,-140 GUICOL "N/W+"   // <)
@ 15,30 DRAW PIE RADIUS 4 ANGLE  110, 140 GUICOL "R+/RG+" // (>
@ 15,40 DRAW PIE RADIUS 4 ANGLE -110,-140 GUICOL "G+/G+"  // (>
@ 15,50 DRAW PIE RADIUS 4 ANGLE    0,  90 GUICOL "R+/GR+" // <) top
@ 15,60 DRAW PIE RADIUS 4 ANGLE    0, -90 GUICOL "G+/BG+" // <) bot
```

Output:



Press any key to continue ...

Example:

```
See complete example in <FlagShip_dir>/examples/printergui.prg
```

Classification:

screen oriented output in GUI mode, GUI printout

Compatibility:

New in FS7, not available in Clipper nor in FoxPro.

Translation: *in std.fh*

GuiDrawPie(...)

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW ARC, @..DRAW LINES, @..DRAW POLYON, @..DRAW IMAGE, @..DRAW RECTANGLE, @...BOX, @...TO.., SET GUIPRINT, PrintGui()

@...DRAW POLYGON

Syntax:

```
@ [GUI] DRAW POLYGON <expA1>
        [CLOSED]
        [COLOR <color> | GUICOLOR <color>]
        [PRINTCOLOR <expC2>]
        [LINEWIDTH <expN3>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN4>)]
```

Purpose:

Draws (open or closed) polygon according to given array of coordinate pairs. Applicable in GUI mode only.

Arguments:

<expA1> is a two-dimensional array of coordinates in specified or default <units>, e.g. {{row,col},{row,col},{row,col},...}. At least two coordinate pairs are required.

Options:

CLOSED forces to close the polygon, i.e. the last point in <expA1> array is implicitly connected to the first point, and the polygon is filled by background color, if any.

COLOR <color> or **GUICOLOR <color>** is optional color specification. The polygon lines are drawn by foreground color in width of <expN6> pixel, and with CLOSED clause filled by background color. To draw the polygon in mono color, use the same color for foreground and background.

PRINTCOLOR <expC5> specifies the color for printing by SET GUIPRINT ON, or with PrintGui(.T.). The polygon is drawn by foreground and with CLOSED clause filled by background color. If PRINTCOLOR is not given, GUICOLOR is used also for printer.

LINEWIDTH <expN6> is the line width in pixels. If not specified, line width of 1 pixel is used.

PIXEL : the <expN1>, <expN2> are values in pixel

NOPIXEL : the <expN1>, <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN7> specifies unit for <expN1> .. <expN2> coordinates. The <expN7> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

Description:

@...DRAW POLYGON connects given coordinate points by lines; with the CLOSED clause, it also closes the polygon, and fills it by background color. It may also be used to draw line charts, see example. This command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active).

The row() and col() values are set accordingly to draw end.

You alternatively may use @..DRAW LINES to draw lines, it is similar to @...DRAW POLYGON without CLOSED clause.

Tuning:

In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
_aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

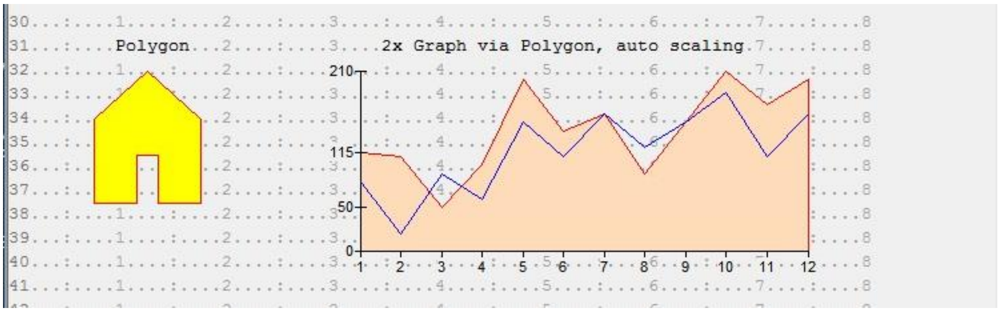
Example 1:

```
aCoord := {{34.5,8},{32.5,13},{34.5,18},{38,18},{38,14},{36,14}, ;
           {36,12},{38,12},{38,8}}
@ DRAW POLYGON (aCoord) GUICOLOR "R+/RG+" PRINTCOLOR "R+/RG+" ;
                        LINEWIDTH 2 CLOSED NOPIXEL
```

Example 2:

```
See complete example in <FlagShip_dir>/examples/printergui.prg with
body of diagram chart.
```

Output:



Classification:

screen oriented output in GUI mode, GUI printout

Compatibility:

New in VFS7, not available in Clipper nor in FoxPro.

Translation: *in std.fh*

GuiDrawPolygon(...)

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW ARC, @..DRAW PIE, @..DRAW LINES, @..DRAW IMAGE, @..DRAW RECTANGLE, @..BOX, @...TO.., SET GUIPRINT, PrintGui()

@...DRAW RECTANGLE

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4> [GUI]
        DRAW RECTANGLE
        [ROUNDED <expN5>]
        [COLOR <color> | GUICOLOR <color>]
        [PRINTCOLOR <expC6>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN7>)]
        [LINEWIDTH <expN8>]
```

Purpose:

Draws rectangle (in GUI mode) optionally rounded and filled by background color.

Arguments:

<expN1...expN4> are the coordinates, upper, left, lower, and right respectively, starting at 0. With the default row/col units, the coordinates specify mid of the character or line. If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

Options:

ROUNDED <expN5> is a rounding ratio (0..99) for the corners. Zero value draws angled corners, 99 is maximum roundedness.

COLOR <color> or **GUICOLOR <color>** is optional color specification. Rectangle lines (and corners) are drawn by foreground color in width of <expN8> pixel, the rectangle is filled by background color (if such given).

PRINTCOLOR <expC6> specifies the color for printing. If not given, GUICOLOR is used also for printer. Considered only in GUI mode when SET GUIPRINT is ON, or with PrintGui(.T.), and ignored otherwise.

PIXEL : the <expN1> .. <expN4> are values in pixel

NOPIXEL : the <expN1> .. <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN7> specifies unit for <expN1> .. <expN4> coordinates. The <expN7> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_CO-ORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

LINEWIDTH <expN8> is the line width in pixels. If not given, line width of 1 pixel is used.

Description:

@...DRAW RECTANGLE is an alternative command to @..BOX and supports also rounded corners. Applicable on screen and/or printer output in GUI mode. This command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui(.T.) is active).

Tuning:

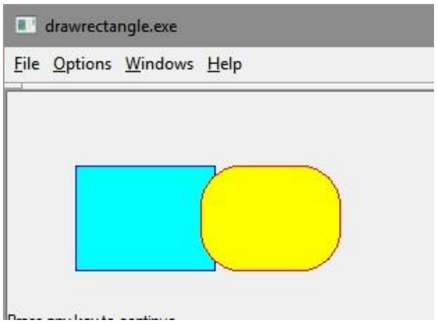
In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
_aGlobSetting[GSET_G_N_REFRESHDRAW] := -1 // default = 300 ms
```

Example 1:

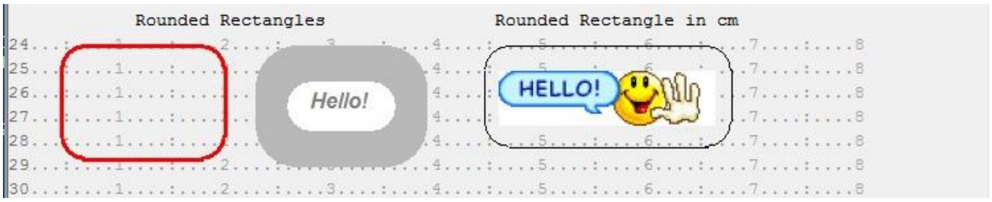
```
@  3, 5, 8, 16 DRAW RECTANGLE COLOR "B+/BG+" PRINTCOLOR "B/B+"
@  3,15, 8, 26 DRAW RECTANGLE ROUNDED 75 COLOR "R+/RG+"
```

Output:



Example 2:

```
See complete example in <FlagShip_dir>/examples/printergui.prg with
alternative rounded corners by given radius.
```



Classification:

screen oriented output in GUI mode, GUI printout

Compatibility:

New in FS7.

Translation:

GuiDrawRectangle(...)

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW ARC, @..DRAW PIE, @..DRAW POLYON, @..DRAW IMAGE, @..DRAW LINE. @...BOX, @...TO.., SET GUIPRINT, PrintGui()

@...PROMPT

Syntax:

```
@ <expN1>, <expN2>
     PROMPT <expC3>
        [MESSAGE <expC4>]
        [FONT <oFont>]
        [HEIGHT <nRows>]
        [WIDTH <nCols>]
        [CENTER]
        [COLOR <color>]
        [GUICOLOR <guicol>]
        [STYLE <naBox>]
        [LINEWIDTH <naPix>]
        [SELECT <block>]
        [TOOLTIP <cTip>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN6>)]
```

Purpose:

Defines menu prompts and their messages used in MENU TO and displays them on the screen.

Arguments:

<**expN1**> and <**expN2**> are the row and column where the prompt is displayed. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

<**expC3**> is the character string displayed in the menu. If null- string "" is given, an un-selectable item is generated. You may specify hot-key by prefacing the selected character by "&", "\&" or "\<". If you wish to display ampersand "&", add a space behind it. The hot-key is displayed underscored in GUI mode. In Terminal i/o mode, the hot-key it is displayed by using the 4th color pair of COLOR clause if such available, otherwise using inverse intensity of 1st (standard) color pair.

Options:

MESSAGE <**expC4**> is the character string displayed on the message line. If this option is specified, the message of the highlighted prompt is displayed on the line defined with SET MESSAGE. The screen section below the message is saved, and restored later at clear of the next MESSAGE, or manually by invoking _Message("") function. In GUI mode, the message is displayed in status bar, except the SET(_SET_MESSAGE_GUI, .T.) was set, see SET MESSAGE. In Terminal i/o mode, the message is displayed either in the status bar (if active) or in the SET MENU row otherwise. The <expC4> message can also be a code block which is evaluated at the time of MENU TO and must return a string to be displayed, otherwise no status bar message appears.

FONT <oFont> (GUI only) You may specify other than the default font e.g. @...PROMPT...FONT Font{"Helvetica",12}

HEIGHT <nRow> specifies the height (rows/pixel) of prompt (GUI mode only). The default height is one row. If HEIGH is not specified and the FONT clause is given the displayed height is increased to the font height, except the default or given height is sufficient to display the <expC3> text. The same apply if the given <nRow> is too small for the font.

WIDTH <nCol> specifies the width (chars/pixels) of the prompt text, which may be displayed centered when the **CENTER** clause is used. If WIDTH is specified and the size of text <expC3> exceeds <nCol> width, the text is cropped to fit in <nCol>.

COLOR <color> overwrites temporarily (for this PROMPT) the standard SET COLOR specification in Terminal i/o mode. The <color> parameter is a string containing at least two (standard,enhanced) color pairs. Apply also for GUI mode when SET GUICOLORS is ON and GUICOLOR clause is not specified.

GUICOLOR <guicol> specifies colors of this PROMPT in GUI mode. The <guicol> is either a string containing at least two (std,enh) color pairs, or ColorPair object or an array of RGB triplets. Considered in GUI mode, when the STYLE clause is also given.

STYLE <nBox> is either numeric expression or an array of two numer. elements specifying the frame around the prompt. When <nBox> is an array, the first element is the style of standard display, and 2nd element the style of selected prompt via MENU TO. When <nBox> is numeric, the same style is used for both menu states. For the <nBox> or {<nBox>,<nBox>} styles, use constants specified in box.fh:

BOX_NONE	0	display the prompt plain, w/o any frame
BOX_PLAIN	1	draw plain 2-d frame around the prompt
BOX_SUNKEN	2	draw sunken 3-d frame around the prompt
BOX_RAISED	3	draw raised 3-d frame around the prompt

When the STYLE clause is not given, or the <nBox> style is invalid, a standard button-alike prompt is used. STYLE is considered in GUI mode only, and ignored otherwise.

LINEWIDTH <nPix> is optional numeric value specifying the line width (in pixel) of a box drawn by the STYLE clause. The default value is 2. Same as with STYLE, you may specify <nPix> as array of two numeric elements for the standard and selected item. LINEWIDTH is ignored when STYLE is not used or when in other than GUI mode.

SELECT <block> is optional codeblock evaluated in MENU TO when the item was selected (by enter or mouse double-click or hotkey). The code block receives three parameters: <posOfSelItem>, <oMenuItem>, <oPrompt>. If the codeblock returns .F., MENU TO selection will be continued, otherwise MENU TO is terminated thereafter.

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the PROMPT item, even w/o focus

PIXEL : the <expN1>,<expN2> are values in pixel

NOPIXEL : the <expN1>,<expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN6> specifies unit for <expN1> .. <expN2> coordinates. The <expN6> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

Description:

A highlight bar menu is constructed in two stages. First the menu choices are painted on the screen using @...PROMPTs. Then, MENU TO may be used to activate the highlight-bar. The highlight-bar can be navigated by the cursor keys in the same order that the prompts were specified. In addition to, you may select an item via hotkey. In GUI mode also mouse left(double)click activates the prompt item. See details about navigation keys in MENU TO command.

Menu items can be specified in any order and configuration of row and column positions. MENU TO, however, navigates the current list of menu items in the order they were defined. After a choice is made, its sequence number is returned in the MENU TO variable.

After each @...PROMPT command, the cursor is placed one column position to the right of the last menu item character and ROW() and COL() are updated to reflect the new cursor position, so the next @ ROW(),COL() PROMPT.... aligns to previous. In GUI mode, the COL() position is set so, that also the next @ ROW(),COL() PROMPT.. aligns to previous one.

Colors: in Terminal i/o mode, either the supplied colors via COLOR clause is used, or the standard color otherwise. In GUI mode, the GUICOLOR clause is considered together with STYLE, if both are given. The standard, unselected @..PROMPT item is displayed by the 1st color pair, selected item in MENU TO by using 2nd color pair. Hot-keys, if specified, are underscored (in GUI mode) or displayed in Terminal i/o by using the 4th color pair. Unselectable items are displayed by the 5th color pair. See SET COLOR for further details.

Nesting: FlagShip supports nested @...PROMPT / MENU TO, triggered either by SET KEY TO <myUdf> or by SELECT <block> clause. The only pre-requirement is, you declare either LOCAL _oPrompt [AS Usual] or PRIVATE _oPrompt := NIL variable which then automatically hold the nested prompts. If _oPrompt is not explicitly declared, internally declared PUBLIC _oPrompt variable is used otherwise.

Selection: The selection of @..PROMPT items is handled by MENU TO command. Refer there for further information about navigation and supported keys. If you wish to clear all @..PROMPT items without invoking MENU TO, use the CLEAR MENU command or _oPrompt:Clear()

The Prompt class is used internally for @..PROMPT items and MENU TO processing, the object is hold in _oPrompt. See also menuclass.fh

Example 1:

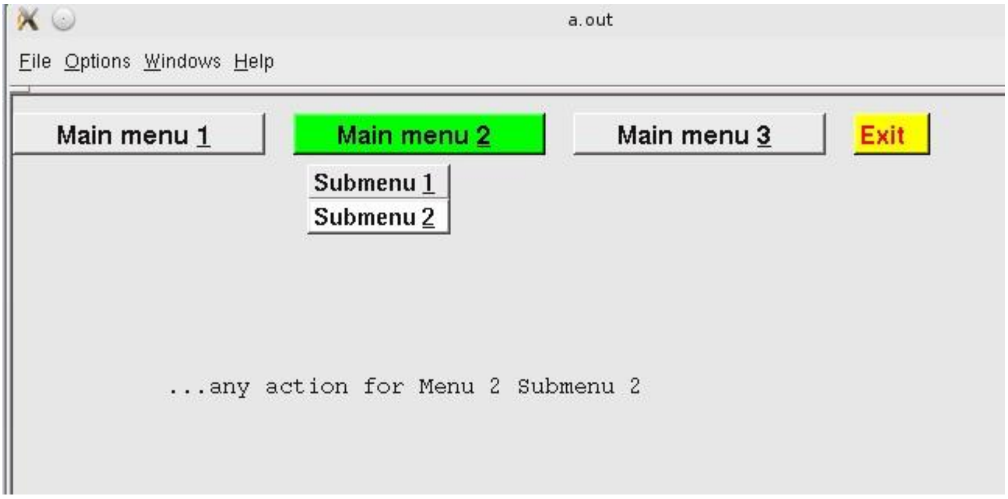
```
SET MESSAGE TO (MAXROW())
@ 10,20 PROMPT "One"
@ 12,20 PROMPT "Two" MESSAGE "Help message for option Two"
@ 14,20 PROMPT "Three"
MENU TO choice
```

Example 2:

Build an SAA look-alike menu, using un-nested @..PROMPT/MENU TO:

```
PRIVATE choice1 := 1, choice2 := 0
set font to "Courier", 10              // default font
oApplic:Resize(25,80,,.T.)             // resize window acc.to font
oFont1 := FontNew("Arial",12,"B")      // the PROMPT font
oFont2 := FontNew("Arial",10,"B")      // second PROMPT font

DO WHILE .T.
   @ 0, 0   CLEAR
   @ 0.5, 0 PROMPT "Main menu &1" FONT (oFont1) WIDTH 18 CENTER
   @ 0.5,20 PROMPT "Main menu &2" FONT (oFont1) WIDTH 18 CENTER  ;
                      GUICOLOR "#FFFFFF/#005F00,#000000/#00FF00" ;
                      STYLE BOX_RAISED
   @ 0.5,40 PROMPT "Main menu &3" FONT (oFont1) WIDTH 18 CENTER
   @ 0.5,60 PROMPT "Exit " FONT (oFont1) ;
                           GUICOLOR "R+/GR+,GR+/R+" ;
                           STYLE BOX_RAISED
   MENU TO choice1                    // process horizontal menu

   SET KEY K_RIGHT TO my_right_left   // redirect CuR passed to UDP
   SET KEY K_LEFT  TO my_right_left   // redirect CuL passed to UDP
   choice2 := 0
   DO WHILE choice1 > 0 .AND. choice1 < 5
      DO CASE
      CASE choice1 = 1
         @ 2,1 PROMPT "1.text" FONT (oFont2) // choice1=1,choice2=1
         @ 3,1 PROMPT "2.text" FONT (oFont2) // choice1=1,choice2=2
         @ 4,1 PROMPT "3.text" FONT (oFont2) // choice1=1,choice2=3
      CASE choice1 = 2
         @ 2,21 PROMPT "Submenu &1" FONT (oFont2)
         @ 3,21 PROMPT "Submenu &2" FONT (oFont2)
      CASE choice1 = 3
         @ 2,41 PROMPT "text"
         @ 3,41 PROMPT "other text"
      CASE choice1 = 4
         if alert("Really exit menu?", {"Yes","No"}) == 1
            quit
         endif
         choice1 := 0
      ENDCASE
      MENU TO choice2                   // process vertical menu
      IF choice2 == 0 .OR. LASTKEY() == K_ESC ;
            .OR. LASTKEY() == K_PGDN
         EXIT                        // exit the submenu
      ENDIF
      IF choice1 == 1
         @ 8, 11 SAY "...any action for Menu 1, item " + ;
                     ltrim(choice2) + ".text"
      ELSEIF choice1 == 2
         @ 8, 11 SAY "...any action for Menu 2 Submenu " + ;
                    ltrim(choice2)
      ELSEIF choice1 == 3
         @ 8, 11 SAY "...any action for Menu 3, item#" + ;
                    ltrim(choice2)
      ENDIF
   ENDDO
   SET KEY K_RIGHT TO
   SET KEY K_LEFT  TO                // disable CuR/CuL redirection
   IF LASTKEY() == K_ESC             // K_ESC = 27 = exit
      EXIT
   ENDIF
ENDDO
setpos(10,0)
? "choice1=",choice1,"choice2=",choice2
wait "done..."

PROCEDURE my_right_left (p1, p2, p3)  // Cursor left or right
if choice2 > 0
  @ 2,0 CLEAR
  choice2 := 0
  choice1 := IF(LASTKEY()==4,if(choice1 >= 4,4,choice1+1), ;
                             if(choice1 <= 1,1,choice1-1) )
  KEYBOARD chr(K_PGDN)                 // PgDn = 3 = leave Submenu
endif
RETURN
```

Output:



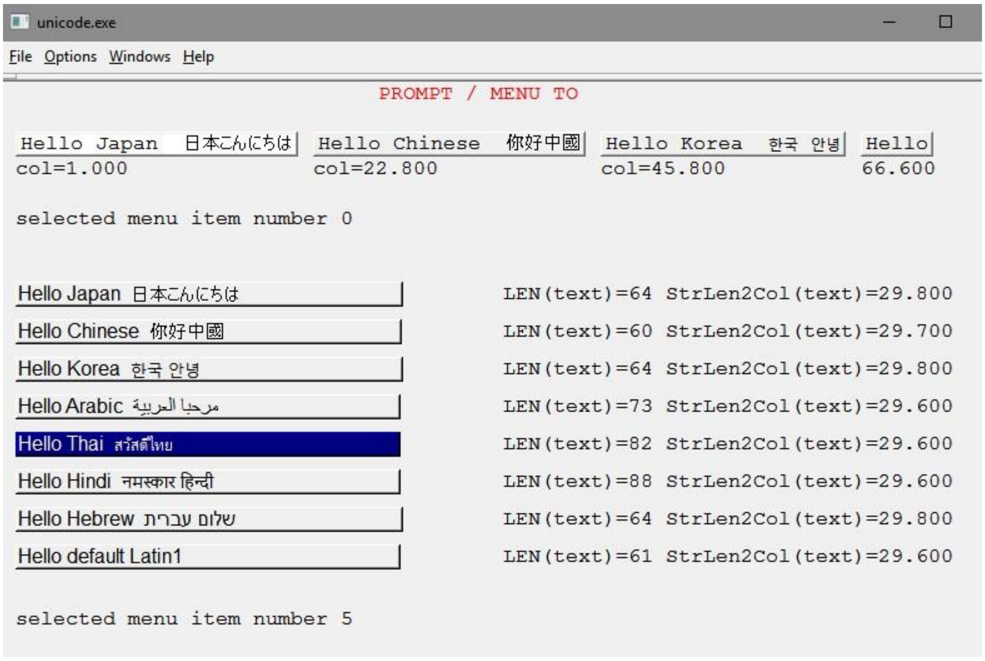
```
...any action for Menu 2 Submenu 2
```

Example 3:

For using the HEIGH, WIDTH, COLOR, GUICOLOR, STYLE, SELECT clauses and nesting, see complete example in .../examples/menu_prompt.prg

Example 4:

With Unicode font, you may use any language (also Asian, Arabic, Chinese etc.), see <FlagShip_dir>/examples/unicode.prg



```
                         PROMPT / MENU TO

 Hello Japan  日本こんにちは  Hello Chinese  你好中國  Hello Korea  한국 안녕  Hello
 col=1.000               col=22.800               col=45.800           66.600

 selected menu item number 0


 Hello Japan 日本こんにちは        LEN(text)=64 StrLen2Col(text)=29.800
 Hello Chinese 你好中國            LEN(text)=60 StrLen2Col(text)=29.700
 Hello Korea 한국 안녕              LEN(text)=64 StrLen2Col(text)=29.800
 Hello Arabic مرحبا العربية        LEN(text)=73 StrLen2Col(text)=29.600
 Hello Thai สวัสดีไทย               LEN(text)=82 StrLen2Col(text)=29.600
 Hello Hindi नमस्कार हिन्दी           LEN(text)=88 StrLen2Col(text)=29.600
 Hello Hebrew שלום עברית           LEN(text)=64 StrLen2Col(text)=29.800
 Hello default Latin1              LEN(text)=61 StrLen2Col(text)=29.600

 selected menu item number 5
```

Classification:

screen oriented input/output

Compatibility:

Unlimited PROMPTs for one MENU TO are supported in FlagShip, up to 32 in Clipper, which also does not support nesting.

Translation:

__ATPROMP2 (@oPrompt , expN1, expN2, expC3 [, expC4] , . . .)

old syntax (FS4.48 and VFS up to 5.1.4), w/o nesting:

__ATPROMPT (expN1, expN2, expC3 [, expC4])

Related:

MENU TO, CLEAR MENU, SET MESSAGE, SET WRAP, Achoice(), PushButton()

@...SAY

Syntax:

```
@ <expN1>,<expN2>
     SAY <exp3>
        [PICTURE <expC4>]
        [COLOR <expC5>]
        [GUICOLOR <expC5>]
        [PRINTCOLOR <expC6>]
        [FONT <expC7>, <expN8> | FONT <expO9>]
        [PIXEL|NOPIXEL]
        [SPLIT | COLUMN [<expN10>,<expN11>]]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN12>)]
```

Purpose:

Displays data at the specified row and column positions according to an optional picture format on screen or printer.

Arguments:

<**expN1**>, <**expN2**> are numeric expressions for positioning data at specific row and column coordinates (0..24 and 0..79 for 25x80 screen or MAXROW() and MAXCOL() respectively). In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

<**exp3**> is evaluated by SAY and the result of a character, date, logical, or numeric expression is shown on the display (or the current DEVICE).

Options:

COLOR <expC5> specifies the color in which to display <exp3>. Only the first color pair (standard) is significant. If this clause is not given, the current color setting is used. In GUI mode, first the GUICOLOR is checked if set. If not, the COLOR <expC5> 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 i/o mode w/o switching the SET GUICOLOR setting.

GUICOLOR <expC5> specifies the color for the <exp3> data display considered in GUI mode, where only the first color pair (standard) is significant. Instead of string, you also may use RGB triplets (or stringified triplets), see SET COLOR for details, and example below. If GUICOLOR is set, it is used regardless the current SET GUICOLOR on/off setting. If omitted and SET GUICOLOR is ON, either the COLOR <expC5> is used if given, or the current SetColor() is used. The GUICOLOR clause apply for GUI mode only, and is ignored otherwise.

PRINTCOLOR <expC6> specifies the color for printing. If not given, GUICOLOR is used also for printer. Considered only in GUI mode when SET GUIPRINT is ON, or with PrintGui(.T.), and ignored otherwise.

FONT <expC7>, <expN8> (GUI only) You may specify other than the default font e.g. @...SAY...FONT "Arial",12

FONT <expO9> (GUI only) This is alternative font specification. The <expO9> is already instantiated font object, which allows also setting of font attributes like bold, underscore, italic and so on.

PICTURE <expC4> gives formatting rules for outputting <exp3>. When no PICTURE is given, the format is determined by analyzing the value of <exp3>.

PIXEL : the <expN1>, <expN2> are values in pixel

NOPIXEL : the <expN1>, <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

SPLIT will split long string into two or more lines. The available size is calculated from current Col() position up to MaxCol() for current line and MaxCol() -1 for subsequent lines. If PrintGui(.T.) is active or SET GUIPRINT is ON, oPrinter:GuiMaxCol() is used instead. The string is splitted at the left next space or tab or dash if any. You may add conditional split position (separators) by chr(1) or chr(247), which are then interpreted as dash at line end and ignored otherwise.

COLUMN <expN10>,<expN11> or **SPLIT <expN10>,<expN11>** is similar to SPLIT, but instead of full line, it will split large text column- wise, from column <expN5> to <expN6> (in row/cols or units). Note that FONT <expC7>,<expN8> is not accepted here, only FONT <expO9>. See example in <FlagShip_dir>/examples/printergui.prg

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN12> specifies unit for <expN1>...<expN2> and <expN10>,<expN11> coordinates. The <expN6> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_MM or UNIT_ROWCOL etc). If the UNIT=.. clause is not specified, default is row/col, or the current setting by SET(_SET_PIXEL, logVal) or SET(_SET_COORD_UNIT, numVal). Apply for GUI mode only, ignored otherwise.

Description:

SAY <exp3> displays the result of the expression at the given coordinates on the current device (printer, screen). The output obeys the optional PICTURE formatting.

SAY uses the "standard" pair from the current or given SET COLOR string. See also SETENHANCED and SETUNSELECTED commands to use another color pair.

By default, the output is directed to the screen, but if SET DEVICE TO PRINT is specified, the output is re-directed to the printer. To direct the output to a text file, use SET PRINTER TO <file_name> followed by SET DEVICE TO PRINT. Unlike console commands (like ? and ??), the @...SAY output to the printer is not echoed to the screen and SET CONSOLE has no effect on SAY output.

When using SAY to produce printer output, care must be taken to proceed sequentially from top to bottom. An EJECT is performed if the current row is less than the last position printed. If the column is greater than the previous one (including the SET MARGIN), BACKSPACEs are sent to reposition the printer head. To override such default repositioning, SETPRC() can be used to define a new logical "printer head" position. You may tune the printer device driver by FS_SET("prset").

After @..SAY output, the cursor is left one column position to the right of the last character displayed. ROW() and COL() (or PROW() and PCOL() respectively) are then updated with this position. Note that when different FONT is used, the COL() is adapted automatically to a larger/smaller font size but the ROW() only when SET ROWADAPT is ON (default is OFF). You also may force the adaption manually by invoking RowAdapt() thereafter. To align an output using different fonts on the same base line, use SET ROWALIGN BASELINE.

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 (e.g. SET FONT "Courier", 12).

For additional hints how to manage proportional fonts, see further details in LNG.5.3, LNG.5.4, Col2pixel(), Row2pixel(), SET FONT, SET ROWALIGN, SET ROWADAPT.

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.

In GUI mode, you may display PC-8 special symbols chr(1..31) by any character set when SET GUITRANSL LOWCP437 ON is set. Another CP-437 characters may be transferred to Unicode via CP437_UTF8(), see also Unicode below.

The @..SAY command is processed also for GUI/GDI printout (when SET GUIPRINT ON or PrintGui() is active) and accepts PRINTCOLOR, PICTURE, FONT, PIXEL and UNIT= clauses, other are ignored.

Picture:

<expC4>, the PICTURE clause, is a string and consists of two optional parts, the FUNCTION and the TEMPLATE, separated by at least one blank when both are present. Functions apply to the entire <exp3> while templates mask corresponding characters of <exp3>. Function and template symbols are not case sensitive.

The FUNCTION part, when given, must precede the template and start with the "@" sign. All the symbols which follow the first blank are interpreted as functions. The rest is taken as TEMPLATE. In the absence of the "@", the whole string is considered a template.

Picture FUNCTIONS are applied to the entire SAY <exp>. Multiple function definitions are allowed. Characters not belonging to the TEMPLATE symbol set overwrite existing characters of the <exp>. The "@R" function enables the insertion instead of the overwriting of non-template characters.

If you set `FS_SET`("devel", .T.), `PICTURE` problems and fixes are displayed as developer's warning.

Picture FUNCTION in the `PICTURE "@..."` part. For this @..SAY command, the SAY or S/G mode apply:

Func	Type	Mode	Definition
A	C	GET	in SAY: same as 'X' template
B	N	SAY	Numbers are displayed left-justified
C	N	SAY	'CR' for credit is displayed after positive numbers
D	D	S/G	Dates are displayed in the SET DATE format
E	D	S/G	Dates are displayed in European format (day and month are exchanged)
E	N	S/G	Numerics are displayed in European format (comma & period are exchanged)
K	all	GET	GET is cleared if the first key is not a cursor or Insert key
P	C	GET	Password: display '*' instead of text
R	C	S/G	Non-template characters from the TEMPLATE part of picture are inserted during in/output but removed from the value
Sn	C	S/G	Horizontal scrolling within a GET window of <n> columns is allowed, SAY displays only the first <n> characters
X	N	SAY	'DB' for debit is displayed after negative numbers
Z	ND	S/G	Leading zeros are displayed as blanks
(	N	SAY	Negative numbers are enclosed in parentheses with leading spaces
)	N	SAY	Negative numbers are enclosed in parentheses without leading spaces
!	C	S/G	Alphabetic characters are converted to uppercase
F	N	SAY	fill leading spaces with stars "*"
T	all	SAY	remove leading and trailing spaces

Picture TEMPLATE Symbols. For this @..SAY command, the SAY or S/G mode apply:

TEM	Type	Mode	Definition
X	C	S/G	Any character is accepted
A	C	GET	in SAY: same as 'X' template
B	C	GET	in SAY: same as 'X' template
N	C	GET	in SAY: same as 'X' template
9	CND	S/G	Digits for any data type including the sign for numeric are displayed
#	CND	S/G	Digits, signs and spaces for any data type are displayed
L	L	S/G	The logical "T" or "F" are displayed
Y	CL	S/G	Only "Y" or "N" are allowed
!	all	S/G	An alphabetic character is converted to uppercase
$	N	SAY	The Dollar sign $ is displayed in place of a leading space in a numeric
*	N	SAY	The asterisk is displayed in place of a leading space in a numeric
.	N	S/G	The period defines the decimal point position, regardless of the given @E conversion
,	N	S/G	The comma defines the 'thousands' comma position, regardless of the given @E conversion

Unicode:

In GUI mode, FlagShip supports also Unicode (UTF-8 and UTF-16). If the <expO7> font is set to Unicode by expO7:CharSet(FONT_UNICODE) or globally by `oApplic:Font:CharSet(FONT_UNICODE)` or SET GUICHARSET FONT_UNICODE, Unicode glyphs displays e.g. for Asian languages. Predefined strings needs to be stored in UTF-8 encoding, or transformed from UTF-16 by Utf16_Utf8(). In Linux, you may need to set Unicode font, e.g. SET FONT "mincho" as well. Since glyphs usually uses multiple bytes chr(128..255), it is recommended not to use SET GUITRANSLATE TEXT ON to draw semi-graphics; you may convert PC-8 ASCII characters to Unicode by Cp437_utf8(). See also example in <FlagShip_dir>/examples/unicode.prg

Note that PICTURE cannot be considered with Unicode, except the "@S" in FUNCTION part.

Tuning:

In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
_aGlobSetting[GSET_G_N_REFRESHDRAW] := -1    // default = 300 ms
```

If you write to row or column not currently visible on the screen, you may force an automatic horizontal/vertical scroll by assigning

```
_aGlobSetting[GSET_G_L_ROW_VISIBLE ] := .T.  // default = .T.
_aGlobSetting[GSET_G_L_COL_VISIBLE ] := .T.  // default = .F.
```

In GUI mode, when scrollbars are enabled or set to auto (default), vertical and/or horizontal scrollbar displays when current screen output exceeds the visible window size. Max scrollbar area is set to 2000 pixels by default (approx. 100 lines * 200 chars, depends on current font), but may be changed by assigning e.g.

```
_aGlobSetting[GSET_G_N_MAXSIZE_ROW] := row2pixel(500)
_aGlobSetting[GSET_G_N_MAXSIZE_COL] := col2pixel(300)
```

The minimal scroll area size is 100 pixel, maximal 32100 pixel (it is automatically fixed). See also 3rd example in section CMD ?,??.

In Terminal i/o mode, there are sometimes special characters like arrows and boxes/lines not displayed correctly. You may emulate arrows chr(24,25,26,27) and/or chr(16,17,30,31) by setting

```
_aGlobSetting[GSET_T_N_EMUL_ARROWS] := num  // default = 0
```

where <num>	= 0	uses default display by font/mapping
	= 1	emulates arrows chr(24,25,26,27) by ASCII ^ v > <
	= 2	emulates arrows chr(24,25,26,27) by curses
	+ 16	emulates also chr(16,17,30,31) by (1) or (2)

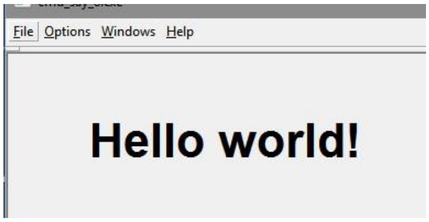
To emulate boxes and lines chr(179 to 218) in Terminal i/o, use

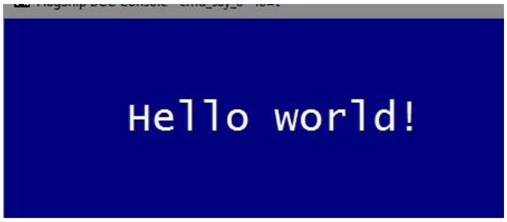
```
_aGlobSetting[GSET_T_N_EMUL_BOXES] := num  // default = 0
```

where <num>	= 0	uses default display by font/mapping
	= 1	emulates boxes chr(179..218) by ASCII + - |
	= 2	emulates boxes chr(179..218) by curses

Example 1:

```
SET FONT "Arial",16 BOLD          // here: GUI mode only
SET COLOR TO "W+/B"               // here: Terminal i/o only
oApplic:Resize(25,80,,.T.)        // application size accord. to font
CLS
@ 2,5 say "Hello world!"
inkey(0)                          // wait for keypress
```





Example 2:

The string is shown in the 11th row and the 5th column. Non- template characters will overwrite a part of the value string if no @R function is used.

```
value = "Peter"
@ 11,5 SAY value                          // Result: Peter
@ 11,5 SAY value PICTURE "@!"             // Result: PETER
@ 11,5 SAY value PICTURE "xx-x"           // Result: Pe-e
@ 11,5 SAY value PICT "@! xxtxx"          // Result: PEtER
@ 11,5 SAY value PICT "@R! xx xxx"        // Result: PE TER
```

```
value1 = "long pict"
value2 = "long string"
@ 5,5 SAY value1 PICT "xxxxxxxxxxxxxxx"  // "long pict"
@ 6,5 SAY value2 PICT "XXXX"             // "long"
```

Example 3:

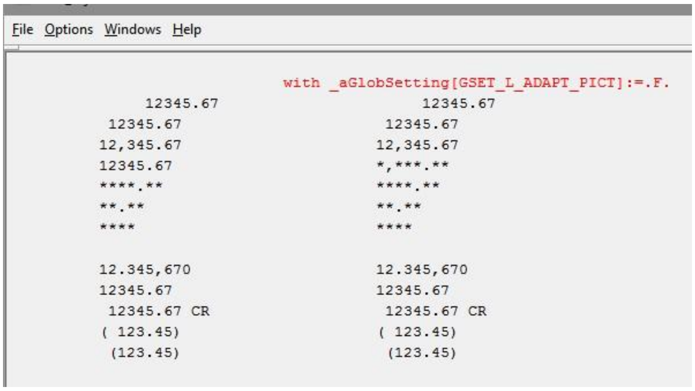
With an 80-column terminal, the string value will be shown on screen as depicted, starting from 11th row and 77th column.

```
       value = "abcdefgh"                          // Result : -----┐
       @ 11,77 SAY value                           // line 11:   abc|
                                                   // line 12:   def|
                                                   // line 13:   gh |
```

Example 4:

Numbers are converted by default, according to the SET DEFAULT or the current field length, or by using the PICTURE clause

```
set font to "Courier", 10             // set default font
oApplic:Resize(25,80,,.T.)            // resize window acc.to font

col := 10
value  = 12345.67
negval = -123.45
@ 2,col SAY value                       //      12345.67
@ 3,col SAY value PICT "999999.99"      //  12345.67
@ 4,col SAY value PICT "99,999.99"      // 12,345.67
@ 5,col SAY value PICT "9,999.99"       // 12345.67 <-note!
@ 6,col SAY value PICT "9999.99"        // ****.**  <-note!
@ 7,col SAY value PICT "99.99"          // **.**    <-note!
@ 8,col SAY value PICT "9999"           // ****     <-note!

@ 10,col SAY value PICT "@E 99,999.999"   // 12.345,670
@ 11,col SAY value PICT "@B 99999999.99"  // 12345.67
@ 12,col SAY value PICT "@C 999999.99"    //   12345.67 CR
@ 13,col SAY negval PICT "@( 99999.99"    // (  123.45)
@ 14,col SAY negval PICT "@) 99999.99"    // (123.45)
```



```
                        with _aGlobSetting[GSET_L_ADAPT_PICT]:=.F.
      12345.67                        12345.67
  12345.67                        12345.67
 12,345.67                       12,345.67
 12345.67                        *,***.**
 ****.**                         ****.**
 **.**                           **.**
 ****                            ****

 12.345,670                      12.345,670
 12345.67                        12345.67
  12345.67 CR                     12345.67 CR
 ( 123.45)                       ( 123.45)
  (123.45)                        (123.45)
```

Example 5:

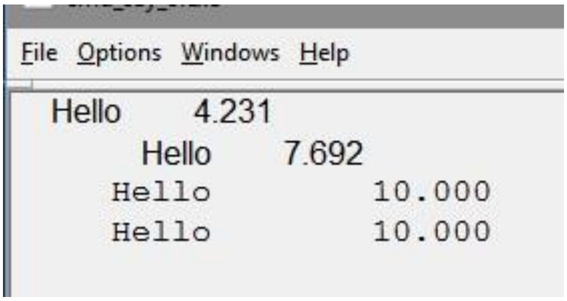
The date may be displayed and entered according to PICTURE and/or settings by SET DATE and SET CENTURY

```
value = CTOD("12/31/93")
@ 14,10 SAY value                         // 12/31/93
@ 14,10 SAY value PICTURE "@D"            // 12/31/93
@ 14,10 SAY value PICTURE "@E"            // 31/12/93
@ 14,10 SAY value PICTURE "@E 99.99.99"   // 31.12.93
SET CENTURY ON
@ 14,10 SAY value PICTURE "@E"            // 31/12/1993
SET DATE USA
@ 14,10 SAY value PICTURE "@D"            // 12-31-1993
SET DATE GERMAN
@ 14,10 SAY value                         // 31.12.1993
@ 14,10 SAY value PICTURE "@E"            // 12.31.1993
```

Example 6:

The selected row and column depends on the used font and pitch (fixed or proportional):

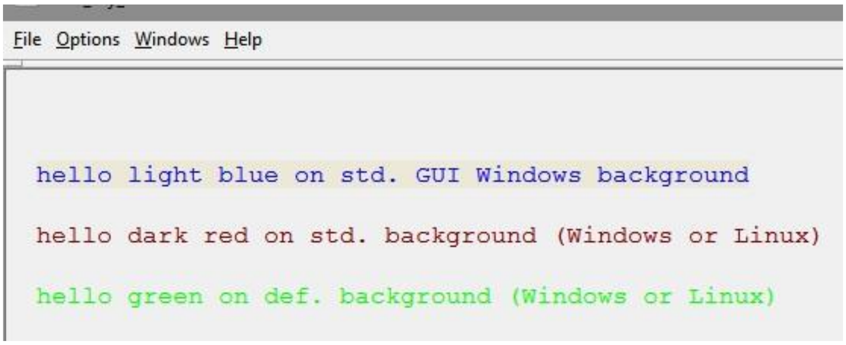
```
SET FONT "Helvetica", 12
@ 0,0 say "     Hello" ; ?? COL()       // text columns differs
@ 1,5 say "Hello" ; ?? COL()            // with proportional font
SET FONT "Courier", 12
@ 2,0 say "     Hello" ; ?? COL()       // text columns aligns
@ 3,5 say "Hello" ; ?? COL()            // with fixed font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



```
Hello      4.231
      Hello      7.692
    Hello          10.000
    Hello          10.000
```

Example 7:

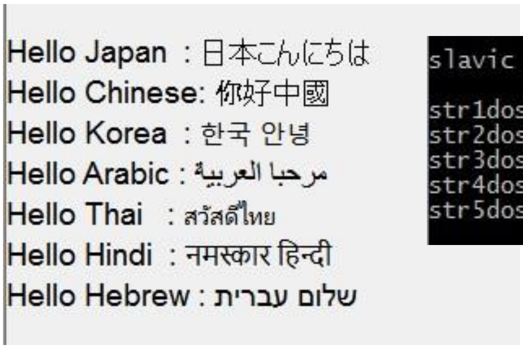
```
#include "color.fh"
set font "Courier", 12
set color to "N/W+" ; CLS                          // for terminal i/o
@ 3,2 SAY "hello light blue on std. GUI Windows background" ;
  COLOR "B+/W+" ;                                  // Terminal mode
  GUICOLOR {{0,0,255},{RGBCOLOR_BG_WINDOWS}}   // GUI mode

@ 5,2 SAY "hello dark red on std. background (Windows or Linux)" ;
  GUICOLOR ("R/" + RGBSTRING_BG) COLOR ("R/W+")

@ 7,2 SAY "hello green on def. background (Windows or Linux)" ;
  GUICOLOR "G+/?" COLOR ("G/?")
```



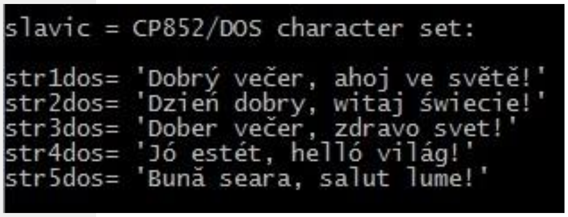
```
hello light blue on std. GUI Windows background

hello dark red on std. background (Windows or Linux)

hello green on def. background (Windows or Linux)
```

Example 8:

With Unicode font or setting proper character set, you may use any language (also Asian, Arabic, Chinese, Slavic, Greek etc.), see <FlagShip_dir>/examples/unicode.prg, slavic.prg, western.prg, greek.prg, arabic.prg



```
Hello Japan  : 日本こんにちは
Hello Chinese: 你好中國
Hello Korea  : 한국 안녕
Hello Arabic : مرحبا العربية
Hello Thai   : สวัสดีไทย
Hello Hindi  : नमस्कार हिन्दी
Hello Hebrew : שלום עברית
```



```
slavic = CP852/DOS character set:

str1dos= 'Dobrý večer, ahoj ve světě!'
str2dos= 'Dzień dobry, witaj świecie!'
str3dos= 'Dober večer, zdravo svet!'
str4dos= 'Jó estét, helló világ!'
str5dos= 'Bună seara, salut lume!'
```

Classification:

screen oriented output (SET DEVICE TO SCREEN), coordinates oriented, printer/file output via SET DEVICE TO PRINT or via PrintGui()

Compatibility and Tuning:

Clipper ignores illegal PICTURE characters, FlagShip reports them in developer mode i.e. with `FS_SET("devel",.T.)`. FlagShip tries to correct them: if no space is given between the PICTURE function and template, it will be corrected by inserting the whitespace character. The leading and trailing spaces in the PICTURE definition will be truncated.

FlagShip does not cut off the most significant digits of numeric output within short pictures, it tries, if possible, to output the whole number by removing inserted chars or by shortening the PICTURE deci part containing zeros. To disable this feature, and to display stars instead, assign `_aGlobSetting[GSET_L_ADAPT_PICT] := .F.`

When using a numeric PICTURE which does not display all decimal digits stored in the variable, FlagShip cuts the remaining decimal digits like all other Xbase dialects, but unlike Clipper which rounds it. For example, the statement

```
  @ y,x SAY 1234.567 PICT "9999.99"
```

will display 1234.57 in Clipper 5.2, but 1234.56 in FlagShip and other Xbase dialects. For fully compatible output to Clipper 5.2, use e.g.

```
num = ROUND(num,2); @..GET num PICTURE "99.99"
```

or

```
@..SAY ROUND(num,2) PICTURE "99.99" etc.
```

Similarly, to strings: if the template PICTURE characters does not match the string length, the template is automatically extended by "X" instead of truncating the output (like Clipper illegally do). This means in generally: FlagShip does not modify the output variable length, but only the PICTURE template, if required.

If the line number <expN1> is out of range, FlagShip displays the text at the first or last line available, according to C87. When the text is longer than the available column, and the "@S" picture is not specified, the rest will be continued in subsequent lines, see example above.

The physical output on the screen depends on the chosen terminal emulation (environment variable TERM), the ability of the terminal to display the required graphical characters, and the output mapping defined in the file FSchrmap.def.

In contrast to DOS, the color capability and the size of the screen is not fixed to 80x25, but depends on the current terminal used (environment variable TERM) and the terminal description in the terminfo file, e.g. FStinfo.src. If possible, use one of the extended terminal descriptions FSxxx, see (REL) Predefined Terminals.

In GUI mode, the @..SAY cannot overwrite widgets located in higher layer, see also LNG.5.3. You may use @ r,c CLEAR TO r2,c2 to clear the GET widget(s) when READ finishes.

Embedded zero bytes are not supported, since cannot be displayed.

Note: because some older Unix terminals allow only 24 instead of 25 lines to be used, use `@ MAXROW(),x SAY...` instead of `@ 24,x SAY...` for programs running on different terminals. See also LNG.5.1, section SYS, and FS_SET ("outmap").

Translation:

```
DEVPOS (expN1, expN2) ; DEVOUT (exp3 [, expC5] )
DEVPOS (expN1, expN2) ; DEVOUTPICT (exp3, expC4 [, expC5] )
```

Related:

?/??, @...GET, @...TO, @...CLEAR, CLEAR, SET DEVICE, COL(), ROW(), FS_SET("devel"), FS_SET("term"), FS_SET("prset"), PCOL(), PROW(), SETPRC(), SETSTANDARD, SETENHANCED, SETUNSELECTED, SET HTMLTEXT, SET ROWADAPT, SET ROWALIGN, SET GUIPRINT, PRINTGUI(), description in LNG.5.1, LNG.5.3 and sections REL, SYS.

@...SAY BITMAP
@...SAY IMAGE

Syntax 1:

```
@ <expN1>, <expN2>, [<expN3>], [<expN4>]
     SAY IMAGE|BITMAP [FROM] FILE <expC6>
        [SCALE] [CLIP|NOSCALE]
        [IMGTYPE <expC7>]
        [BORDER|FRAME <expN8>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)]
```

Syntax 2:

```
@ <expN1>, <expN2>, [<expN3>], [<expN4>]
     SAY IMAGE|BITMAP [USING] <expC5>
        [SCALE] [CLIP|NOSCALE]
        [IMGTYPE <expC7>]
        [BORDER|FRAME <expN8>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN9>)]
```

Purpose:

Display bitmap image at specified screen position. This command is equivalent to @...DRAW IMAGE command. It is considered also by GUI printout (when SET GUIPRINT ON or PrintGui(.T.) is active). Applicable in GUI mode only, ignored otherwise.

Syntax 1 reads the image from file,

Syntax 2 uses image data stored in database or character variable.

Arguments:

<**expN1**>, <**expN2**> are numeric expressions, specifying the row and column coordinates (i.e. the the top left corner) where the image is displayed. You may use numeric values with decimal fractions for row and column. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

<**expN3**> is optional numeric value, specifying the bottom line (or Y pixel value) bounding the image height. If not given, the image height is used.

<**expN4**> is optional numeric value, specifying the right column (or X pixel value) bounding the image width. If not given, the image width is used.

[**USING**] <**expC5**> is the character variable or field containing the image data displayed according to Syntax 1.

[**FROM**] **FILE** <**expC6**> is the image file name (optionally with path) displayed according to Syntax 1, optionally with path. The file name extension is not relevant; FlagShip determines the image type from the data self, or by considering the IMGTYPE clause.

CLIP or **NOSCALE** will clip the image at bottom and/or right if it does not fit into the specified rectangle <expN1>...<expN4>. If neither CLIP nor NOSCALE was specified, SCALE is the default.

SCALE will scale the image to fit into the specified rectangle <expN1>...<expN4>. If <expN3> is not given, the image is scaled to fit into width of <expN4> - <expN2>. If <expN4> is not given, the image is scaled to fit into height of <expN3> - <expN1>. If neither <expN3> nor <expN4> was specified, the image is displayed as is.

IMGTYPE <**expC7**> is optional image specification. If not given, FlagShip reads few bytes of the image to determine the image type. You may override this pre-scan by specifying <expC7>:

"BMP"	(Windows Bitmap) is uncompressed image format common on MS-Windows
"GIF"	(Graphic Interchange Format) is compressed lossless image format used often for Web images. Note: GIF format uses LZW compression patented by Unisys and needs to be licensed (by Unisys) in some countries (alternative is PNG format).
"JPEG"	(Joint Photographic Experts Group) is a compressed lossy image format that gives high compression for real-world and photo-realistic images.
"PNG"	(Portable Network Graphics) is compressed lossless image format, offering almost better compression than JPEG, used also as patent-free replacement of GIF or TIFF images.
"PPM"	(Portable PixMap) is uncompressed image format common on Unix, offering few advantages over PNG or JPEG
"XBM"	(X11 BitMap) is uncompressed monochrome image format.
"XPM"	(X11 PixMap) is uncompressed image format, which can be trivially included in source files as they are C code.

BORDER <**expN8**> or **FRAME** <**expN8**> specifies optional frame around the image, where the <expN8> is a constant specified in box.fh:

BOX_NONE	0	don't draw any border around image (default)
BOX_PLAIN	1	draw plain 2-d frame around the image
BOX_SUNKEN	2	draw sunken 3-d frame around the image
BOX_RAISED	3	draw raised 3-d frame around the image

PIXEL : the <expN1> ... <expN4> are values in pixel

NOPIXEL : the <expN1> ... <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified, the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN9> specifies unit for <expN1> .. <expN4> coordinates. The <expN9> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set(_SET_CO-ORD_UNIT,num). Apply for GUI mode only, ignored otherwise.

Description:

This command is supported in GUI mode and ignored otherwise. It displays bitmap image at specified position, optionally scaled or circumcised and/or surrounded by a frame. The image is either read from any named file (when using the FILE ... clause) or is passed as data stream in a character variable. The image may be cleared by usual CLS, CLEAR SCREEN, @..CLEAR TO.. command and corresponding functions.

Supported image types are bmp, gif, jpeg/jpg, png, ppm, xbm and xpm. See their description in the IMGTYPE clause. All other file types can easily be converted to one of these supported formats by any graphic image program like Gimp, Photoshop, Paint, IfranView etc. (use export to... or save as...).

When you want to store images in databases, use MEMO field and MemoCode() to store and MemoDecode() to access the image. If the image size is too large for MEMO fields (32/64kb), you may compress/uncompress it by CharPack() / CharUnpack() from the FS2 Toolbox. Alternatively, you may use directly FlagShip's variable database fields VB or VBZ to store images up to 2GB, see also DbCreate() for details.

The image is displayed on screen only when SET DEVICE TO SCREEN is active, and printed when when SET GUIPRINT ON or PrintGui(.T.) is active; the printout is independent on SET DEVICE setting.

The @..SAY IMAGE command is processed also for GUI/GDI printout (when SET GUIPRINT ON is active) and accepts all except BORDER clauses.

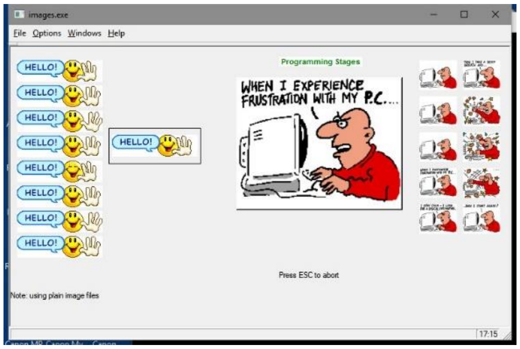
Alternative syntax for @..SAY IMAGE is @..DRAW IMAGE

Example 1:

```
#include "box.fh"
@ 10,40 SAY IMAGE file "myimg.gif"
cImgVar := "..\images\otherimage.bmp"
@ 15,50,18 SAY IMAGE from file (cImgVar) border BOX_PLAIN
@ 350,500,480,600 SAY IMAGE file "myimg.jpg" PIXEL NOSCALE

local cImgData := memoread("../images/myimg.png")
@ 10,40,,60 SAY IMAGE cImgData SCALE border BOX_SUNKEN
```

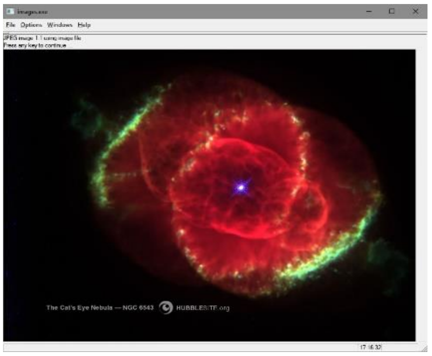
Example 2:

see also <FlagShip_dir>/examples/images.prg and printergui.prg


images.exe
File Options Windows Help
Programming Stages
WHEN I EXPERIENCE FRUSTRATION WITH MY P.C....
HELLO!
17:15

Press ESC to abort

Note: using plain image files


images.exe
File Options Windows Help
The Cat's Eye Nebula — NGC 6543 HUBBLESITE.org

JPEG image 1: 1 using image file
Press any key to continue

Classification:

screen oriented output in GUI mode as well as GUI printout

Compatibility:

New in FS5, printer support is new in FS7

Translation:

DisplImageData() or DisplImageFile()

Related:

@..DRAW CIRCLE, @..DRAW ELLIPSE, @..DRAW ARC, @..DRAW LINES, @..DRAW PIE, @..DRAW POLYON, @...BOX, @...TO.., SET GUIPRINT, MemoCode(), MemoDecode(), SET GUIPRINT

@...[SAY..] GET

Syntax 1:

```
@ <expN1>,<expN2>
     GET <variable>
        [CAPTION <capt>]
        [CLEAR|DESTROY]
        [COLOR <expC9>]
        [GUICOLOR <expC9>]
        [DEFAULT <defa>]
        [ENABLE|DISABLE]
        [ERRORVALID <errBlk>]
        [MESSAGE <text>]
        [NOALIGN]
        [PICTURE <expC4>]
        [RANGE <expN6>,<expN7>]
        [SEND <exp11>]
        [TOOLTIP <ttip>]
        [USERMSG <exp10>]
        [USING <obj>]
        [VALID <expL8>]
        [WHEN <expL5>]
        [HEIGHT <expN11>]
        [WIDTH <expN12>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN13>)]
        [FONT <expO14>]
        [MULTIBYTE]
```

Syntax 2:

```
@ <expN1>,<expN2>
     SAY <exp3>
        [PICTURE <expC4>]
        [COLOR <expC9>]
        [GUICOLOR <expC9>]
     GET <variable>
        [CAPTION <capt>]
        [CLEAR|DESTROY]
        [COLOR <expC9>]
        [GUICOLOR <expC9>]
        [DEFAULT <defa>]
        [ENABLE|DISABLE]
        [ERRORVALID <errBlk>]
        [MESSAGE <text>]
        [NOALIGN]
        [PICTURE <expC4>]
```

```
[RANGE <expN6>,<expN7>]
[SEND <exp11>]
[TOOLTIP <ttip>]
[USERMSG <exp10>]
[USING <obj>]
[VALID <expL8>]
[WHEN <expL5>]
[HEIGHT <expN11>]
[WIDTH <expN12>]
[PIXEL|NOPIXEL]
[UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN13>)]
[FONT <expO14>]
[MULTIBYTE]
```

Purpose:

Prepares one entry field for the full-screen data input using the READ command. Displays the current data at the specified row and column positions. Creates a new object within the GETLIST array. Optionally, displays an additional text using the SAY clause.

Syntax 2 represents a combination of syntax 1 and the command @...SAY. In the following description syntax 1 will normally be referred to.

Arguments:

<**expN1**>, <**expN2**> are numeric expressions, specifying the row and column coordinates of the input field (syntax 1) or of the text being displayed (syntax 2). In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

The coordinates are in the range 0..24 and 0..79 for a 25x80 screen or MAXROW() and MAXCOL() respectively. With syntax 2, the input field starts at the end of the <**exp3**> text and an additional space character. Output which extends beyond the visible end of the display is clipped and does not appear.

GET <**variable**> is a database field or a memory variable, the contents of which is displayed and added to the list of pending GETs to be enabled for input and editing by the READ command. The <variable> can be of the type character, numeric, date or logical. If the storage class is ambiguous, FIELD is assumed. Fields from other working areas can be used as <variable> by referring to them via their alias.

Options: *(in alphabetical order)*

CAPTION <capt> is the displayed text instead of @..SAY

CLEAR|DESTROY forces to clear the GET display at the end of variable visibility scope. See also READ CLEAR

COLOR <**expC9**> is a standard color string with 5 to 8 color pairs for displaying the GET field <variable>. An inactive GET field is displayed by using color pair 5 (unselected), an active GET field with input focus (get:HasFocus) is displayed by color pair 2 (enhanced). Disabled GETs by using the DISABLE clause is displayed by color pair 7 if such is available, otherwise by pair 5. In GUI mode the color pair 8, if available, is used for GETs in unselected window. See SET COLOR for more details.

With syntax 2, different colors for the SAY and GET command parts can be used. If omitted, the current color setting is used. In GUI mode, first the GUICOLOR is checked if set. If not so, either this COLOR <expC9> or the current color is used, but only when SET GUICOLOR is ON. Specifying COLOR and GUICOLOR allows you to handle different colors for Terminal and GUI i/o mode.

GUICOLOR <**expC9**> specifies the color for the display of the input field <variable> considered in GUI mode only. If set, it is used regardless the current SET GUICOLOR setting. If omitted and SET GUICOLOR is ON, either the COLOR <expC9> is used if given, or the current SetColor() is used. The GUICOLOR clause applies for GUI mode only, and is ignored otherwise. <expC9> is a standard color string with 5 to 8 pairs, where color pair 2 is used to display the active GET, color pair 5 is used for the unselected GETs, color pair 7 for disabled GETs, and color pair 8 for GETs in an unselected window. See 'SET COLOR TO' for more details.

DEFAULT <defa> set the GET <variable> to <defa> value if <variable> is NIL, empty() or of different type than <defa>

ENABLE|DISABLE enable (default) or disable the item from READ processing. It is equivalent to atail(Getlist):GetEnabled := .T./.F. The object property can also be set at any time later; to re-display the GET use getObject:Display(.T.). For enabling/ disabling the GET at run-time according to current condition, you may preferably use the WHEN clause.

ERRORVALID <errBlk> use the code block <errBlk> to display post-validate error/ failure

MESSAGE <text> display message <text> in status bar or in the SET MESSAGE line when the GET field receives focus

NOALIGN don't align this field even if SET GUIALIGN is ON

PICTURE <**expC4**> gives formatting rules for the <variable> input. With syntax 2, different pictures for the SAY and GET command parts may be used. When no PICTURE is given, the format is determined by examining the value in <exp3> and/or <variable>. Note that PICTURE cannot be considered with Unicode, except the "@S" in FUNCTION part.

RANGE <**expN6**>,<**expN7**> (post-validation) are the lower and upper limits of acceptable numeric input. The lower limit <expN6> must always precede the upper limit <expN7>. If the input or the edited value is not inside the interval, a message to this effect will be displayed and the control will be returned to the GET. This check is performed only when a new value is entered or the available data edited (same as in Clipper), except you set

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

which is then tested always at exiting GET, same as VALID clause.

SAY <**exp3**> is an expression displayed prior to the entry field and evaluated by the SAY clause (see more @...SAY).

SEND <**exp11**> is a full object message to be sent to the current object.

TOOLTIP <ttip> (GUI only) short pop-up message/info displayed when mouse cursor is over the GET field, even w/o focus

USERMSG <**exp10**> is a message (expression) of any type, which will be sent (assigned) to the current `get:CARGO` instance variable.

USING <obj> use already instantiated GET object <obj>, avoid a new creation/ instantiation

VALID <**expL8**> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding GET. Should the expression return a .F. value, the cursor will remain on the current field. Note: return value other than logical assumes success and developer warning occurs when FS_SET("devel",.T.) was set. This feature is often used for lookups using post-processing functions.

WHEN <**expL5**> (pre-validation) specifies an expression (or UDF or codeblock returning a logical value) that must be satisfied in order to enter the GET field during a READ. Note: return value other than logical assumes success, additionally a developer's warning occurs when FS_SET("devel",.T.) was set.

HEIGHT <**expN11**> specifies the widget height in rows or pixels according to the current SET PIXEL on/OFF setting. This clause is considered in GUI mode only and is equivalent to the oGet:Height access.

WIDTH <**expN12**> specifies the widget width in columns or pixels according to the current SET PIXEL on/OFF setting. This clause is considered in GUI mode only and is equivalent to the oGet:Width access. It behaves similarly to PICTURE "@S..." but is applicable also for non-character fields.

PIXEL : the <expN1>, <expN2> are values in pixel

NOPIXEL : the <expN1>, <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN13> specifies unit for <expN1> .. <expN2> coordinates. The <expN13> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_CO-ORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

FONT <expO14> (GUI only, ignored otherwise) The <expO14> is already instantiated font object, which should be used for the GET field. The field width and height is calculated according to this font. If not given, the default font or SET FONT or oApplic:Font is used.

MULTIBYTE enables MULTIBYTE entry for this GET, overwrites temporary SET MULTIBYTE ON/OFF. Set if this GET in READ should accept Unicode but other GETs in this READ should behave according to SET MULTIBYTE which is usually OFF. See also Unicode section below.

Description:

The @...GET command performs several actions. It displays the default contents of a variable or a database field at row and column, and formats the output according to the optional picture format. Appending it to the GETLIST array will create a new GET object. The <variable> is associated with that object, as well as the other, optional modifiers. A subsequent READ command enables full-screen editing, using the data stored by the GET command.

In GUI mode, the GET field appears in widget (control), which is a kind of small sub-window. You therefore cannot overwrite the GET field by subsequent SAY or by other widget. The GET will be cleared automatically by CLEAR GETS, end of READ or by any kind of CLEAR.

When SET DELIMITERS are set ON, the @...GET command will display the default or user defined delimiters around the GET edit field and will shift the GET object display by one column right. This is fully supported in Terminal i/o mode; in GUI mode the delimiters are not displayed, but the GET column is corrected.

If the GETLIST variable has not been declared PRIVATE or LOCAL in the current procedure, the predefined PUBLIC GETLIST variable is used. Declaring a PRIVATE, LOCAL (or STATIC) array allows you to nest GET/READs to any depth in subsequent UDFs.

The READ command performs a full-screen edit of the GETs in the GETLIST array. As the user moves the cursor into each GET field, evaluating the user defined or default code block saved in the GET object retrieves the value of the associated <variable>. The value is converted to textual form and placed in a buffer within the GET field (object). This buffer is displayed on the screen, and the user is allowed to edit the text from the keyboard. When the user moves the cursor out of the GET, the updated buffer is converted back to the appropriate data type and assigned to <variable>.

For more information, refer to CMD.READ.

Unicode:

In GUI mode, FlagShip supports also Unicode (UTF-8 and UTF-16). If the <expO14> font is set to Unicode by `oFont:CharSet(FONT_UNICODE)` or globally by `oApplic:Font:CharSet(FONT_UNICODE)` or by `SET GUICHARSET FONT_UNICODE`, Unicode glyphs displays eg. for Asian languages. To accept multi-byte strings for the glyph in READ, SET MULTIBYTE ON or the MULTIBYTE clause must be set as well. Predefined strings need to be stored in UTF-8 encoding, or transformed from UTF-16 by `Utf16_Utf8()`. When the SET MULTIBYTE ON is active or the MULTIBYTE clause is set, all automatic conversions for SET GUITRANSL etc. are disabled. Note that PICTURE cannot be considered with Unicode except some FUNCTION parts like "@S".

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 set the GET field correspondingly (e.g. to 30 or more characters to accept 10 Japanese or Chinese glyphs). In Linux you may need to set Unicode font, e.g. SET FONT "mincho" as well. See further info in section LNG.5.4.5 and example in <FlagShip_dir>/examples/unicode.prg

Color:

If the COLOR clause is not specified, the GET field is displayed in the current "enhanced" color pair (see SET COLOR and SETCOLOR()). If the "unselected" color is specified in SETCOLOR() as well, only the currently active GET is displayed in enhanced color while executing READ. All the other GET fields are then displayed in the "unselected" color.

Each GET field can have a different color specification defined by using the COLOR clause or the command SET COLOR; the current color setting is stored in the GET object during execution of the @...GET command.

When the COLOR clause of @..GET is specified, this color attributes are passed to the instance variable `get:COLORSPEC` (see sect. OBJ). Since the special COLORSPEC notation <inactiveField>,<activeField>, the COLOR clause of @...GET..COLOR command differ from the standard SETCOLOR() notation. For your convenience, you may use the SETCOL2GET() function to transform the current (or any user defined) color setting into the proper notation, see example below.

In GUI mode, colors are disabled by default. You may enable it by SET GUICOLOR ON and/or use the GUICOLOR clause, see details above.

Picture:

<expC4>, the PICTURE clause, is a string and consists of two optional parts, the FUNCTION and the TEMPLATE, separated by at least one blank. Functions apply to the entire <variable> while templates mask corresponding characters of <variable>. Function and template symbols are not case-sensitive.

The FUNCTION part, when given, must precede the template and start with the "@" sign. All the symbols thereafter up to the first blank are interpreted as functions. The rest is taken as TEMPLATE. In the absence of the "@", the whole string is considered a template. Picture FUNCTIONS are applied to the entire <variable> field. Multiple function definitions are allowed.

A TEMPLATE part specifies formatting or validation rules on a character by character basis. The template string consists of a series of characters, some of which have special meanings (see the following table). Each position in the template string corresponds to a position in the displayed GET value. Characters in the template string that do not have assigned meanings are copied verbatim into the displayed GET value as un-editable characters. If you use the @R picture FUNCTION, these characters are inserted between characters of the display value, and are automatically removed when the display value is reassigned to <variable>; otherwise, they overwrite the corresponding characters of the display value and also affect the value assigned to <variable>. You may specify a template string alone or with a function string. If you use both, the function string must precede the template string, and the two must be separated by a single space.

If you set FS_SET("devel", .T.), PICTURE problems and fixes are displayed as developer's warning.

Picture FUNCTION symbols in "@**...**". For the @..GET command or the GET part, the S/G or GET mode apply in active READ field; for the SAY part or for inactive READ the SAY or S/G mode apply:

Func	Type	Mode	Definition
A	C	GET	Only alphabetic characters are allowed
B	N	SAY	Numbers are displayed left-justified
C	N	SAY	'CR' for credit is displayed after positive numbers
D	D	S/G	Dates are displayed in the SET DATE format
E	D	S/G	Dates are displayed in European format (day and month are exchanged)
E	N	S/G	Numerics are displayed in European format (comma & period are exchanged)
K	all	GET	GET is cleared if the first key is not a cursor or Insert key
P	C	S/G	Password, displayed as '*'
R	C	S/G	Non-template characters from the TEMPLATE part of picture are inserted during in/output but removed from the value
Sn	C	S/G	Horizontal scrolling within a GET window of <n> columns is allowed, SAY displays only the first <n> characters
X	N	SAY	'DB' for debit is displayed after negative numbers
Z	ND	S/G	Leading zeros are displayed as blanks
(	N	SAY	Negative numbers are enclosed in parentheses with leading spaces
)	N	SAY	Negative numbers are enclosed in parentheses without leading spaces
!	C	S/G	Alphabetic characters are converted to uppercase
F	N	SAY	fill leading spaces with stars "*"
T	all	SAY	remove leading and trailing spaces
_	C	S/G	(= underscore) replace "_" in template picture to protected space.
~	C	S/G	(= tilde) replace "~" in template picture part to protected space.

Note that PICTURE cannot be considered with Unicode, except the "@S" in FUNCTION part.

Picture TEMPLATE symbols. The GET or S/G mode apply in active READ field and SAY or S/G apply otherwise:

TEM	Type	Mode	Definition
X	C	S/G	Any character is accepted
A	C	GET	Only alphabetic characters w/o space are accepted
B	C	GET	Only alphabetic characters and space are accepted
N	C	GET	Only alphanumeric characters w/o space are accept.
9	CND	S/G	Digits for any data type including the sign for numerics are accepted
#	CND	S/G	Digits, signs and spaces for any data type are accepted
L	L	S/G	The logicals "T" or "F" are accepted
Y	CL	S/G	Only "Y" or "N" are allowed
!	all	S/G	An alphabetic character is converted to uppercase
$	N	SAY	The dollar sign $ is displayed in place of a leading space in a numeric
*	N	SAY	The asterisk is displayed in place of a leading space in a numeric
.	N	S/G	The period defines the decimal point position, regardless of the given @E conversion
,	N	S/G	The comma defines the 'thousands' comma position, regardless of the given @E conversion
^	C	S/G	(= circumflex) Un-editable (protected) output char
_	C	S/G	(= underscore) Set un-editable space in output when also "@_" is set, as alternative to " " in picture
~	C	S/G	(= tilde) Alternative to "_", set un-editable space in output when also "@~" is set
any	other		Template symbol is copied to output and treated as un-editable character

Validation:

During GET execution, no validation takes place. The READ command checks the pre-valid condition (WHEN clause) to decide whether to enter the corresponding field. If the condition returns FALSE, the field is skipped. Post-validation, (RANGE and VALID clause) will be done any time the user wants to leave the current field. If the VALID condition returns TRUE, the user is allowed to leave; otherwise, the cursor remains in the current GET field.

If RANGE is specified, the data entered has to be within the defined limits to enable leaving the field. If the test fails, a message appears on the screen. In FlagShip, the message is user- definable using FS_SET("load"/"set") and can be enabled or disabled using the SET SCOREBOARD command.

With SET ESCAPE ON, no post-validation is performed.

Executing an UDF:

For user friendly programs, it is common to create a context sensitive help system, using F1 (or other key) for the current data being edited: the command SET KEY TO may redefine any required key to execute a user procedure. On application start, FlagShip pre-defines SET KEY 28 TO HELP, so that by adding the PROCEDURE HELP to your application, you may access it automatically when pressing [F1] in any READ field. You may also use other keys or redefine F1 using SET KEY <code> TO <udp> prior the READ command and, if necessary, disable it afterwards using SET KEY <code> TO. For information, refer to LNG.5.2.2 and (CMD) SET KEY.

Within the UDP or UDF, the current <variable> being edited can be determined by READVAR() or with the current GET object using GETACTIVE().

The UDP or UDF may change the contents of any GET field being edited. To abort the READ, CLEAR, CLEAR GETS or KEYBOARD CHR(27) commands can be used.

Cut & Paste:

Depending on the currently used i/o mode (GUI, Terminal), you may insert/overwrite characters in the GET field by 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. Insert the clipboard text into GET field by **Alt-V** key, copy by **Alt-C** key (both user modifiable). See CMD.READ for further details and settings.

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, e.g. SET TYPEAHEAD TO 500.

Tuning:

In GUI mode with proportional font, `@ 5,1 SAY "XXXX" GET var1` ; `@ 6,1 SAY "iiii" GET var2` would set fields var1 and var2 at different column position, since the GET column in this compound statement is calculated from the last SAY position + one space. Because the width of "XXXX" differs from "iiii" with proportional font, also these GET columns differs. FlagShip's READ therefore re-calculates and adjust all GETs to fit among each other, if applicable. You may disable this feature by NOALIGN clause or globally by SET GUIALIGN OFF or e.g. by SET FONT "courier",10 - see also example in SET GUIALIGN

In GUI mode, drawing graphic lines sometimes requires refresh. If your display flickers, you may disable the refresh by assigning

```
  _aGlobSetting[GSET_G_N_REFRESHDRAW] := -1  // default = 300 ms
```

In GUI mode, the GETs are displayed using "widgets" (or "controls" MS terminology). When the READ finishes, you may re-display them via SAY by assigning

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

latest before executing the READ statement.

In GUI mode, the @..GET/READ field size width is calculated to fit the requested amount of characters. In some cases and fonts, you may need to add a small displacement to avoid horizontal scrolling, by assigning some more pixel to

```
   _aGlobSetting[GSET_G_N_GET_WIDTH ] := 8   // default = 8 pixel
```

You also may modify general adjustment to field width/heght

```
_aGlobSetting[GSET_G_L_GET_ADJ   ] := .T. // adjust row/col?
_aGlobSetting[GSET_G_N_GET_ROW   ] := 0   // add pix Get row
_aGlobSetting[GSET_G_N_GET_COL   ] := -2  // add pix Get column
_aGlobSetting[GSET_G_N_GET_HEIGHT] := 0   // add pix Get height
```

Additional tuning is described in the READ command.

Example 1:

The cursor is positioned under the first element (during the READ command) of the string and the program waits for input. Only numerical input is allowed for the first three positions. Three non-template symbols then follow, which allows no input. The rest of the string allows only alphabetic input. The blank is a non- template character and input is not possible in its place in the PICTURE string.

```
value = "123---paris  london"
@ 11,11 GET value PICTURE "@! 999---AAAAA  XXXXXX"
READ
* Result:                     123---PARIS  LONDON
```

Example 2:

When the first key pressed is not a cursor key, the input field will be cleared. The same effect appears on numeric data entry, where the "@K" is by default.

```
value = "Text  "
@ 11,11 GET value PICTURE  "@K! XXXXXX"
*       Result :                TEXT
```

Example 3:

The entry of long string within a short input window is supported using the "@S" format function:

```
@ 6,5 GET value PICTURE "@S10 !!!!XXXXXXXXXXXXX"
*     Result :                THE long string
*     pressing the -> key    THE long string
*     etc.                  THE long string
*                                |       |
                                 |       └-- currently invisible
                                 └---------- the input field
```

Example 4:

All the following statements set the active GET field to yellow on red, the inactive GET field is displayed in white on cyan:

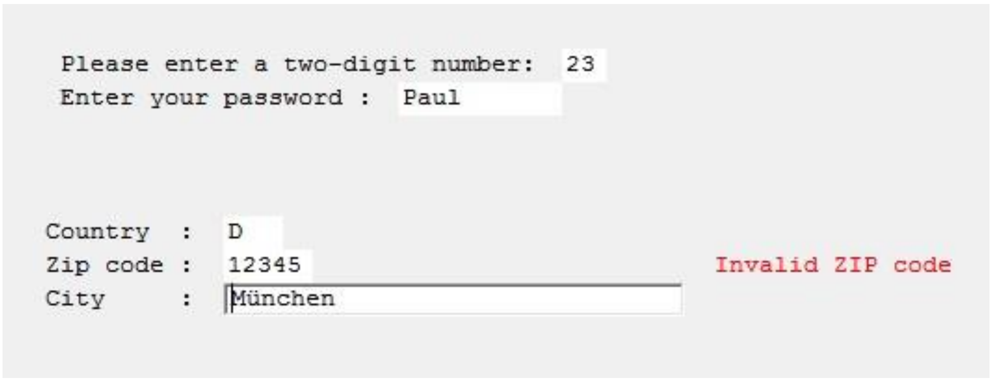
```
SET COLOR TO "W+/B,GR+/R,,,W+/BG"
xxx = "W+/BG,GR+/R"
@...GET varname                 // uses automatically SETCOL2GET()
@...GET varname COLOR SETCOL2GET() // the same color as above
@...GET varname COLOR xxx       // passes xxx to get:COLORSPEC
GETNEW (,,,,,SETCOL2GET())      // sets get:COLORSPEC to curr.color
get:COLORSPEC := SETCOL2GET()   // sets get:COLORSPEC to curr.color
get:COLORSPEC := "W+/BG,GR+/R"  // equivalent to the above
```

Example 5:

Example of several validity checks:

```
LOCAL   value := 0, passw := space(10)
PRIVATE numzip := 0, country := "   ", city := space(30)
@ 10,11 SAY "Please enter a two-digit number: " ;
        GET value PICTURE "99" RANGE 10, 99
@ 11,11 SAY "Enter your password : " ;
        GET passw ;
        VALID ","+trim(passw)+"," $ ",Peter,Paul,"
@ 15,10 SAY "Country  : " GET country PICTURE "@!"
@ 16,10 SAY "Zip code : " GET numzip ;
        PICTURE "99999" ;
        WHEN TRIM(country) == "D" ;
        VALID check_zip()
@ 17,10 SAY "City     : " GET city
READ
if lastkey() = 27                // Exit per ESC ?
   return                        // yes, back to menu
endif

FUNCTION check_zip               /* check zip codes */
LOCAL act_select, ok
act_select = SELECT()            // save act.working area
SELECT 25                        // select ZIP database
SEEK numzip                      // seek current entry
ok = FOUND()                     // found ?
IF ok                            // yes,
   city := FIELD->zip_city       // predefine city name
ELSE
   @ 16,55 say "Invalid ZIP code" GUICOLOR "R+"
ENDIF
SELECT (act_select)              // restore act.working area
RETURN (ok)                      // validation .T. or .F.
```

Output:



```
  Please enter a two-digit number:  23
  Enter your password :  Paul



 Country  :  D
 Zip code :  12345                           Invalid ZIP code
 City     :  München
```

Example 6:

Example of an array validation. Do not use the FOR index ii to VALIDate, since in READ it will already have the value 4.

```
LOCAL ii, arr := {1,2,3}, check := {{1,10}, {2,22}, {3,33}}
LOCAL col := {SETCOLOR(), "W+/B,GR+/G,,,W+/R", "W/B,N/W"}
FOR ii := 1 TO LEN(arr)
   @ ii,1 GET arr[ii] PICTURE "999" ;
          COLOR (col[ii]) ;
          SEND CARGO := ii ;             // or: USERMSG ii;
          VALID arrcheck (arr, check)
NEXT
READ

FUNCTION arrcheck (inarr, checkarr)
LOCAL element := GETACTIVE():CARGO, value
value := inarr [element]
IF value >= checkarr[element, 1] .AND. ;
   value <= checkarr[element, 2]
   RETURN .T.
END
inarr[element] += 1                     // assign new value
RETURN .F.
```

Example 7:

The same array check routine, generalized for any GET type and for multi-dimensional GET array entry and access. The SEND clause in a @..GET statement is used for checking purposes only and may be omitted, using e.g. get:SUBSCRIPT[1].

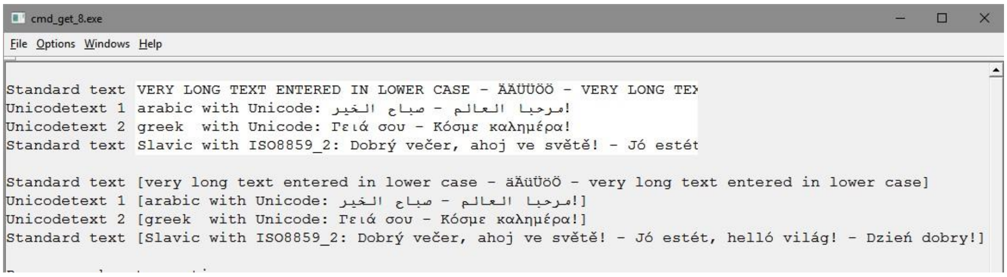
```
FUNCTION arrcheck (inarr, checkarr)
LOCAL get := GETACTIVE(), elem, value, chkidx
PRIVATE arrname := get:NAME
PRIVATE &arrname := inarr            // get orig.arr ptr
elem  := READVAR()                   // e.g. "ARR[3]"
value := &(elem)                     // current value
chkidx:= get:CARGO                   // or get:SUBSCRIPT[1]

IF value < checkarr[chkidx, 1] .OR. ;
   value > checkarr[chkidx, 2]
   RETURN .F.
ENDIF
&(elem) += 1                         // assign new value
RETURN .T.
```

Example 8:

Enables Unicode for two @..GET fields UNITXT1 and UNITXT2, all other are processed by default or by special charset. See also <FlagShip_dir>/examples/arabic.prg, greek.prg, slavic.prg, unicode.prg, western.prg

```
 #include "font.fh"
 SET FONT "Courier", 12                // default font
 m->oApplic:Resize(25,110,,.T.)       // resize screen

 LOCAL cText1, cText2, unitxt1, unitxt2
 cText1 := cText2 := unitxt1 := unitxt2 := space(100)

 LOCAL oUniFont := FontNew()           // For Unicode input/output
 oUniFont:CharSet(FONT_UNICODE)
 oUniFont:FontName("Courier")
 oUniFont:Size := 12

 LOCAL oFontSlav := FontNew()         // For Slavic input/output
 oFontSlav:FontName("Courier")
 oFontSlav:Size := 12
 oFontSlav:CharSet(FONT_ISO8859_2)

#ifdef FS_LINUX
    oUniFont:FontName := "mincho"    // Linux: Unicode font required
 #endif

 @ 1,0 say "Standard text" GET cText1  PICT "@S60!"
 @ 2,0 say "Unicodetext 1" GET unitxt1 ;
           PICT "@S60" FONT (oUniFont)
 @ 3,0 say "Unicodetext 2" GET unitxt2 ;
           PICT "@S60" FONT (oUniFont)
 @ 4,0 say "Standard text" GET cText2  PICT "@S60" FONT (oFontSlav)
 READ
 @ 6,0 say "Standard text [" + trim(cText1) + "]"
 @ 7,0 say "Unicodetext 1 [" + trim(unitxt1) + "]" FONT (oUniFont)
 @ 8,0 say "Unicodetext 2 [" + trim(unitxt2) + "]" FONT (oUniFont)
 @ 9,0 say "Standard text [" + trim(cText2) + "]"  FONT (oFontSlav)
```

Output:



```
Standard text VERY LONG TEXT ENTERED IN LOWER CASE - ÄÄÜÜÖÖ - VERY LONG TEX
Unicodetext 1 arabic with Unicode: مرحبا العالم - صباح الخير!
Unicodetext 2 greek  with Unicode: Γειά σου - Κόσμε καλημέρα!
Standard text Slavic with ISO8859_2: Dobrý večer, ahoj ve světě! - Jó estét

Standard text [very long text entered in lower case - äÄüÜöÖ - very long text entered in lower case]
Unicodetext 1 [arabic with Unicode: مرحبا العالم - صباح الخير!]
Unicodetext 2 [greek  with Unicode: Γειά σου - Κόσμε καλημέρα!]
Standard text [Slavic with ISO8859_2: Dobrý večer, ahoj ve světě! - Jó estét, helló világ! - Dzień dobry!]
```

Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND(), used for subsequent screen oriented input (via READ)

Compatibility:

New in FS4 are the clauses SEND and USERMSG, as well as the usage of the GET object.

In FlagShip, both the RANGE and the VALID clauses may be specified. RANGE is checked first.

Clipper ignores wrong PICTURE characters, FlagShip reports them in development mode when FS_SET("devel",.T.) is set. When there are no separating spaces between the function and the template part, FlagShip tries to determine the template from the context, where possible.

FlagShip does not truncate the most significant digits of numeric output within short pictures; it tries, if possible, to output the whole number by removing inserted chars or by shortening the PICTURE deci part containing zeros. To disable this feature, and to display stars instead, assign _aGlobSetting[GSET_L_ADAPT_PICT] := .F.

Similarly, to strings: if the template PICTURE characters does not match the string length, the template is automatically extended by "X" instead of truncating the input variable (as Clipper illegally do). This means in generally: FlagShip does not modify the input variable length, but only the PICTURE template, if required.

FlagShip's GETs are performed via the GET class (see section OBJ), and are therefore fully user modifiable. The standard READ command is available in source code in the getsys.prg file.

Unicode support is available in VFS7 and later in GUI mode only

See also terminal & GUI information in @..SAY and LNG.5.

Unicode is supported in VFS7 and later.

The clauses DEFAULT, GUICOLOR, USING, ERRORVALID, CAPTION, MESSAGE, TOOLTIP, CLEAR, DESTROY, ENABLE, DISABLE, PIXEL, NOPIXEL, NOALIGN, WIDTH are new in FS5

Class:

GET, prototyped in <FlagShip_dir>/include/getclass.fh

Translation:

```
__SUBSCARR (.T.) ; __scratch := variable ; __SUBSCARR (.F.)
AADD (GetList, _FSGET_ (expN1, expN2, "variable", ;
        Standard_GET_CodeBlock, [expC4], ;
        [Range_Valid], [{expL4}], [expC9], __subscarr) )
[ ATAIL(GetList):Cargo := exp10 ]
[ ATAIL(GetList):exp11 ]

Standard_GET_CodeBlock := {|input| ;
        IF(input == NIL, variable, variable := input) }
Range_Valid := {|input| .T. ;
        [ .AND. RANGE_CHECK (input, expN6, expN7) ] ;
        [ .AND. expL8] }
```

READ Handler Source:

<FlagShip_dir>/system/getsys.prg

Related:

?/??, @...SAY, @...TO, @...CLEAR, CLEAR, CLEAR GETS, KEYBOARD, READ, SET BELL, SET CONFIRM, SET DELIMITERS, SET DEVICE, SET KEY, SET FORMAT, SET INTENSITY, Col(), Row(), FS_Set(), ReadGetPos(), ReadSelect(), ReadExit(), ReadInsert(), ReadKey(), ReadKill(), ReadModal(), ReadSave(), ReadUpdated(), ReadVar(), OBJ.Get, LNG.5.4

@...[SAY..] GET CHECKBOX

Syntax 1:
```
     @ <expN1>,<expN2>
           GET <varL> CHECKBOX
              [CAPTION <cCapt>]
              [COLOR <cBoxColor>]
              [DEFAULT <lDef>]
              [ENABLE|DISABLE]
              [ERRORVALID <bError>]
              [FOCUS <fblock>]
              [HEIGHT<nHeight>]
              [MESSAGE <cText>]
              [PIXEL|NOPIXEL]
              [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
              [SEND|GUISEND <snd>]
              [STATE <sBlock>]
              [STYLE <cStyle>]
              [TOOLTIP <cTip>]
              [USERMSG <cargo>]
              [USING <obj>]
              [VALID <lValid>]
              [WHEN <lWhen>]
```

Syntax 2:
```
     @ <expN1>,<expN2>
           SAY <cSaytext>
              [PICTURE <cSayPict>]
              [COLOR <cSayColor>]
           GET <varL> CHECKBOX
              [CAPTION <cCapt>]
              [COLOR <cBoxColor>]
              [DEFAULT <lDef>]
              [ENABLE|DISABLE]
              [ERRORVALID <bError>]
              [FOCUS <fblock>]
              [HEIGHT<nHeight>]
              [MESSAGE <cText>]
              [PIXEL|NOPIXEL]
              [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
              [SEND|GUISEND <snd>]
              [STATE <sBlock>]
              [STYLE <cStyle>]
              [TOOLTIP <cTip>]
              [USERMSG <cargo>]
              [USING <obj>]
```

```
            [VALID <lValid>]
            [WHEN <lWhen>]
```

Purpose:

Creates CheckBox widget and let it process via common READ.

Arguments:

<expN1>, <expN2> are numeric expressions, specifying the row and column coordinates. With Syntax 1: coordinates of the widget. With Syntax 2: coordinate of the capture text, the widget column coordinate is the over-next column behind the text <cSaytext> end, same as in @..SAY..GET. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

SAY <cSaytext> is a text caption identifying the CheckBox on the screen.

GET <varL> is a database field or a memory variable of logical type storing the "checked" status of the CheckBox.

CHECKBOX clause is mandatory here.

Options:

CAPTION <cCapt> is a text explaining the CheckBox

COLOR <cSayColor> is an optional color specification for the @..SAY text in terminal i/o mode, or in GUI with SET GUICOLOR ON.

COLOR <cBoxColor> defines the color settings for the check box, applicable for Terminal i/o only. The string may contain 5 color pairs:

Pair#	Used for	Default
1	Check box without input focus	Unselected
2	Check box with input focus	Enhanced
3	The check box's caption	Standard
4	The check box caption's accelerator key	Background
5	Border	Border

For not specified pair, the default from current SetColor() is used

DEFAULT <lDef> set the GET <varL> to <lDef> value if <varL> is NIL, empty() or of different type than <lDefa> which must be logical.

ENABLE|DISABLE enable (default) or disable the item from READ processing

ERRORVALID <bError> specifies to use the <bError> code block to display post-validate error/failure

FOCUS <fblock> specifies a code block that is evaluated each time the CheckBox receives focus. The code block receives two parameters, the current CheckBox object, and the oBox:HasFocus status.

HEIGHT <nHeight> is an optional height of the check box, the default is 1 row.

MESSAGE <cText> displays message <text> in status bar or in the SET MESSAGE line when the CheckBox receives focus

PICTURE <cSayPict> is the optional picture of @..SAY text

PIXEL : the <expN1> and <expN2> are values in pixel

NOPIXEL : the <expN1> and <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN2> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

SEND <instance> } allows you to assign any valid class instance
GUISEND <instance> } or method. Supported for Clipper compatibility.

STATE <sBlock> specifies a code block that is evaluated each time the CheckBox state changes, i.e. is checked or unchecked. The code block receives two parameters, the current CheckBox object, and the oBox:Buffer status.

STYLE <cStyle> specifies a character string that indicates the CheckBox delimiter characters for Terminal i/o. The default style is pre-defined in the global array element

```
   _aGlobSetting[GSET_T_C_CHBOX_STYLE] := "[X ]?"
```

which is used when the STYLE clause is omitted.

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the CheckBox widget, even w/o focus

USERMSG <cargo> assigns the <cargo> value to the CheckBox:Cargo instance

USING <obj> use already instantiated CheckBox object <obj>, avoid a new creation/instantiation

VALID <lValid> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding GET. Should the expression return a .F. value, the cursor will remain on the current field. This feature is often used for lookups using post-processing functions.

WHEN <lWhen> (pre-validation) specifies an expression that must be satisfied in order to enter the CheckBox during a READ

Description:

The @...GET...CHECKBOX uses the CheckBox class. You may use it additional properties by e.g. Atail(GetList):<instance> := <value>

Tuning:

The action on a key or mouse button press is defined in the user modifiable handler <FlagShip_dir>/system/checkboxhand.prg. Mouse is supported in GUI mode only.

The default behavior on mouse button click is: Left mouse click selects/clears the CheckBox and leaves the CheckBox to next GET (if any), same as the `+,-,space,x,y,t,n,f` key press. Mid and right mouse button toggles the button but stays in the CheckBox until the corresponding key leaves it. Space or 'x' key toggles the status, the '+','y','t' key sets the CheckBox on, and '-','n','f' key press sets it off. Return only skips to next field.

By assigning _aGlobSetting[GSET_G_L_CHBOX_SINGLE] := .F. you may avoid leaving CheckBox by left mouse button. The supported mouse buttons are specified in _aGlobSetting[GSET_G_A_CHBOX_MOUSE] array, see *<FlagShip_dir>/system/initio.prg*

Mouse click on CheckBox caption (if any) may or may not toggle it, same as click within the box. You may control it by setting

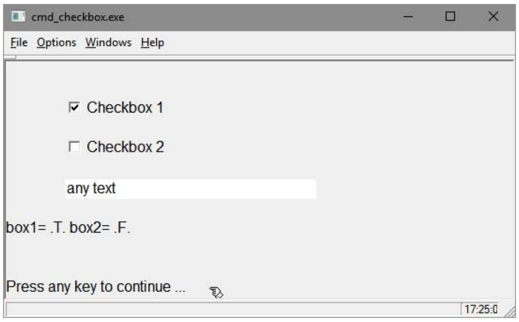
```
   _aGlobSetting[GSET_G_L_CHBOX_SELCAPT] := .T.   // default
```

You may control the pixel adjustment of the checkbox by elements in the global variable _aGlobSetting[GSET_G_*_CHBOX*], see details in <FlagShip_dir>/system/initio.prg

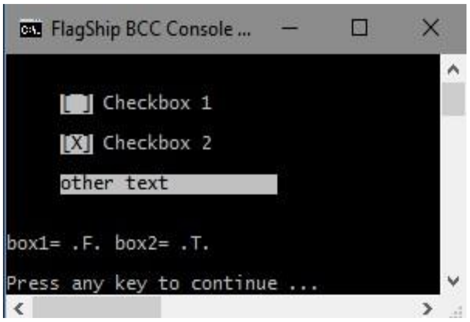
Example:

```
local lBox1 := .F., lBox2 := .T., cText := space(20)
@ 2,5 get lBox1 CHECKBOX CAPTION "Checkbox 1"
@ 4,5 get lBox2 CHECKBOX CAPTION "Checkbox 2"
@ 6,5 get cText
read
setpos(8,0)
? "box1=", lBox1, "box2=", lBox2
wait
```

Output:



```
☑ Checkbox 1
☐ Checkbox 2
any text
box1= .T. box2= .F.
Press any key to continue ...
```



```
[ ] Checkbox 1
[X] Checkbox 2
other text

box1= .F. box2= .T.

Press any key to continue ...
```

Example:

see <FlagShip_dir>/examples/getread*.prg

Classification: screen oriented i/o (via READ)

Compatibility: New in FS5

Handler Source: <FlagShip_dir>/system/checkboxhand.prg

Related: @..GET, READ,OBJ. CheckBox class

@...GET COMBOBOX

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
     GET <varN>
     COMBOBOX <aData> | USING <obj>
        [CAPTION <cCapt>]
        [COLDBOX <cFrame>]
        [COLOR <cColor>]
        [GUICOLOR <cGuiColor>]
        [DEFAULT <defN>]
        [DROPMARK <cDrop>]
        [ENABLE|DISABLE]
        [ERRORVALID <bError>]
        [FOCUS <fblock>]
        [HOTBOX <cFrame>]
        [MESSAGE <cText>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
        [SCROLLBAR]
        [SEND|GUISEND <snd>]
        [STATE <sBlock>]
        [TOOLTIP <cTip>]
        [USERMSG <cargo>]
        [USING <obj>]
        [VALID <lValid>]
        [WHEN <lWhen>]
```

Purpose:

Creates ComboBox widget and let it process via common READ. ComboBox is a special case of the ListBox and can be created also by using the @..GET..LISTBOX command with DROPDOWN clause.

Arguments:

<expN1>,<expN2>,expN3>,<expN4> are numeric expressions, specifying the top, left, bottom, right coordinates (in that order) of the open ComboBox widget. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

GET <varN> is a database field or a memory variable of numeric type specifying the start item (if > 0) in the list, and returning the selected position (or 0 on ESC).

COMBOBOX <aData> where the <aData> contains strings with combo box items.

COMBOBOX USING <obj> is an alternative syntax, specifying to use an already instantiated object <obj> of ComboBox class with assigned items. When the coordinates <expN1>...<expN4> are 0 or positive, they will overwrite previously set <obj> coordinates, negative coordinate let previous setting untouched.

Options:

The optional clauses of ComboBox are equivalent to Listbox, please refer to the @...GET..LISTBOX command.

Description:

The @...GET...COMBOBOX command uses the ComboBox class. You may add other class properties by e.g.

```
    Atail(GetList):<ComboBox_instance> := <value>
```

or by instantiating the object extra, set instances and using the USING <obj> clause in this @..GET..COMBOBOX command.

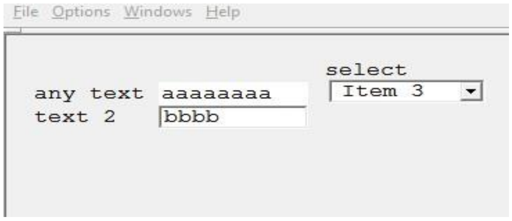
To open ComboBox, click on the drop-mark or use Cursor-Down, Space or # key, and TAB, shift-TAB or ^ key to close the box. These keys are user-modifiable by obj:Exec() or by assigning corresponding inkey-value(s) to

```
  _aGlobSetting[GSET_A_COMBO_OPEN ] := {K_DOWN, K_SPACE, 35} // def
  _aGlobSetting[GSET_A_COMBO_CLOSE] := {K_TAB, K_SH_TAB, 94} // def
```

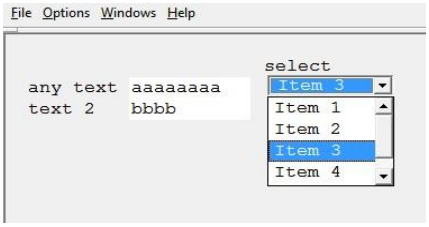
before instantiating the ComboBox by @..GET..COMBOBOX command.

Example:

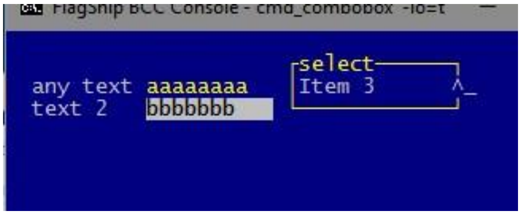
```
set font "Courier", 12
oapplic:Resize(12,40,,.T.)
set color to "W/B,N/W,GR+/B,N/W,GR+/B,GR+/B,R+/B,R+/B" // term only
cls
local nBox, cText := cText2 := space(10)
local aData := {"Item 1","Item 2", "Item 3", "Item 4", "other"}
@ 2,2 say "any text" get cText
@ 1,22,3,35 box B_SINGLE COLOR "GR+/B"
@ 1,23 say "select" COLOR "GR+/B"
@ 2,23,7,35 get nBox COMBOBOX aData
@ 3,2 say "text 2  " get cText2
read
```



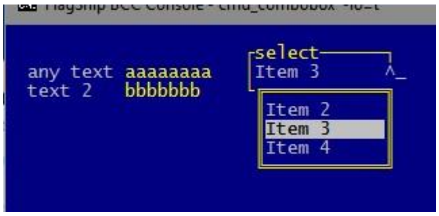
```
                            select
 any text  aaaaaaaa         Item 3
 text 2    bbbb
```



```
                            select
 any text  aaaaaaaa         Item 3
 text 2    bbbb             Item 1
                            Item 2
                            Item 3
                            Item 4
```



```
                     ┌select──────┐
 any text aaaaaaaa   Item 3       ^_
 text 2   bbbbbbb    └────────────┘
```



```
                     ┌select──────┐
 any text aaaaaaaa   Item 3       ^_
 text 2   bbbbbbb
                      Item 2
                      Item 3
                      Item 4
```

Classification:
screen oriented i/o (via READ)

Compatibility:
New in FS5

Handler Source:
<FlagShip_dir>/system/checkboxhand.prg

Related:
@..GET, READ, @..GET..LISTBOX, ComboBox class

@...GET LISTBOX

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
     GET <varN>
     LISTBOX <aData> | USING <obj>
        [CAPTION <cCapt>]
        [COLDBOX <cFrame>]
        [COLOR <cColor>]
        [GUICOLOR <cGuiColor>]
        [DEFAULT <defN>]
        [DROPDOWN]
        [DROPMARK <cDrop>]
        [ENABLE|DISABLE]
        [ERRORVALID <bError>]
        [FOCUS <fblock>]
        [HOTBOX <cFrame>]
        [MESSAGE <cText>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
        [SCROLLBAR]
        [SEND|GUISEND <snd>]
        [STATE <sBlock>]
        [TOOLTIP <cTip>]
        [USERMSG <cargo>]
        [USING <obj>]
        [VALID <lValid>]
        [WHEN <lWhen>]
```

Purpose:

Creates ListBox or ComboBox widget and let it process via common READ.

Arguments:

<expN1>,<expN2>,expN3>,<expN4> are numeric expressions, specifying the top, left, bottom, right coordinates (in that order) of the ListBox widget. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

GET <varN> is a database field or a memory variable of numeric type specifying the start item (if > 0) in the list, and returning the selected position (or 0 on ESC).

LISTBOX <aData> is one- or two-dimensional array. With one-dimensional, the array elements contain the displayed text. With two- dimensional, the aData[n,1] is the displayed text and aData[n,2] is a "hidden" item value available via obj:Value for the selected item, or obj:GetVal(pos) for any item. The current <obj> object is passed to the FOCUS and STATE code block.

LISTBOX USING <obj> is an alternative syntax, specifying to use an already instantiated object <obj> of ListBox or ComboBox class with assigned items.

Options:

CAPTION <cCapt> is a text explaining the ListBox

COLDBOX <cFrame> (considered in Terminal i/o mode only) specifies the frame displayed when the ListBox has no input focus. The default style is pre-defined in the global array element

```
  _aGlobSetting[GSET_T_C_COLDBOX] := B_SINGLE
```

which is used when the COLDBOX clause is omitted.

COLOR <cColor> (considered in Terminal i/o mode only) defines the color settings for the ListBox. The string may contain 8 color pairs:

Pair#	Used for	Default
1	Unselected items, without input focus	Standard
2	Selected item, without input focus	Unselected
3	Unselected items with input focus	Standard
4	Selected item with input focus	Enhanced
5	The list box's border	Border
6	The list box's caption	Standard
7	The list box caption's accelerator key	Background
8	The list box's drop-down button	Standard

For not specified pair, the default from current SetColor() is used

GUICOLOR <cGuiColor> (considered in GUI i/o mode only) defines the color settings for the ListBox. The string may contain 4 color pairs:

Pair#	Used for	Default
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

For not specified pair or for pair specified N/N, the default is used. Note that the standard background for selected item (with and without input focus) is usually set by the window manager and may hence differ according to the used platform. It is usually W+/RGB(49,106,195) = W+/#316AC3 in Windows, and W+/RGB(8,93,139) = W+/#085D8B in Linux/KDE.

DEFAULT <defN> set the GET <varN> to <defN> value if <varN> is NIL, empty() or of different type than <defN> which must be numeric.

DROPDOWN indicates to create ComboBox instead of ListBox.

DROPMARK <cDrop> (considered in Terminal i/o mode only) specifies the drop-down character displayed for ComboBox. The default style is pre-defined in the global array element `_aGlobSetting[GSET_T_C_COMBOMARK] := chr(31)` which is used when the DROPMARK clause is omitted.

ENABLE|DISABLE enable (default) or disable the item from READ processing

ERRORVALID <bError> specifies to use the <bError> code block to display post-validate error/failure

FOCUS <fblock> specifies a code block that is evaluated each time the ListBox receives focus. The code block receives two parameters, the current ListBox object, and the obj:HasFocus status.

HOTBOX <cFrame> (considered in Terminal i/o mode only) specifies the frame displayed when the ListBox has input focus. The default style is pre-defined in the global array element `_aGlobSetting[GSET_T_C_HOTBOX] := B_DOUBLE` which is used when the HOTBOX clause is omitted.

MESSAGE <cText> displays message <text> in status bar or in the SET MESSAGE line when the ListBox receives focus

PIXEL : the <expN1> and <expN2> are values in pixel

NOPIXEL : the <expN1> and <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN2> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL, log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

SCROLLBAR is available for Clipper compatibility only. In FlagShip, the scrollbar is used automatically when the list is larger than the available widget size.

SEND <instance>	} allows you to assign any valid class instance
GUISEND <instance>	} or method. Supported for Clipper compatibility.

STATE <sBlock> specifies a code block that is evaluated each time the ListBox selection changes, i.e. is checked or unchecked. The code block receives two parameters, the current ListBox object and the select status.

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the ListBox widget, even w/o focus

USERMSG <cargo> assigns the <cargo> value to the ListBox:Cargo instance

USING <obj> specify to use an already instantiated object <obj> of ListBox or ComboBox class. Optional only with Syntax 1, i.e. when LISTBOX <aData> is used. When the coordinates <expN1>...<expN4> are 0 or positive, they will overwrite previously set <obj> object coordinates; negative <expN> coordinate let previous <obj> setting untouched.

VALID <lValid> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding field. Should the expression return a .F. value, the cursor will remain on the current field. This feature is often used for lookups using post-processing functions. To determine the currently selected Listbox item number, use

```
item := GetActive():Buffer
```

WHEN <lWhen> (pre-validation) specifies an expression that must be satisfied in order to enter the ListBox during a READ

Description:

The @...GET...LISTBOX command uses the ListBox or ComboBox class. You may add other class properties by e.g.

```
 Atail(GetList):<ListBox_instance> := <value>
```

or by instantiating the object extra, set instances and using the USING <obj> clause in this @..GET..LISTBOX command.

The ListBox class is also used per default in Achoice().

To open the ComboBox, use the TAB or # key, and shift-TAB or ^ key to close the box. These keys are user-modifiable by obj:Exec(...) or by assigning corresponding inkey-value(s) to global variable `_aGlobSetting[GSET_A_COMBO_*]`, see Tuning below.

Tuning:

The <expN1>..<expN4> coordinates usually specifies the outer box frame, common for both GUI and Terminal i/o mode. If you wish in GUI mode these coordinates specify the inner box, set

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

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
```

where the defaults are set in <FlagShip_dir>/system/initio.prg API

The action on a key or mouse button press is defined in the user modifiable handler <FlagShip_dir>/system/listboxhand.prg. Mouse is supported in GUI mode only.

The default behavior on mouse button click is: - Left mouse click selects the ListBox item and leaves the ListBox to next GET (if any), same as press of the Enter or Space key. - Mid and right mouse button select the item but stays in the ListBox until the

Enter or Space key leaves it, except when the object instance SelectBySingleClick is set .T. (default is .F.) which then behaves same as left mouse click.

By assigning `_aGlobSetting[GSET_G_L_LISTBOX_SINGLE]:= .F.` you may avoid leaving ListBox by left mouse button. Assigning .F. to SelectBySpace instance prevent selection by Space key. The supported mouse buttons are specified in the array `_aGlobSetting[GSET_G_A_LISTBOX_MOUSE],` see <FlagShip_dir>/system/initio.prg

You may change the default ComboBox dropdown shortkey (Cursor-Down, Space or # key to open, and TAB, shift-TAB or ^ key to close) by assigning corresponding inkey-value(s) to

```
  _aGlobSetting[GSET_A_COMBO_OPEN ] := {K_DOWN, K_SPACE, 35} // def
  _aGlobSetting[GSET_A_COMBO_CLOSE] := {K_TAB, K_SH_TAB, 94} // def
```

before instantiating the ComboBox by @..GET..LISTBOX DROP command.

Example 1:

```
local ii, aItem, nListb, cTxt := "any text    "

set font "courier", 10
// _aGlobSetting[GSET_G_L_LISTBOX_BOX] := .F.  // coord = inner

aItem  := {}
nListb := 1
for ii := 1 to 20
   aadd(aItem, "Listbox (array) line#" + ltrim(ii))
next
@ 4, 5, 11,35 GET nListb LISTBOX aItem
@ 12,5 get cTxt
read
wait
```

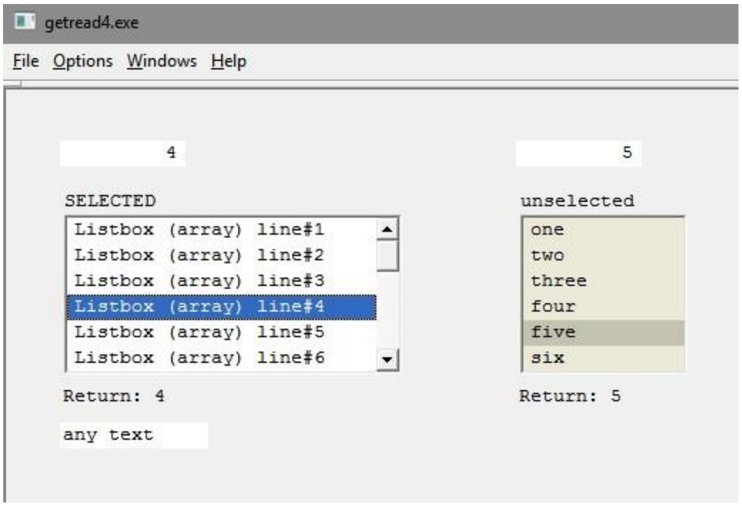
Example 2:

see <FlagShip_dir>/examples/getread4.prg (extract):

```
local ii, aItem, oBox, nListb1, nListb2, cTxt := "any text    "
#ifdef FS_WIN32
 local cGuiColor := "N/#ECE9D8,N/#C4C2B0,N/W+,W+/#316AC3"
#else
 local cGuiColor := "N/#DEDEDE,W+/#A0A0A0,N/W+,W+/#085D8B"
#endif

set font "courier", 10

aItem  := {}
nListb1 := 4
nListb2 := 2
for ii := 1 to 20
   aadd(aItem, "Listbox (array) line#" + ltrim(ii))
next
@ 2,5  get nListb1
@ 2,45 get nListb2
@ 4, 5, 11,35 GET nListb1 LISTBOX aItem  ;
              GUICOLOR (cGuiColor) valid(myReport(1,11.5,5))
```

```
getlist[len(getlist)]:Fblock := {|obj,on| showFocus(obj,on)}

@ 4, 45,11,60 GET nListb2 LISTBOX {"one","two","three", ;
                                   "four","five","six"} ;
              GUICOLOR (cGuiColor) valid(myReport(1,11.5,45))
oBox := getlist[len(getlist)]
oBox:Fblock := {|obj,on| showFocus(obj,on)}
oBox:SelectBySpace := .F.        // disable space selection
oBox:SelectBySingleClick := .T.  // MMB and RMB same as LMB

@ 13,5 get cTxt
read
wait

Function showFocus(obj, OnOff)
  @ obj:top -1, obj:left ;
      say if(OnOff, "SELECTED     ", "unselected     ")
return
Function myReport(mode,row,col)
  local rr := row(), cc := col(), obj := getactive()
  if mode == 0               // clear
     @ row,col say space(30)
  else                       // display curr. selection
     @ row,col say "Return: " + ltrim(obj:Buffer) + space(5)
  endif
  setpos(rr, cc)
return .T.
```

Output:



```
         4                               5

SELECTED                         unselected
Listbox (array) line#1           one
Listbox (array) line#2           two
Listbox (array) line#3           three
Listbox (array) line#4           four
Listbox (array) line#5           five
Listbox (array) line#6           six
Return: 4                        Return: 5

any text
```

Classification: screen oriented i/o (via READ)
Compatibility: New in FS5, available also (with less options) in CL53
Handler Source: <FlagShip_dir>/system/listboxhand.prg
Related: @..GET, READ, @..GET..COMBOBOX, ListBox and ComboBox classes

@...GET PUSHBUTTON

Syntax:

```
@ <expN1>,<expN2>
     GET <varL>
     PUSHBUTTON
        [CAPTION <cCapt>] | [IMAGE <cImage>]
        [COLOR <cColor>]
        [GUICOLOR <cColor>]
        [DEFAULT <defL>]
        [ENABLE|DISABLE]
        [ERRORVALID <bError>]
        [FOCUS <fblock>]
        [FONT <oFontObj> | <cFontName>[,<nFontSize>]]
        [HEIGHT <nHeight>]
        [IMAGE <cImage>]
        [MESSAGE <cText>]
        [NOTIFY <nBlock>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
        [SEND|GUISEND <snd>]
        [SKIP]
        [STATE <sBlock>]
        [STYLE <cFrame>]
        [TOOLTIP <cTip>]
        [USERMSG <cargo>]
        [USING <obj>]
        [VALID <lValid>]
        [WIDTH <nWidth>]
        [WHEN <lWhen>]
```

Purpose:

Creates PushButton widget and let it process via common READ.

Arguments:

<expN1>,<expN2> are numeric expressions, specifying the row and column coordinate of the push button widget. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

GET <varL> is a database field or a memory variable of logical type returning the status, i.e. TRUE if the push button was pressed.

PUSHBUTTON is a mandatory clause.

Options:

CAPTION <cCapt> is a text displayed in the push button

COLOR <cColor> (considered in Terminal i/o mode only) defines the color settings for the PushButton. The string may contain 4 color pairs:

Pair#	Used for	Default
1	push button w/o input focus	Unselected
2	push button with input focus, not pressed	Enhanced
3	push button with input focus, pressed	Enhanced
4	push button caption's accelerator key	Background

For not specified pair, the default from current SetColor() is used

GUICOLOR <cColor> (considered in GUI mode only) defines the color settings for the PushButton, similarly to COLOR clause

DEFAULT <defL> set the GET <varL> to <defL> value if <varL> is NIL, empty() or of different type than <defL> which must be logical.

ENABLE|DISABLE enable (default) or disable the item from READ processing. See also SKIP clause for partial disabling.

ERRORVALID <bError> specifies to use the <bError> code block to display post-validate error/failure

FOCUS <fblock> specifies a code block that is evaluated each time the PushButton receives focus. The code block receives two parameters, the current obj:HasFocus status and PushButton object.

FONT <oFontObj> displays button caption using font object <oFontObj>, e.g.
`FONT Font{<cFontName>[,<nFontSize>[,<cFontAttrib>]] }`
displays button caption using font name and optional font size and attributes

FONT <cFontName> displays button caption using font name <cFontName>

FONT <cFontName>,<nFontSize> displays button caption using font name and font name and font size

HEIGHT <nHeight> is the button height (in current Row/Col or pixel settings), default is one row (corresponding to FONT). The alternative is `oGet:Height(nHeight, lPixel)`, see example 1

IMAGE <cImage> fills the button with an image (.bmp, .gif, .png) named < cImage>. If the HEIGHT or WIDTH is not specified or is 0, the image is automatically scaled.

MESSAGE <cText> displays message <text> in status bar or in the SET MESSAGE line when the PushButton receives focus

NOTIFY <nBlock> specifies a code block that is evaluated each time the PushButton is pressed or clicked by mouse, to enable the application to react on the Enter or mouse button press. The code block takes one argument, the PushButton object self. Since the code block is evaluated immediately at mouse click on the button, even if the current GET object yet differs, it is not advisable to push key(s) via KEYBOARD within the Notify code block body; it may cause unexpected READ behavior. Instead, assign key value(s) to be processed to **objPush:OnClickKeys** instance. Alternatively, you may specify action to be taken next in READ by assigning GE_* value to the **objPush:OnClickAction** instance. The GE_* values are defined in getexit.fh and described in OBJ.Get:ExitState; e.g. GE_WRITE = 6 to save GETs by simulating press of Ctr-W key, or GE_ESCAPE = 7 to exit READ same as press on ESC key, or GE_TOP to skip to first item, etc. To be able to check READ exit, LASTKEY() is set K_CTRL_W on GE_WRITE and K_ESC on passing GE_ESCAPE. See example in section FUN.ReadSelect() and below.

PIXEL : the <expN1> and <expN2> are values in pixel

NOPIXEL : the <expN1> and <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN2> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by set(_SET_PIXEL,log) or set (_SET_COORD_UNIT,num). Apply for GUI mode only, ignored otherwise.

SEND <instance>	} allows you to assign any valid class instance
GUISEND <instance>	} or method. Supported for Clipper compatibility.

SKIP disables selecting this button by cursor movement within READ, the item is accessible by mouse click or by SELECT or ReadSelect() assignment from/within READ only. If you wish to generally disable this item, use DISABLE clause (or objPush:Enabled := .F.) instead.

STATE <sBlock> specifies a code block that is evaluated each time the PushButton selection changes, i.e. is pressed or released. The code block receives two parameters, the obj:Buffer indicating the button status (pressed/released) and the current PushButton object.

STYLE <cFrame> (considered in Terminal i/o mode only) specifies a character string that indicates the PushButton delimiters. The default style is pre-defined in the global array element `_aGlobSetting[GSET_T_C_PUSHB_STYLE] := "<>"` which is used when the STYLE clause is omitted.

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the PushButton widget, even w/o focus

USERMSG <cargo> assigns any <cargo> value to the PushButton:Cargo instance

USING <obj> specify to use an already instantiated object <obj> of PushButton class.

VALID <lValid> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding field. Should the expression return a .F. value, the cursor will remain on the current field. This feature is often used for lookups using post-processing functions. At this stage, as opposite to NOTIFY, you also may push keys by KEYBOARD in UDF, the code block however need to return logical.

WIDTH <nWidth> is the button width (in current Row/Col or pixel settings), default is the size of <cCaption>. The alternative is oGet:Width(nWidth,lPixel), see example 1

WHEN <lWhen> (pre-validation) specifies an expression that must be satisfied in order to enter the PushButton during a READ. At this stage, as opposite to NOTIFY, you also may push keys by KEYBOARD in UDF, the code block however need to return logical.

Description:

The @..GET..PUSHBUTTON command uses the PushButton class. You may add other class properties by e.g.

```
    Atail(GetList):<PushButton_instance> := <value>
```

or by instantiating the object extra, set instances and using the USING <obj> clause in this @..GET..PUSHBUTTON command.

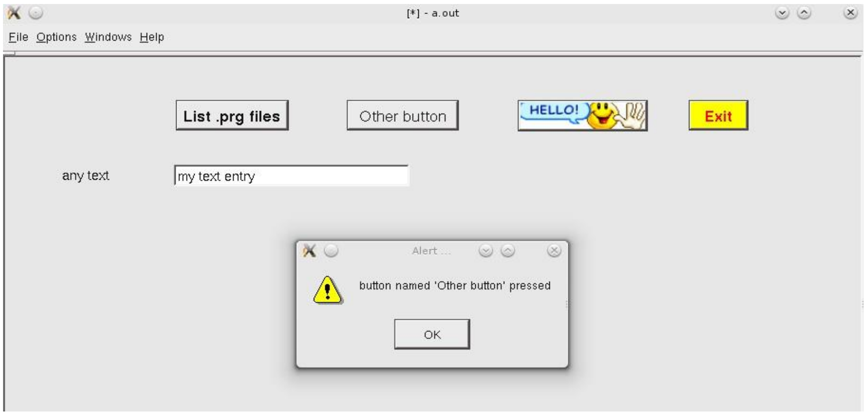
Pressing the button by mouse click (GUI only) or by Enter, space, X, Y, T or P keys calls the codeblock specified by STATE option, which then performs the requested program action. The codeblock may also perform e.g. KEYBOARD chr(K_DOWN) or chr(K_ESC) to skip to next @..GET field or to terminate the READ, otherwise the GET field is not left. But since the code block my be entered twice (one time at key press and once on key release), better is to use obj:OnClickAction instead of KEYBOARD, see NOTIFY above. If there is neither STATE nor FOCUS codeblock, the default behavior at pressing the button simulates Enter key to continue READ process in next GET field.

Example 1:

```
#include "getexit.fh"
 set font "Corier",10
 LOCAL lPush := lPush2 := lPush3 := lExit := .F.
 LOCAL cText := space(20)
 LOCAL oFont := Font{"Arial",12,"B"}
 @ 2,15 GET lPush PUSHBUTTON CAPTION "List .prg files" ;
              STATE {|push, obj| myList(push, obj) } ;
              FONT oFont ;    // or FONT font{"Arial",12}
              TOOLTIP "List sources in current directory"
 @ 2,30 GET lPush2 PUSHBUTTON CAPTION "Other button" SKIP ;
              NOTIFY {|obj| myUdf(obj) } ;
              FONT "Arial",12 ;    // or FONT font{"Arial",12}
              TOOLTIP "Other button accessible by mouse only"
 @ 2,45 GET lPush3 PUSHBUTTON NOTIFY {|obj| myUdf3(obj) } ;
              FONT oFont
 // Atail(getlist):Width  (120, .T.)        // width = 120 pixel
 Atail(getlist):Height (35, .T.)            // height = 35 pixel
 Atail(getlist):SetImage("anim3.bmp")       // auto-scale to width
 Atail(getlist):Display()                   // redisplay button

 @ 5, 5 SAY "any text"
 @ 5,15 GET cText
 @ 2,60 GET lExit PUSHBUTTON CAPTION "Exit" ;
```

```
       NOTIFY {|obj| lExit := .T., obj:OnClickKeys := chr(K_ESC) } ;
       GUICOLOR "R+/GR+" ;
       FONT FontNew("Arial",12,"B") ;
       SKIP TOOLTIP "Accessible by mouse click only"
READ
setpos(10,0)
? "lastkey()=",ltrim(lastkey())
wait

FUNCTION myList(lPressed, oPush) // for Button#1
   local aDir, iWin, ii
   if !lPressed                  // button released:
       return              // nothing to do
   endif
   aDir := Directory("*.prg")
   // uses sub-window via FS2 toolbox
   iWin := Wopen(1,28, min(Maxrow()-1,len(aDir)+6), 55)
   for ii := 1 to len(aDir)
      @ ii-1, 1 say aDir[ii,1]
   next
   ?
   wait
   Wclose(iWin)                     // close sub-window
   oPush:OnClickAction := GE_WRITE  // write & exit READ
return

FUNCTION myUdf(oPush)               // for Buttons#2 and #4
   alert("button named '" + oPush:Caption + "' pressed")
return

FUNCTION myUdf3(oPush)              // for Button#3
  alert("button 3 pressed")
  oPush:OnClickAction := GE_ENTER   // go to next GET
return
```

Output:



button named 'Other button' pressed

Example 2: see FUN.ReadSelect(), FUN.PushButton() and <FlagShip_dir>/examples/getread3.prg

Classification:

screen oriented i/o (via READ)

Compatibility:

New in FS5, available also (with less options) in CL53

Handler Source:

<FlagShip_dir>/system/pushbutthand.prg

Related:

@..GET, READ, PushButton(), PushButton class

@...[SAY..] GET RADIOBUTTON

Syntax 1:

```
@ <expN1>,<expN2>
     GET <varL>
     RADIOBUTTON
        [CAPTION <cCapt>]
        [COLOR <cColor>] | [GUICOLOR <cColor>]
        [DEFAULT <lDef>]
        [ENABLE|DISABLE]
        [ERRORVALID <bError>]
        [FOCUS <fblock>]
        [HEIGHT <nHeight>]
        [MESSAGE <cText>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
        [SEND|GUISEND <snd>]
        [STATE <sBlock>]
        [STYLE <cStyle>]
        [TOOLTIP <cTip>]
        [USERMSG <cargo>]
        [USING <obj>]
        [VALID <lValid>]
        [WHEN <lWhen>]
```

Syntax 2:

```
@ <expN1>,<expN2>
     SAY <cSaytext>
        [PICTURE <cSayPict>]
        [COLOR <cSayColor>] | [GUICOLOR <cColor>]
        [FONT <cFont> ]
     GET <varL>
     RADIOBUTTON
        [CAPTION <cCapt>]
        ...etc...
```

Purpose:

Creates RadioButton widget and let it process via common READ. Usually, the radio button is not used stand alone, but is grouped in RadioGroup to get an exclusive ON status of one button of the group.

Arguments:

<expN1>, <expN2> are numeric expressions, specifying the row and column coordinates. With Syntax 1: coordinates of the widget. With Syntax 2: coordinate of the capture text, the widget column coordinate is the over-next column behind the text <cSaytext> end, same as in @..SAY..GET. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

SAY <cSaytext> is a text caption identifying the RadioButton on the screen. Better, common practice is to use the CAPTION clause.

GET <varL> is a database field or a memory variable of logical type storing the "on" status of the RadioButton.

RADIOBUTTON clause is mandatory here.

Options:

CAPTION <cCapt> is a text explaining the radio button

COLOR <cSayColor> is an optional color specification for the @..SAY text.

COLOR <cGetColor> (considered in Terminal i/o mode only) defines the color settings for the radio button. The string may contain 8 color pairs:

Pair#	Used for	Default
1	Radio button without input focus, unselected	Unselected
2	Radio button without input focus, selected	Unselected
3	Radio button with input focus, unselected	Unselected
4	Radio button with input focus, selected	Enhanced
5	Radio button's caption	Standard
6	Radio button caption's accel. key w/o focus	Standard
7	Radio button caption's accel. key with focus	Background
8	Radio button and caption, disabled	Border

For not specified pair, the default from current SetColor() is used

DEFAULT <lDef> set the GET <varL> to <lDef> value if <varL> is NIL, empty() or of different type than <lDefa> which must be logical.

ENABLE|DISABLE enable (default) or disable the item from READ processing

ERRORVALID <bError> specifies to use the <bError> code block to display post-validate error/failure

FOCUS <fblock> specifies a code block that is evaluated each time the Radio button receives focus. The code block receives two parameters, the current RadioButton object, and the oBox:HasFocus status.

HEIGHT <nHeight> is the button height (in current Row/Col or pixel settings), default is one row. The alternative is `oGet:Height(nHeight,lPixel)`,

MESSAGE <cText> displays message <text> in status bar or in the SET MESSAGE line when the RadioButton receives focus

PICTURE <cSayPict> is the optional picture of @..SAY text

PIXEL : the <expN1> and <expN2> are values in pixel

NOPIXEL : the <expN1> and <expN2> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN2> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL, log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

SEND <instance> } allows you to assign any valid class instance
GUISEND <instance> } or method. Supported for Clipper compatibility.

STATE <sBlock> specifies a code block that is evaluated each time the RadioButton state changes, i.e. is checked or unchecked. The code block receives two parameters, the current RadioButton object, and the oBox:Buffer status.

STYLE <cStyle> (considered in Terminal i/o mode only) specifies a character string that indicates the RadioButton delimiter characters and status display. The default style is pre-defined in the global array element

```
   _aGlobSetting[GSET_T_C_RADBUT_STYLE] := "( *)"
```

which is used when the STYLE clause is omitted.

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the RadioButton widget, even w/o focus

USERMSG <cargo> assigns the <cargo> value to the RadioButton:Cargo instance

USING <obj> use already instantiated RadioButton object <obj>, avoid a new creation/instantiation

VALID <lValid> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding GET. Should the expression return a .F. value, the cursor will remain on the current field. This feature is often used for lookups using post-processing functions.

WHEN <lWhen> (pre-validation) specifies an expression that must be satisfied in order to enter the RadioButton during a READ

Description:

The @...GET...RADIOBUTTON uses the RadioButton class. You may add other class properties by e.g. Atail(GetList):<RadioButton_instance> := <value> or by instantiating the object extra, set instances and using the USING <obj> clause in this @..GET..RADIOBUTTON command.

Tuning:

The action on a key or mouse button press is defined in the user modifiable handler *<FlagShip_dir>/system/radbutthand.prg*. Mouse is supported in GUI mode only. The default behavior on mouse button click is: Left mouse click selects/clears the RadioButt and leaves the button to next GET (if any), same as the +,-,space,x,y,t,n,f key press. Mid and right mouse button toggles the button but stays in the RadioButton until the corresponding key leaves it. Space or 'x' key toggles the status, the '+','y','t' key sets the RadButton on, and '-','n','f' key press sets it off. Return only skips to next field.

By assigning `_aGlobSetting[GSET_G_L_RADBUT_SINGLE] := .F.` you may avoid leaving RadioButton by left mouse button. The supported mouse buttons are specified in `_aGlobSetting[GSET_G_A_RADBUT_MOUSE]` array, see <FlagShip_dir>/system/initio.prg

Mouse click on RadioButton caption (if any) may or may not toggle it, same as click within the box. You may control it by setting

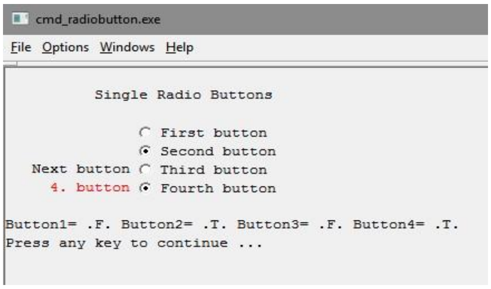
```
   _aGlobSetting[GSET_G_L_RADBUT_SELCAPT] := .T.   // default
```

You may control the RadioButton pixel adjustment by elements in the global variable `_aGlobSetting[GSET_G_*_RADBUT*]`, see details in <FlagShip_dir>/system/initio.prg

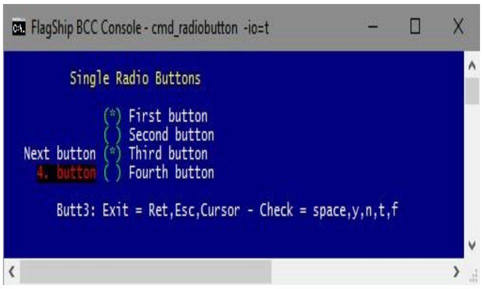
Example 1:

```
set font "courier",10
oApplic:Resize(11,70,,.T.)

LOCAL lRadio1, lRadio2, lRadio3, lRadio4
LOCAL cMsg := " Exit = Ret,Esc,Cursor - Check = space,y,n,t,f"
SET MESSAGE TO 8 CENTER
SET COLOR TO "W+/B,W/B,R+/B,R/B,G+/B,R+/B,B/BG"
// SET GUICOLOR ON
cls

col := 15
@ 1,10 say "Single Radio Buttons " color ("GR+/B")

@ 3,col GET lRadio1 RADIOBUTTON ;
        CAPTION "First button" ;
        MESSAGE "Butt1:" + cMsg

@ 4,col GET lRadio2 RADIOBUTTON ;
        CAPTION "Second button" ;
        MESSAGE "Butt2:" + cMsg

@ 5,col-12 SAY "Next button" GET lRadio3 RADIOBUTTON ;
        CAPTION "Third button" ;
        MESSAGE "Butt3:" + cMsg

@ 6,col-10 SAY "4. button" COLOR "R+" GUICOLOR "R+";
        GET lRadio4 RADIOBUTTON ;
        CAPTION "Fourth button" ;
        MESSAGE "Butt4:" + cMsg

READ
setpos(7,0)
? "Button1=",lRadio1, "Button2=",lRadio2, ;
  "Button3=",lRadio3,  "Button4=",lRadio4
wait
```

Output:



```
         Single Radio Buttons

              ( ) First button
              (•) Second button
   Next button ( ) Third button
     4. button (•) Fourth button

Button1= .F. Button2= .T. Button3= .F. Button4= .T.
Press any key to continue ...
```



```
       Single Radio Buttons

            (*) First button
            ( ) Second button
Next button (*) Third button
  4. button ( ) Fourth button

     Butt3: Exit = Ret,Esc,Cursor - Check = space,y,n,t,f
```

Example 2:

see <FlagShip_dir>/examples/getread3.prg, radiocheckbox.prg

Classification:

screen oriented i/o (via READ)

Compatibility:

New in FS5

Handler Source:

<FlagShip_dir>/system/radbutthand.prg

Related:

@..GET, READ, @..GET..RADIOGROUP, RadioButton class

@...GET RADIOGROUP

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
     GET <varN>
     RADIOGROUP <aData> | USING <obj>
        [CAPTION <cCapt>]
        [COLDBOX <cFrame>]
        [COLOR <cColor>]
        [DEFAULT <defN>]
        [ENABLE|DISABLE]
        [NONEXCLUSIVE]
        [ERRORVALID <bError>]
        [FOCUS <fblock>]
        [HOTBOX <cFrame>]
        [MESSAGE <cText>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
        [SEND|GUISEND <snd>]
        [STATE <sBlock>]
        [TOOLTIP <cTip>]
        [USERMSG <cargo>]
        [USING <obj>]
        [VALID <lValid>]
        [WHEN <lWhen>]
```

Purpose:

Creates RadioGroup widget and let it process via common READ. The radio group summarizes radio buttons and allows to exclusively select one button at a time. You may change this behavior to non- exclusive choice allowing multiple button selection by assigning Atail(GetList):Exclusive := .F. or by using already instantiated Radio-Group object and Syntax 2.

Arguments:

<expN1>,<expN2>,expN3>,<expN4> are numeric expressions, specifying the top, left, bottom, right coordinates (in that order) of the RadioGroup widget. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

GET <varN> is a database field or a memory variable of numeric type specifying the pre-selected button (if > 0) in the list, and returning the selected position (or 0 on ESC).

RADIOGROUP <aData> is one-dimensional array containing the buttons capture (string text) or RadioButton objects. You may instantiate the row/col of each

RadioButton absolutely or relative if the coordinate is negative. In such a case, row/col == -1 is 1st row/column of the RadioGroup, -2 is the 2nd row or column of the RadioGroup, and so forth.

RADIOGROUP USING <obj> is an alternative syntax, specifying to use an already instantiated object <obj> of RadioGroup class with assigned RadioButton items. You may use absolute or relative RadioButton coordinates, see above.

Options:

CAPTION <cCapt> is a text explaining the RadioGroup

COLDBOX <cFrame> (considered in Terminal i/o mode only) specifies the frame displayed when the RadioGroup has no input focus. The default style is pre-defined in the global array element `_aGlobSetting[GSET_T_C_COLDBOX] := B_SINGLE` which is used when the COLDBOX clause is omitted.

COLOR <cColor> (considered in Terminal i/o mode only) defines the color settings for the RadioGroup. The string may contain 3 color pairs:

Pair#	Used for	Default
1	Radio group border	Border
2	Radio group caption	Standard
3	Radio group capt key	Background

For not specified pair, the default from current SetColor() is used

DEFAULT <defN> set the GET <varN> to <defN> value if <varN> is NIL, empty() or of different type than <defN> which must be numeric.

ENABLE|DISABLE enable (default) or disable the item from READ processing

NONEXCLUSIVE The default RadioGroup behavior is "exclusive", which allows to set only one button within the group. When specifying the NONEXCLUSIVE clause, you may select any RadioButton within the group and/or disable it. It is equivalent to oGetItem:Exclusive := .F. assignment. In non-exclusive mode, the return value is the first selected item, see example below for determining all of them. In the default handler <FlagShip_dir>/system/radgrouphand.prg, the LeftMouse Button selects and RightMouseButton de-selects the status.

ERRORVALID <bError> specifies to use the <bError> code block to display post-validate error/failure

FOCUS <fblock> specifies a code block that is evaluated each time the RadioGroup receives focus. The code block receives two parameters, the current RadioGroup object, and the obj:HasFocus status.

HOTBOX <cFrame> (considered in Terminal i/o mode only) specifies the frame displayed when the RadioGroup has input focus. The default style is pre-defined in the global array element `_aGlobSetting[GSET_T_C_HOTBOX] := B_DOUBLE` which is used when the HOTBOX clause is omitted.

MESSAGE <cText> displays message <text> in status bar or in the SET MESSAGE line when the RadioGroup receives focus

PIXEL : the <expN1> ... <expN4> are values in pixel

NOPIXEL : the <expN1> ... <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN2> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_CO-ORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

SEND <instance> } allows you to assign any valid class instance
GUISEND <instance> } or method. Supported for Clipper compatibility.

STATE <sBlock> specifies a code block that is evaluated each time the RadioGroup selection changes, i.e. if one radio button is set ON or OFF. The code block receives two parameters, the current RadioGroup object, and the obj:Buffer content specifying the selected position of the button. To check the button status, you may determine it from the radio button object available via obj:GetItem():Buffer or obj:GetItem (pos):Buffer

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the RadioGroup widget, even w/o focus

USERMSG <cargo> assigns the <cargo> value to the RadioGroup:Cargo instance

USING <obj> specify to use an already instantiated object <obj> of RadioGroup class. Optional only with Syntax 1, i.e. when the clause RADIOGROUP <aData> is used.

VALID <lValid> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding field. Should the expression return a .F. value, the cursor will remain on the current field. This feature is often used for lookups using post-processing functions.

WHEN <lWhen> (pre-validation) specifies an expression that must be satisfied in order to enter the RadioGroup during a READ

Description:

The @...GET...RADIOGROUP command uses the RadioGroup class. You may add other class properties by e.g. Atail(GetList):<RadioGroup_instance> := <value> or by instantiating the object extra, set instances and using the USING <obj> clause in this @..GET..RADIOGROUP command. Note that some instances/settings apply after displaying the buttons.

Tuning:

The action on a key or mouse button press is defined in the user modifiable handler <FlagShip_dir>/system/radgrouphand.prg. Mouse is supported in GUI mode only. The default behavior on mouse button click is: Left mouse click selects the RadioButton and leaves the RadioGroup to next GET (if any), same as the

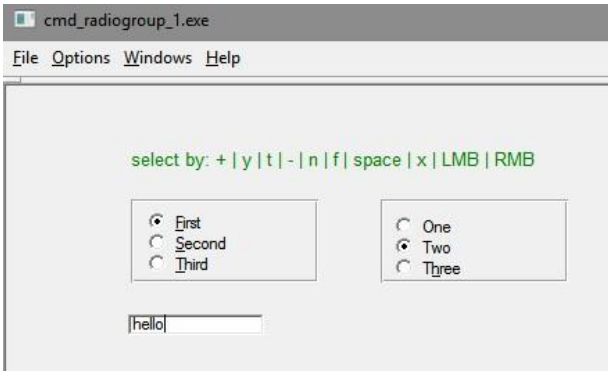
`+,-,space,x,y,t,n,f` key press. Mid and right mouse button select the button but stays in the RadioGroup until the corresponding key leaves it. With non-exclusive RadioGroup, left, mid and right mouse click toggles the button on/off status, same as space or 'x' key, the +,y,t key sets the RadioButton on, and `-,n,f` key press sets it off.

By assigning `_aGlobSetting[GSET_G_L_RADBUT_SINGLE] := .F.` you may avoid leaving RadioGroup by left mouse button. The supported mouse buttons are specified in `_aGlobSetting[GSET_G_A_RADBUT_MOUSE]` array, see <FlagShip_dir>/ system/initio.prg

Example 1:

```
local nButt := 1, nButt2 := 2, cTxt := space(10)
@ 3,10 say "select by: + | y | t | - | n | f |space| x | LMB | RMB";
            GUICOLOR "G" FONT "Arial",10
 @ 5,10,9,25 GET nButt RADIOGROUP ;
                { RadioButton{-1,-1, "&First", ""} , ;
                  RadioButton{-2,-1, "&Second", ""}, ;
                  RadioButton{-3,-1, "&Third", ""} }
 @ 5,30,9,45 GET nButt2 RADIOGROUP {"One","Two","T&hree"}
 @ 11,10 GET cTxt
 READ
 setpos(15,0)
 ? "Selected buttons=", ltrim(nButt), "and", ltrim(nButt2)
 wait
```

Output:

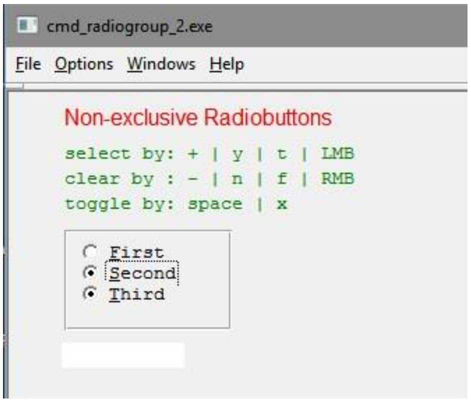

cmd_radiogroup_1.exe
File Options Windows Help
select by: + | y | t | - | n | f | space | x | LMB | RMB
First
Second
Third
One
Two
Three
hello

Example 2:

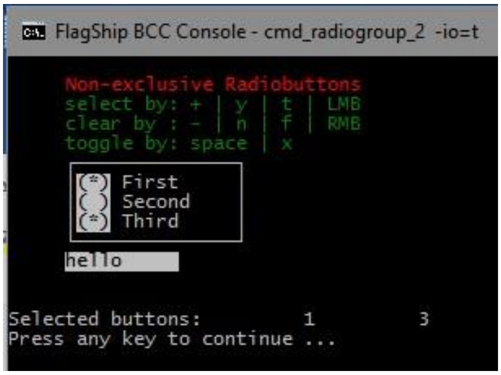
```
 local aGroup := array(3), nButt := 2, cTxt := space(10)
 local aButtons := {}
 aGroup[1] := RadioButton{-1,-1, "&First", ""}
 aGroup[2] := RadioButton{-2,-1, "&Second", ""}
 aGroup[3] := RadioButton{-3,-1, "&Third", ""}

 set font "courier", 10
 oApplic:Resize(25,80,,.T.)
 @ 0.5,5 say "Non-exclusive Radiobuttons" ;
             GUICOLOR "R+" FONT "Arial",12
 @ 2,5 say "select by: + | y | t | LMB" GUICOLOR "G"
 @ 3,5 say "clear by : - | n | f | RMB" GUICOLOR "G"
 @ 4,5 say "toggle by: space | x"       GUICOLOR "G"
```

```
@ 5,5,9,20 GET nButt RADIOGROUP aGroup NONEXCL ;
                     VALID CheckButtons(@aButtons)
aGroup[3]:Checked := .T.            // set addit. button 3 on
@ 10,5 GET cTxt
READ

setpos(12,0)
? "Selected buttons:"
aeval(aButtons, {|x| qqout(x)})
wait

FUNCTION CheckButtons(arr)          // called in Valid()
local ii, obj := GetActive()
arr := {}
if IsObjProperty(obj, 2, "exclusive") .and. !obj:Exclusive
   for ii := 1 to obj:ItemCount
      if obj:getitem(ii):Checked    // RadioButton item
         aadd(arr, ii)
      endif
   next
endif
return .T.
```

Output:



```
Non-exclusive Radiobuttons
select by: + | y | t | LMB
clear by : - | n | f | RMB
toggle by: space | x
```



```
Non-exclusive Radiobuttons
select by: + | y | t | LMB
clear by : - | n | f | RMB
toggle by: space | x
(*) First
( ) Second
(*) Third
hello

Selected buttons:        1        3
Press any key to continue ...
```

Example 3:

see <FlagShip_dir>/examples/getread3.prg, radiocheckbox.prg

Classification:

screen oriented i/o (via READ)

Compatibility:

New in FS5, available also (with less options) in CL53

Handler Source:

<FlagShip_dir>/system/radgrouphand.prg

Related:

@..GET, READ, @..GET..COMBOBOX, RadioGroup and ComboBox classes

@...GET TBROWSE

Syntax:

```
@ <expN1>,<expN2>,<expN3>,<expN4>
      GET <var>
      TBROWSE [USING] <obj>
         [CAPTION <cCapt>]
         [COLDBOX <cFrame>]
         [COLOR <cColor>]
         [ENABLE|DISABLE]
         [ERRORVALID <bError>]
         [HOTBOX <cFrame>]
         [MESSAGE <cText>]
         [PIXEL|NOPIXEL]
         [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN5>)]
         [SEND|GUISEND <snd>]
         [TOOLTIP <cTip>]
         [USERMSG <cargo>]
         [VALID <lValid>]
         [WHEN <lWhen>]
```

Purpose:

Creates TBrowse widget and let it process via common READ.

Arguments:

<expN1>,<expN2>,expN3>,<expN4> are numeric expressions, specifying the top, left, bottom, right coordinates (in that order) of the TBrowse widget. In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

GET <var> is a memory variable of any type not explicitly used but required for READ.

TBROWSE [USING] <obj> specifies the already instantiated object <obj> of TBrowse class with assigned TbColumn items.

Options:

CAPTION <cCapt> is not used by @..GET..TBROWSE but is supported for cross-compatibility purpose to other @..GET commands only.

COLDBOX <cFrame> (considered in Terminal i/o mode only) specifies the frame displayed around the TBrowse widget, containing either zero or at least eight characters (e.g. the B_SINGLE constant in box.fh). The default is no frame, see Tbrowse:Border instance.

COLOR <cColor> (considered in Terminal i/o mode only) defines the color settings for the TBrowse. Unlike in other @..GET.. commands, this clause assigns the

`Tbrowse:ColorSpec` instance which can contain many different color pairs as you need and the pair is selected thereof via `Tbrowse:ColorBlock`.

ENABLE|DISABLE enable (default) or disable the item from READ processing

ERRORVALID <bError> specifies to use the <bError> code block to display post-validate error/failure

HOTBOX <cFrame> (considered in Terminal i/o mode only) is equivalent to the COLDBOX <cFrame> clause and supported for cross-compatibility purpose to other @..GET commands only.

MESSAGE <cText> displays message <text> in status bar or in the SET MESSAGE line when the TBrowse receives focus

PIXEL : the <expN1> ... <expN4> are values in pixel

NOPIXEL : the <expN1> ... <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN5> specifies unit for <expN1> .. <expN4> coordinates. The <expN5> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_CO-ORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

SEND <instance> } allows you to assign any valid class instance
GUISEND <instance> } or method. Supported for Clipper compatibility.

TOOLTIP <cTip> (GUI only) short pop-up message/info displayed when mouse cursor is over the TBrowse widget, even w/o focus

USERMSG <cargo> assigns the <cargo> value to the TBrowse:Cargo instance

VALID <lValid> (post-validation) is a logical expression (or UDF returning a logical value) which is evaluated whenever the user attempts to leave the corresponding field. Should the expression return a .F. value, the cursor will remain on the current field. This feature is often used for lookups using post-processing functions.

WHEN <lWhen> (pre-validation) specifies an expression that must be satisfied in order to enter the TBrowse during a READ

Description:

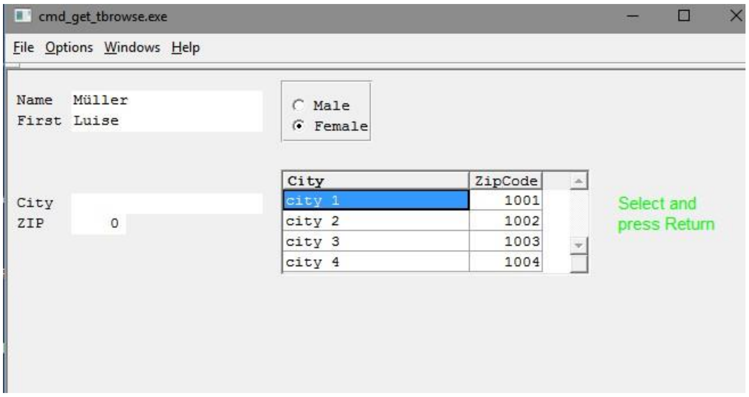
The @...GET...TBROWSE command uses the TBrowse class. To exit the Tbrowse, click TAB or Shift-Tab.

Example :

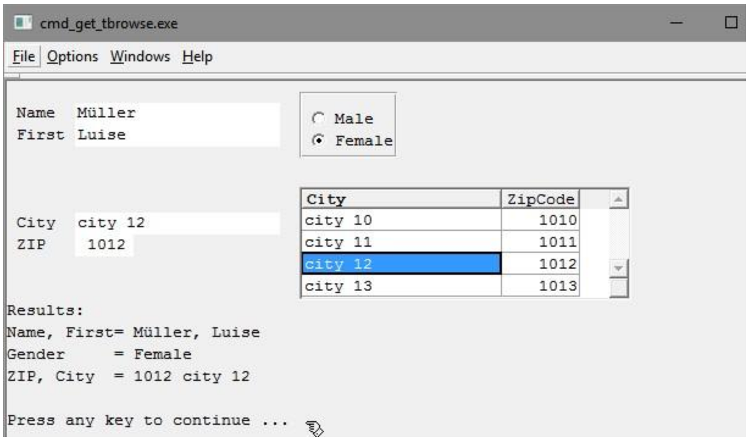
```
#include "box.fh"           // for terminal i/o @..box definitions
#include "tbrowse.fh"
local ii, oBr, oTbcol, myArray := {}
local aHeader := {"City","ZipCode" }

SET FONT "Courier", 10           // default font
```

```
oApplic:Resize(25, 80, , .T.)    // resize application screen

/* Create Tbrowse object
 */
for ii := 1 to 100
  aadd(myArray, {padr("city " + ltrim(ii),20), ii + 1000 } )
next
oBr := TbrowseArr(5,30,9,63, NIL, NIL, "My Browse")
oBr:UserArray := myArray
oBr:ReadOnly  := .T.
oBr:SetKey(K_ENTER, {|oBr,key| TBR_EXIT})
for ii := 1 to len(myArray[1])
  oTbcol := TbColumnNew(aHeader[ii], .T.) // use def.array block
  oBr:AddColumn(oTbcol)
next

/* GET/READ
 */
local cName := space(20), cFirst := space(20), nMale, nTemp
local cCity := space(20), nZip := 0, nMsg := 6

@ 1,1 SAY "Name " GET cName
@ 2,1 SAY "First" GET cFirst
@ 0,30,3,40 GET nMale RADIOGROUP {"Male","Female"} ;
            VALID DisplMsg(nMsg)
@ 4,30,7,50 GET nTemp TBROWSE oBr ;
            VALID {|| fillit(oBr, @cCity, @nZip, nMsg)}
@ 6,1 say "City " GET cCity
@ 7,1 say "ZIP  " GET nZip PICTURE "99999"
READ

setpos(9,0)
? "Results:"
? "Name, First=", trim(cName)+ ",", trim(cFirst)
? "Gender     =", if(nMale == 1, "Male", "Female")
? "ZIP, City  =", ltrim(nZip), trim(cCity)
?
wait

/* On Tbrowse exit, fill GET variables
 */
function fillit(oBr, cCity, nZip, nMsgRow)
if Lastkey() != K_ESC
  cCity := oBr:Data(1)
  nZip  := oBr:Data(2)
endif
@ nMsgRow,67 clear to nMsgRow+1,maxcol()
KEYBOARD chr(9)     // TAB = next GET
return .T.

/* Display message for Tbrowse
 */
function DisplMsg(nMsgRow)
@ nMsgRow,67   say "Select and"   GUICOLOR "G+" FONT "Arial",11
@ nMsgRow+1,67 say "press Return" GUICOLOR "G+" FONT "Arial",11
return .T.
```

Output:



```
Name   Müller
First  Luise
                        ◯ Male
                        (•) Female

City                    City        ZipCode
ZIP       0             city 1         1001
                        city 2         1002
                        city 3         1003
                        city 4         1004
```



```
Name   Müller
First  Luise
                        ◯ Male
                        (•) Female

City   city 12          City        ZipCode
ZIP     1012            city 10        1010
                        city 11        1011
                        city 12        1012
                        city 13        1013
Results:
Name, First= Müller, Luise
Gender      = Female
ZIP, City   = 1012 city 12

Press any key to continue ...
```

Classification:

screen oriented i/o (via READ)

Compatibility:

New in FS5, available also (with less options) in CL53

Handler Source:

<FlagShip_dir>/system/tbrowsedbhand.prg and tbrowsehand.prg

Related:

@..GET, READ, TBrowseArr(), TBrowseDb(), TBrowse class

@...TO

Syntax:

```
@ <expN1>,<expN2>
     TO <expN3>,<expN4>
        [DOUBLE]
        [COLOR <expC5>]
        [GUICOLOR <expC6>]
        [PRINTCOLOR <expC7>]
        [PIXEL|NOPIXEL]
        [UNIT=ROWCOL|PIXEL|MM|CM|INCH|DOT|(<expN8>)]
```

Purpose:

Draws boxes on the screen using single or double lines using the IBM-PC8 semi-graphic character set.

Arguments:

<**expN1**...**expN4**> are the coordinates for the upper, left and lower, right corners respectively. If <expN1> and <expN3> are the same, a horizontal line is drawn. If <expN2> and <expN4> are the same, a vertical line is drawn. The <expN3> and <expN4> are limited by MAXROW() and MAXCOL(). In GUI mode, you may use numeric values with decimal fractions for row and column, which are then rounded to integer in Terminal i/o mode. To set coordinates at exact pixel value, use the PIXEL clause (or enable SET PIXEL ON). If PIXEL or UNIT is not set, the GUI coordinate is internally calculated in pixel from current SET FONT (or oApplic:Font)

Options:

DOUBLE: If this clause is specified, a double-line box or otherwise a single-line box is drawn.

COLOR <expC5> defines the color string (see SET COLOR) in which to display the box lines. If not specified, the box is drawn using the current color setting. The "standard" color pair is used.

GUICOLOR <expC6> specifies the color of box lines in GUI mode. Only the first color pair (standard) is significant. If GUICOLOR is set, this color is used in GUI mode regardless the current SET GUICOLOR on/off. If omitted, default color is used. This clause apply for GUI mode only, and is ignored otherwise.

PRINTCOLOR <expC7> specifies the color for printing. If not given, GUICOLOR is used also for printer. Considered only in GUI mode when SET GUIPRINT is ON, or with PrintGui(.T.), and ignored otherwise.

PIXEL : the <expN1> ... <expN4> are values in pixel

NOPIXEL : the <expN1> ... <expN4> are row/col values

If [PIXEL|NOPIXEL] is not specified: the current SET PIXEL status is used. PIXEL is shortcut for UNIT=PIXEL, NOPIXEL is shortcut for UNIT=ROWCOL

UNIT=ROWCOL or PIXEL or MM or CM or INCH or DOT or <expN8> specifies unit for <expN1> .. <expN2> coordinates. The <expN8> is parenthesed numeric value in range 0 to 5 (i.e. UNIT_ROWCOL to UNIT_DOT). If the UNIT=... clause is not specified, default is row/col, or the current setting by `set(_SET_PIXEL,log)` or `set(_SET_COORD_UNIT,num)`. Apply for GUI mode only, ignored otherwise.

Description:

@...TO draws a single or double line box on the screen. The cursor is set into the boxed region at <expN1> +1, <expN2> +1. For customized or filled boxes, use the @...BOX command instead.

It draws always in Term mode, in GUI only if SET GUITRANSL LINES or SET GUITRANSL BOX is ON. See also @..DRAW for GUI only drawing (w/o SET GUITRANSL required settings).

Tuning:

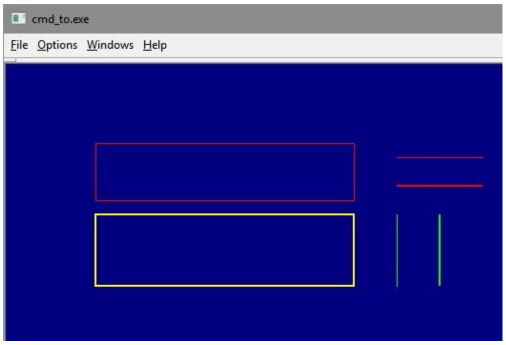
The line width in GUI mode can be set by

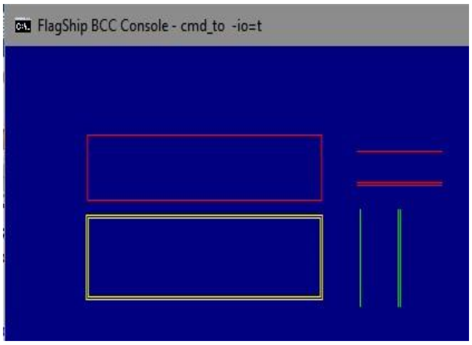
```
_aGlobSetting[GSET_G_N_DRAWLINE] := 1  // pixel, default
_aGlobSetting[GSET_G_N_DRAWBOLD] := 3  // pixel, default
```

Example:

```
SET COLOR TO "W+/B"
SET GUITRANSL LINES ON                // enable lines drawing in GUI
SET GUITRANSL BOX ON                  // enable box drawing in GUI
SET GUICOLOR ON                       // enable COLOR for GUI
CLS
@ 5,10  TO 9,40  COLOR "R+/B"            // box single line
@ 10,10 TO 15,40 DOUBLE COLOR "GR+/B"    // box double lines
@ 10,45 TO 15,45 COLOR "G+/B"            // vertical line
@ 10,50 TO 15,50 DOUBLE COLOR "G+/B"     // vertical line
@ 6,45  TO 6,55  COLOR "R+/B"            // horizontal line
@ 8,45  TO 8,55  DOUBLE COLOR "R+/B"     // horizontal line
```

Output:





Classification:

screen oriented output, buffered via DISPBEGIN()..DISPEND() in terminal i/o mode

Compatibility:

The COLOR option is new in FS4. The physical output on the screen in Terminal i/o mode depends on the terminal emulation chosen (environment variable TERM), the ability of the terminal to display the required graphical characters, and the via FSchrmap.def output mapping applied. See also LNG.5.1.4, section SYS, FS_SET("outmap").

Translation:

DISPBOX (expN1, expN2, expN3, expN4, 1|2 [, expC5])

Related:

@...BOX, @...DRAW, @...CLEAR, SCROLL(), SET COLOR, FS_SET("outmap")

ACCEPT ... TO

Syntax:

`ACCEPT [<exp>] TO <memvar>`

Purpose:

Waits for a string to be typed in from the keyboard. The result is placed in a memory variable.

Arguments:

<**memvar**> is the memory variable where the input is stored. If the variable is not declared or is not visible, a new autoPRIVATE is created.

Options:

Prompt: <**exp**> is the prompt which is displayed in front of the entry area. It can be an expression of any data type. If not given, no prompt is displayed.

Description:

ACCEPT is a waiting console command. First, a NEW LINE and the prompt (or "") is displayed. The characters typed in from the keyboard are stored in the specified memory variable. Keyboard entry is terminated by the ENTER key. If nothing was typed in, the variable contains a null string "".

BACKSPACE is the only special key supported.

Example:

```
ACCEPT "Enter your name: " TO name
```

Classification:

sequential screen output, waiting keyboard input

Translation:

<memvar> := __ACCEPT (exp)

Related:

@...SAY...GET, INPUT, WAIT, KEYBOARD, INKEY()

ACCESS METHOD
ASSIGN METHOD

Syntax:

```
ACCESS [METHOD] <methName> [( )]
        CLASS <className> [AS <type>]
```

Syntax:

```
ASSIGN [METHOD] <methName> (<par>)
        CLASS <className> [AS <type>]
```

See detailed description in the METHOD command.

ANNOUNCE

Syntax:

`ANNOUNCE <module>`

Purpose:

Declares a module identifier for the linker.

Arguments:

<**module**> is the identifier name, given as literal. The name may be of any length and is case-insensitive. Only the first 10 characters are significant. The name may not start with an underscore. The <module> name must be unique for the whole application.

Description:

The ANNOUNCE declaration statement specifies a module identifier (reference name or tag) to satisfy the REQUEST declaration (external request) from other .prg files during the link phase.

ANNOUNCE is generally used together with the **-na** compilation switch to generate a linker definition point, when the UDFs or UDPs declared there are exclusively referred to by macro or are declared as INIT/EXIT PROCEDUREs only.

Note: If the -na compiler switch is omitted, the FlagShip compiler produces an automatic procedure <file>, which will also satisfy an EXTERN <file> or REQUEST <file> declaration given elsewhere. See also the PROCEDURE command.

Only one ANNOUNCE declaration for a .prg file is allowed. All subsequent ANNOUNCE declarations produce a compiler warning and will be ignored.

The ANNOUNCE statement is nearly same as a declaration of

```
PROCEDURE <module>
RETURN .T.
```

Example 1:

```
*** file test.prg, compiled with: FlagShip test*.prg  ***
REQUEST test1
// or: EXTERNAL test1
// or: EXTERNAL test2
var := "test2"
DO &var
QUIT
*** file test1.prg ***
*** PROCEDURE test1                    // automatic procedure
*** RETURN .T.                         // generated by FlagShip
PROCEDURE test2
? "being now in test2"
RETURN
```

Example 2:

```
*** file test.prg, compiled with: FlagShip test*.prg -na
REQUEST test1
// or: EXTERNAL test2
var := "test2"
DO &var
QUIT

*** file test1.prg ***
PROCEDURE test2
? "being now in test2"
RETURN
```

Example 3:

```
*** file test.prg, compiled with: FlagShip test*.prg -na
REQUEST test5
// or: EXTERNAL test2
var := "test2"
DO &var
QUIT

*** file test5.prg ***
ANNOUNCE test5                         // new module name
PROCEDURE test2
? "being now in test2"
RETURN
EXIT PROCEDURE endproc                 // called by FlagShip only
? "bye, bye"
RETURN
```

Compatibility:

Available in FS4 and C5.2 only.

Classification:

compiler/linker

Related:

REQUEST, EXTERNAL

APPEND BLANK

Syntax:

`APPEND BLANK`

Purpose:

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

Description:

After APPENDing, the new blank record becomes the current record. The new field values are initialized to the empty values for each data type: character fields are filled with spaces; numeric fields are zero; logical fields are assigned false (.F.); date fields are assigned CTOD(""); and memo fields are left empty.

Multiuser:

In shared mode, APPEND BLANK automatically locks the new record. The lock remains active until UNLOCK or another lock (or APPEND BLANK) is executed in the corresponding working area. This automatic lock does not release an FLOCK() setting.

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 concurrently in different working areas), see chapter LNG.4.8.7.

Example:

Appending a new record in multiuser/multitasking mode:

```
SET EXCLUSIVE OFF                            // SET.. is not necessary,
USE employee                                 // if USE...SHARED is used
? RECCOUNT()                                 && 100
APPEND BLANK
WHILE NETERR()
   ? "waiting for successful append..."
   INKEY (1)                                 && delay 1 second
   APPEND BLANK                              && try again
ENDDO
? LASTREC(), RECNO()                         && 101 101
FOR i = 1 TO FCOUNT()
   fldnam = FIELD(i)
   ? EMPTY(&fldnam)                          && .T.
NEXT                                         && All fields are empty
REPLACE name WITH "Smith"
UNLOCK
```

Classification:

database

Translation:

DBAPPEND ()

Related:

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

APPEND ... FROM

Syntax:

```
APPEND FROM <file>|(<expC1>)
        [<scope>]
        [FIELDS <fieldList>]
        [FOR <condition>]
        [WHILE <condition>]
        [SDF | DELIMITED [WITH
           BLANK|<delimiter>|(<expC2>)]]
        [VIA <expC3>]
```

Purpose:

Adds records to the current database file from an ASCII or CSV text file, or another database file.

Arguments:

FROM <**file**>|(<**expC1**>) is the name of the source file. If no extension is specified, it is assumed to be .dbf. When specifying the clause SDF or DELIMITED, the default is the .txt extension and the file is assumed to be an ASCII text.

Options:

FIELDS <fieldList> If given, data is APPENDed only to the specified fields. For SDF or DELIMITED, it determines the order of fields in the text file according to the currently open & selected database; otherwise the field order of the target database apply.

<**scope**> is the part of the source database file to APPEND FROM. The default scope is ALL source records. See other valid scope options in the general command description at begin of this section.

<**condition**> specifies additional FOR and/or WHILE filtering of the records to be appended within the given <scope>. See the general command description at begin of the CMD section.

SDF identifies a System Data Format ASCII file. Each record is of a fixed length and ends with a line feed (LF) or CR/LF. Data are read until end-of-file or up to the DOS mark Ctrl-Z (1A hex), when scope is not given.

SDF: file format	
Field separator	None
Record separator	LF or CR/LF = 0A hex or 0D+0A hex
End of file marker	file-end or the DOS eof = 1A hex
Character fields	Padded with trailing blanks
Numeric fields	Padded with leading blanks or zeros
Date fields	YYYYMMDD or MM/DD/[YY]YY or DD.MM.[YY]YY
Logical fields	'T' is true, anything else is false
Memo fields	Ignored

DELIMITED identifies an ASCII text file, where fields are separated by commas and character fields are bounded by double quotation marks, which are the default delimiters. Note that character fields not bounded by delimiters will be appended correctly, if commas are not part of the field contents. Fields and records are variable length and end with a line feed or CR/LF. When scope is not given, data are read until end of the text file, or up to the DOS mark Ctrl-Z (1A hex).

DELIMITED [WITH delimiter]: file format	
Field separator	Comma (,) or specified in <delimiter>
Record separator	LF or CR/LF = 0A hex or 0D+0A hex
End of file marker	file-end or the DOS eof = 1A hex
Character fields	May be delimited by quotas ("...") or the <delimiter>, trailing blanks or TABs may be truncated
Numeric fields	Leading blanks and zeros may be truncated
Date fields	YYYYMMDD or MM/DD/[YY]YY or DD.MM.[YY]YY
Logical fields	'T' is true, anything else is false
Memo fields	Ignored

The date field is checked for YYYYMMDD like STOD(), and on failure by DD.MM.[YY]YY or MM/DD/[YY]YY etc. like CTOD() according to current SET DATE setting.

DELIMITED WITH <**delimiter**>|(<**expC2**>) identifies a delimited ASCII text file, where character fields are delimited with the specified delimiter, and the fields are separated by comma or given separator. Note: this clause, if given, must be the last one in the command. To avoid misinterpretation, it is better to enclose the delimiter in quotas. DELIMITED WITH `'"'` is the same as DELIMITED only clause, and assumes fields either w/o any delimiter, or enclosed in "..." quotas. The <delimiter> string may contain 0, 1, 2 or 3 characters:

0 char:	equivalent to DELIMITED WITH `'"'` or WITH `'",'` or WITH `'"",'`
1 char:	left + right field delimiter, field separator is comma (,)
2 char:	1st=left + right field delimiter, 2nd=field separator
3 char:	1st=left, 2nd=right field delimiter, 3rd=field separator

Field delimiters are considered only when the first character after field separator is the left delimiter and the last character before next separator is the given right delimiter. Delimiters are mostly used to store field separators (like commas) within the field, or to avoid skipping leading/trailing spaces and TABs of the character field.

For CSV text files (like export from Excel etc.), the common clause is DELIMITED WITH `'"";'` or DELIMITED WITH (`'""'` + chr(9))

DELIMITED WITH BLANK identifies an ASCII text file, where fields are separated by one space and character fields are not bounded by delimiters. It is equivalent to DELIMITED WITH (space(3)) clause.

DELIMITED WITH BLANK: file format	
Field separator	Single blank space or TAB = 20 or 09 hex
Record separator	LF or CR/LF = 0A hex or 0D+0A hex
End of file marker	file-end or the DOS eof = 1A hex
Character fields	Not delimited, trailing blanks may be truncated, no leading blanks or TABs are allowed.
Numeric fields	Leading zeros may be truncated
Date fields	YYYYMMDD or MM/DD/[YY]YY or DD.MM.[YY]YY
Logical fields	'T' is true, anything else is false
Memo fields	Ignored

VIA <**expC3**> specifies the name of the RDD (replaceable database driver) to be used to import the desired data, given as a quoted string or character variable. The default driver is "DBFIDX".

Description:

If the source file is a database, only fields with the same name are appended. Fields of different type or size are automatically converted by FlagShip.

Deleted records in the source database are also appended, but not marked as deleted in the target file. If SET DELETED is ON, deleted records from the source file are not appended.

Appending from an ASCII file: if the FIELD clause is not specified, the fields are assumed to be in the order of the target file. Specifying the FIELD clause also declares the order of the fields in the source file.

If the ASCII record is too short, only the available fields will be accepted. An empty source line will be appended as an empty record in the target database.

Note: It is **not** recommended to use TABs in ASCII files, especially not within the SDF file. FlagShip expands the TAB mark (Ctrl-I, 09hex) to **one** space only, because the TAB width is variable on Unix. When editing the ASCII file, use spaces instead of TABs and do not use the auto-indent option of your text editor.

Multiuser:

APPEND FROM automatically locks and unlocks the new record. FLOCK() or exclusive usage by the programmer may be specified but is not required. The source database will be opened in SHARE mode, the text source in read-only mode. If access is denied, a run-time error occurs.

Example:

Get the first 10 records from waitlist.dbf into the current database orders.dbf, also remove them in waitlist.

```
USE orders
? RECCOUNT()                          // 255
APPEND NEXT 10 FROM waitlist
USE waitlist NEW
DELETE NEXT 10
PACK
SELECT orders
```

```
? RECCOUNT()                                 // 265
```

Example:

```
APPEND FROM file1 SDF
APPEND FROM file2 SDF FIELDS name,address,earning
APPEND FROM file3.xyz FIELDS address, name, city, memo ;
       NEXT 50 FOR upper(SUBSTR(name,1)) >= "M" ;
       DELIMITED WITH ";"
```

Classification:

database and ASCII file import

Compatibility:

The automatic data conversion is new in FS4 and is not supported by Clipper. FlagShip supports both Unix/Windows and MS-DOS ASCII text files. Also the end-of-line mark LF or CR/LF and the end-of-file CR/LF/EOF are supported. Unicode (or UTF8) is not supported.

Source:

The text file for SDF or DELIMITED clause is read by ASCIRDD driver, available (and user-modifiable) in <FlagShip_dir>/system/ascirdd

Translation:

```
__DBAPP ("file", {"field1" [,"field2.."]}, ;
     {forCond}>, {whileCond}, [next], [record], [.rest.] )
__DBAPPSDF ("file", {"field1" [,"field2.."]}, ;
     {forCond}>, {whileCond}, [next], [record], [.rest.] )
__DBAPPDELIM ("file", "delim", {"field1" [,"field2.."]}, ;
     {forCond}>, {whileCond}, [next], [record], [.rest.] )
```

Related:

COPY, FREAD(), MEMOREAD(), oRdd:AppendDB, oRdd:AppendSDF(), oRdd:AppendDelimited()

AVERAGE ... TO

Syntax:

```
AVERAGE [<scope>] <expList> TO <memvarList>
        [FOR <condition>]
        [WHILE <condition>]
```

Purpose:

Averages a list of numeric expressions for a range of records in the current database file and puts the results in the memory variables specified.

Arguments:

<**expList**> is a list of numeric expressions to AVERAGE each processed record.

<**memvarList**> specifies the set of variables in which the results of averaging are to be put. This list must have the same number of elements as the <expList>. Existing variables with the same names are overwritten; non-existing ones are created as autoPRIVATE.

Options:

<**scope**> is the part of the current database file to be averaged. The default scope is ALL.

<**condition**> specifies additional FOR and/or WHILE filtering of the averaged records within the given <scope>. See the general command description.

Example:

Check the average age of the male employees. Note the usage of the WHILE clause for other purposes, here for counting:

```
LOCAL count := 0, male := 0
USE employee
year = YEAR(DATE())
AVERAGE year - YEAR(birthdate) TO aver_age ;
   FOR UPPER(sex) == "M" ;
   WHILE (count++, IF(UPPER(sex)=="M",++male,0), .T.)
? "The average age of", male, "men, i.e.", ;
  male * 100/count, "% of the staff is", aver_age, "years"
```

Classification:

database

Translation:

```
M->_Avg := var := 0
DBEVAL ({|| M->__Avg := M->__Avg + 1, var := var+ field }, ;
        {for}, {while}, [next], [rec], [.rest.] )
var := var / M->__Avg
```

Related:

SUM, TOTAL, oRdd:Average()

BEGIN SEQUENCE...END

Syntax:

```
BEGIN SEQUENCE
     <statements>
[BREAK [<exp>]]
     <statements>
[RECOVER [USING <var>]]
     <statements>
END|ENDSEQUENCE
```

Purpose:

A control structure to handle program exceptions.

Arguments:

BEGIN SEQUENCE defines the start of the control structure.

END[**SEQUENCE**] defines the end of the structure. Executing BREAK without RECOVER passes the control past this statement. On nested BEGIN..END, the higher control structure becomes active for the next BREAK.

Options:

BREAK: when encountered, terminates the sequence by branching the execution to the first statement following the corresponding RECOVER statement if one is specified, or the matching ENDSEQUENCE statement. If executed outside of the BEGIN..END structure, run- time errors occur.

BREAK <**exp**> passes the <exp>, which is usually an error object, to the <var> of the RECOVER USING clause.

RECOVER defines an entry point in the BEGIN..END sequence where control branches, following a BREAK statement.

RECOVER USING <**var**> receives the value returned by the BREAK <exp>. In general, <var> is an error object.

Description:

BEGIN SEQUENCE...END allows to BREAK from anywhere inside a sequence of statements, similar to a GOTO or JUMP <label> in other programming languages.

The BREAK statement can also be placed in nested procedures or functions, causing the same effect. FlagShip has no limitation in nesting BEGIN..END, just as with other control structures or UDFs.

A typical use of BEGIN SEQUENCE...END is to define a section of code where errors may occur, as a sequence structure. You can customize the error routines to allow BREAKing the sequence from within. After being out of the sequence, or within the RECOVER section, you can inform the user to check the access rights, turn the printer on-line or whatever happened.

By using the ISBEGSEQ() function, you may check elsewhere in the application, if the BREAK will reach a RECOVER or ENDSEQUENCE statement.

Example 1:

To exit nested control structures at once:

```
       BEGIN sequence
          break_yes = .T.
          if ...
             if ...
                do while .T.
                   if error
                      BREAK         --┐ Jump to end of structure
                   endif              |
                enddo                 |
                BREAK               --┤ Jump to end of structure
             endif                    |
          endif                       |
          break_yes = .F.             |
       END                          <─┘
       IF break_yes
          ? "Error break... "
          QUIT, LOOP, RETURN etc.
       ENDIF
```

Example 2:

Exit above structures using RECOVER:

```
       BEGIN sequence
          if ...
             if ...
                   BREAK            --┐ Jump to the recover section
             end                      |
             BREAK                  --┤ Jump to the recover section
          endif                 --┐   |
       RECOVER                    | <─┘
          ? "Error handling"      |
       END                      <─┘ Continue std. execution
       ? "Continuing..."
```

Example 3:

Exit nested procedures

```
      BEGIN sequence
        DO first
        ? "all o.k."     <─--┐
      END                <─┐ │
      ? "watch for ok"     │ │
                           │ │
      PROCEDURE first      │ │
      if error             │ │
         BREAK           --┤ │
      endif                │ │
      DO nextproc          │ │
      return             --│-┘ normal return
                           │
      PROCEDURE nextproc   │
      if error             │
         BREAK           --┘ Error, jump to end of structure
      endif
```

Example 4:

```
LOCAL name, path := ""
DO WHILE .T.
   BEGIN SEQUENCE
      IF ! FILE("test.prg")
         BREAK "text file test.prg"
      ENDIF
      TYPE test.prg
      IF ! FILE("mydbf.dbf")
         BREAK "database mydbf.dbf"
      ENDIF
      USE mydbf
   RECOVER USING name
      IF empty(path)
         path = "/usr/data"
         SET PATH TO (path)
         LOOP                           // try another path
      ENDIF
      ? "cannot open " + name
      QUIT
   ENDSEQUENCE
   EXIT                                 // exit the loop
ENDDO
```

Example 5:

Check for correctly OPENing the database. One error handle will be activated. The error may be triggered in any UDF called from here:

```
LOCAL brEobj, saveEobj
LOCAL errHand := {|e| my_err(e) }
saveEobj := ERRORBLOCK (errHand)        // install error handler

DO WHILE .T.
   BEGIN SEQUENCE                       // start of sequence
```

```
       my_open_dbf ("address")
       ? ".dbf correctly opened"         // and jump to ENDSEQUENCE
    RECOVER USING brEobj                 // receives error object
       IF brEobj == NIL                  // BREAK only, message
          : (statements)                 // already printed
       ELSE                              // BREAK with object
          ? "Error :"
          IF !EMPTY(brEobj:CARGO)
             ?? brEobj:CARGO             // print user message
          ELSE
             ?? LTRIM(STR(brEobj:GENCODE)) + ;
               " on file access " + brEobj:FILENAME + ;
               ": " + brEobj:DESCRIPTION
          ENDIF
       ENDIF
       IF myErrObj:CANRETRY
LOOP
       ENDIF
       QUIT
    ENDSEQUENCE                          // end of sequence
    EXIT                                 // exit the loop
ENDDO
ERRORBLOCK (saveEobj)                    // reset ErrorObj
QUIT

FUNCTION my_open_dbf (filename)          /* may be included in
--------------------                        another .prg file  */
IF !FILE(filename + ".dbf")
   ? "Install/copy database " + ;
     filename + " first."
   if ISBEGSEQ()
      BREAK                              // BREAK w/o object
   else
      QUIT
   endif
ENDIF
USE (filename) SHARE                     // BREAK may cause
RETURN NIL                               // in my_err()

FUNCTION my_err (myErrObj)               /* executed by the
---------------                             error handler   */
IF myErrObj:OSCODE == 32                 // SHARE error
   myErrObj:CARGO := "Database " + ;
        myErrObj:FILENAME + ;
        " opened by other user"
   myErrObj:CANRETRY := .T.
ENDIF
BREAK myErrObj                           // pass it to RECOVER
RETURN NIL
```

Classification: programming
Compatibility: Unlike C5, FlagShip also supports the branching out of a FOR or WHILE loop
Related: ISBEGSEQ(), RETURN, ERRORBLOCK(), (OBJ) Error objects

CALL

Syntax:

`CALL <name> [WITH <paramList>]`

Purpose:

Execute an external or inline C function.

Arguments:

<**name**> is the true name of the external or inline C (void) function. The default "_bb_" prefix of FlagShip UDFs is not added to the function name by the compiler. Neither will upper/lower conversion be done or the name captured.

Options:

WITH <**paramList**>: up to 8 parameters passed by reference into the C function. The parameters are comma separated FlagShip variables, constants, fields, array elements or expressions. They are passed as pointers to regular C variable types:

"N" type as:	double	*doubleCvar	ptr to (IEEE) double
"D" type as:	long	*longCvar	ptr to julian days
"L" type as:	unsigned	char *chrCvar	ptr to T or F char
"C" type as:	unsigned	char *chrCvar	ptr to \0 termin.string
"S" type as:	WINDOW	*windCvar	ptr to window struct.

When attempting to pass a constant or expression, a pointer to the contents of the resulting temporary variable is passed. The WORD() function converts the FlagShip numeric value to an (int) value and passes it **by value** rather than through a pointer to the C function.

Description:

CALL executes an independent or an inline C (void) function. The parameters are placed on the stack using the C parameter passing convention. External C functions are compiled by cc or by FlagShip.

Warning: different number or types of arguments passed / parameters received, extending the string length and incorrect usage or manipulation of the variable pointer will inevitably result in an application crash; sometimes not in the C function itself, but later during the regular program execution.

It is safer to use the FlagShip Extend System to execute a C function because of parameter checking and passing.

Example:

```
#Cinline
void My_C_Function (aaCvar, bbCvar, ccCvar, ddCvar, iiCvar)
   double        *aaCvar;              /* ptr to FS var "N"  */
   unsigned char *bbCvar;              /* ptr to FS var "L"  */
   unsigned char *ccCvar;              /* ptr to FS var "C"  */
   long          *ddCvar;              /* ptr to FS var "D"  */
   int           iiCvar;               /* passed by WORD()   */
{
   int  my_integer;                     /* local integer      */
   char my_string[100];                 /* local string       */

   (*aaCvar)--;                         /* aa = aa - 1        */
   *bbCvar = 'T';                       /* bb = .T.           */
   *(ccCvar +3) = 'X';                  /* cc = "my Xtring"   */
   *ddCvar += 2;                        /* dd = date + 2      */
   typed_ee = sqrt(typed_ee);           /* TYPED FS vars      */
   my_integer = (int) (*aaCvar);
   my_integer *= 10;
   strncpy (my_string, ccCvar, 100); my_string[99] = 0;
   if (strlen(my_string) < 85)
      strcat (my_string, " ... added in C");
   printw ("\nmy_integer=%d iiCvar=%d my_string=%s",
            my_integer, iiCvar * 2, my_string);
}
#endCinline

STATIC_DOUBLE typed_ee := 9.1           // typed FS var

FUNCTION main ()
LOCAL   aa := 22, bb := .F.
PRIVATE cc := "my string"
STATIC  dd := DATE()
CALL My_C_Function WITH aa, bb, cc, dd, WORD(aa)
? ; ? aa, bb, cc, dd, typed_ee

Compile: $ FlagShip test.prg -na -Mmain
```

Classification:

programming

Compatibility:

Using inline C code or typed variables is not supported by Clipper. LOADing the object program as used in dBASE is not supported, because the required function must already be available when linking. The contents of the logical value pointer differs to Clipper: (char) instead of (short int) and is stored as 'F'/'T' instead of 0/1. The (int) value passed by WORD() is usually the same size as (long) on Unix/Windows.

Related:

WORD(), (LNG,EXT) Open C System, (PRE) #Cinline

CANCEL / QUIT

Syntax:

`CANCEL`

or:

`QUIT`

Purpose:

Terminates program execution, closes all open files, and returns control to Unix/Windows.

Description:

CANCEL or QUIT may be used from anywhere in a program to end the program and return to the operating system. The same result is achieved if the RETURN command is used at the top level (main module). Pressing the break key ^K twice also terminates program execution, if the break key was not disabled with SETCANCEL().

Example:

```
IF LASTKEY() = 27                           // ESC key pressed?
   WAIT "Terminate (y/n) ? " TO answer      // confirm,
   IF UPPER(answer) == "Y"                  // accept Y,y
      QUIT                                  // terminate
   ENDIF
ENDIF
```

Classification:

programming (and database)

Translation:

__QUIT ()

Related:

QUIT, RETURN, FS_SET ("break"), SETCANCEL(), oRdd:Close()

CLASS, INSTANCE

Syntax 1:

```
     [STATIC] CLASS <ClassName>
              [INHERIT <SuperClass>]
              [ALIAS <AliasName>]
```

and optional:

```
     INSTANCE <Name> [:= <exp>] [AS <type>]
     EXPORT [INSTANCE] <Name> ...
     HIDDEN [INSTANCE] <Name> ...
     PROTECT [INSTANCE] <Name> ...
```

Syntax 2:

```
     PROTOTYPE [STATIC] CLASS <ClassName>
              [INHERIT <SuperClass>]
```

and optional:

```
     INSTANCE <Name> [AS <type>]
     EXPORT|HIDDEN|PROTECT [INSTANCE] <Name> [AS <type>]
```

Purpose:

Syntax 1 declares a class name and optional its instances to the compiler. Syntax 2 specifies an already elsewhere declared class to the compiler, without declaring the class again.

Arguments:

STATIC restricts the visibility of the class and its entities to the file in which it is declared. If omitted, the class has application-wide visibility. If a same named global class is already available, it will be hidden by the STATIC CLASS. But you cannot use a globally defined class and same named STATIC class within the same file. This may cause unpredictable results.

PROTOTYPE informs the compiler about the CLASS structure (and it's instances) which is defined elsewhere later in the application. The class entities becomes visible and their usage will be optimized, although the CLASS declaration was not (yet) specified in the same source file. PROTOTYPEing is required, when a class module (or access, assign) declaration is placed before (or in another file than) the class declaration itself. Refer also to chapter LNG.2.11.1 and the PROTOTYPE statement.

All instance names of the PROTOTYPEd class have to be declared (in any order, but with the same names) for proper compile-time addressing. Note: the FlagShip compiler automatically creates the prototyping file named "reposit.fh" (or a file of your choice) for you, see chapters FSC.1.4.2 and LNG.2.11.5. For the FlagShip standard classes, the prototypes are specified in the "stdclass.fh" file, which may be #include'd in your .prg source or in the local copy of the "std.fh" file.

CLASS <**ClassName**> is the class identifier, in the variable naming convention (10 significant characters). The only restriction is, that the resulting string of LEFT("ClassName"+"NEW",10) may not conflict with the same named function (std or UDF), or another class, since this name becomes the creator function.

INHERIT <**SuperClass**> automatically defines all instances (except hidden instances) and methods (including access and assign methods) from the <SuperClass> for the current <ClassName>. Hidden instances and all the methods may be overloaded by a local entity. Since the structure of the parent class <SuperClass> must be known for the compiler, use PROTOTYPE CLASS and PROTOTYPE METHOD. Note that standard FlagShip naming convention (abbreviation to 10 signif. characters) is valid also for the <ClassName> and it is therefore recommendable to use INHERIT myLongClas instead of myLongClassName.

ALIAS <**AliasName>** defines an descriptor which remain unchanged for the current and all inherited sub-classes. It allows you to specify a global name for a group of single classes, and later check an object by using IsObjProperty(obj,6,"aliasName"). For example, the FlagShip's standard classes are usually splitted to a basic class and specialized sub-classes for the corresponding i/o mode. But regardless how the class is instantiated, the object variable can be tested by using it global alias name. An alternative global descriptor used in many standard classes is the ClassName() method.

Options:

INSTANCE <**Name**> declares instance variables that are visible only in methods of the class being defined, and its INHERITed subclasses (except hidden).

EXPORT | HIDDEN | PROTECT [INSTANCE] <**Name**> additionally protects the instances and specifies its visibility and accessibility, see the description below.

AS <**type**> optionally specifies the data type associated with the instance variable. The valid <type>s are all usual and object types according to LOCAL..AS, with exception of the C-like types. If omitted, the instance variables will be polymorphic (an usual, untyped variable) and will have an initial value of NIL.

If the initializer (**:=** <**exp**>) is not given, the instance variable will be initialized to NIL or the empty <type>, resp. The assignment is ignored, if specified within the PROTOTYPEd class.

Description:

After the CLASS declarator statement, any number of optional INSTANCE declarations may follow. The instances are entities of the class, and have to be specified altogether in the same source file. The class declaration ends with any executable statement, another declarator (e.g. ACCESS, ASSIGN, MODULE, FUNCTION, CLASS etc.) or by the end-of-file.

You cannot declare the same class twice in the application, except for the STATIC class, which is then local to the UDF or source file only, similar to a STATIC variable or function.

Prototyping:

When the CLASS <name> is declared elsewhere, you can inform the compiler about its structure, to enable the compile-time optimization (see also chapter LNG.2.11.6). Otherwise, if the class structure, and/or the type of the object variable is unknown to the compiler (when encountering an object entity access), the slower run-time addressing is generated; this applies for **using** the class in the application only.

During the class property **declaration** (i.e. when creating the access, assign and method body), the whole CLASS structure must be known for the compiler. Therefore, if the CLASS statement is **not** specified in the same file, you have to PROTOTYPE the CLASS and all of its METHODs (see example 1 below). The same applies, when you declare a new, inherited CLASS. When the CLASS is specified in the same file, you have to use prototyping for forward declarations. As a rule of thumb: it is always a good programming style to prototype all the used class entities (i.e. all instances and methods). You will so avoid confusions when modifying the application later.

When using (the preferred) include file, it is a common mistake to declare "CLASS xyz" instead of "PROTOTYPE CLASS xyz", resulting in compiler or linker error once #including this header file in different sources. Since the "CLASS xyz" is a declaration (similar to the FUNCTION or PROCEDURE declarator), it hence can be declared only once per application; best in a .prg file.

Instances:

The difference between the INSTANCE and EXPORT, PROTECT and HIDDEN instances is:

- **EXPORT** does not protect the instance at all, but makes it visible (accessible and assignable) both for the application, and the class methods. The name cannot be overloaded by same named ACCESS and/or ASSIGN method.
- **INSTANCE** is hidden for the application, but visible in the class methods. If the same named ACCESS and/or ASSIGN method exists, such is invoked instead of the instance itself (except within the same named Access/ Assign body).
- **PROTECT** is very similar to a usual INSTANCE, except that the instance itself is always invoked in the class method, even if same named ACCESS and/or ASSIGN method exists.
- **HIDDEN** is very similar to the PROTECT instance, except that this instance is not overtaken into inheriting subclasses.

The following table summarizes the instance properties:

Inst.type	applic	method	inherit	acc/ass	acc/ass pref
EXPORT	yes	yes	yes	no	no
INSTANCE	no	yes	yes	yes	yes
PROTECT	no	yes	yes	yes	no
HIDDEN	no	yes	no	yes	no

where: (applic) is the visibility of the instance to the application; (method) is the visibility of the instance to the class methods; (inherit) whether the instance is inherited into a subclass; (acc/ass) if an access or assign method of the same name may be specified; (acc/ass pref) whether the access or assign method of the same name is preferred in the ACCESS, ASSIGN and METHOD body.

Note: Compound send operators are also supported, e.g.

```
self:fillname(cParam):coAutName := cVar
```

which is resolved from left to right, as

```
oTemp := self:fillname(cParam) ; oTemp:coAutName := cVar
```

Since the temporarily created object variable (here named as oTemp) is late evaluated (when the method is not explicitly typed as class name), it requires to un-hide the instance "coAutName" (by EXPORT or ACCESS). Therefore, the equivalent program notation

```
self:fillname(cParam) ; coAutName := cVar
```

is usually better, since it may be early evaluated and can use any instance type.

Example 1:

Defines two classes and theirs entities in two different files. The prototyping in file2 is required for the inheritance and used also (together with typing the object variables) for the compile- time resolution of the object entity addresses (see LNG.11.6).

```
*** file1.prg *******************
CLASS authors
   INSTANCE name  := ""
   INSTANCE first := "" AS character
   PROTECT  title
   EXPORT   issue AS date
* PROTOTYPE ACCESS name             CLASS authors // note 1
* PROTOTYPE ASSIGN name(cValue)     CLASS authors // note 1
* PROTOTYPE METHOD init(cName)      CLASS authors // note 1
* PROTOTYPE METHOD fillname(cInput) CLASS authors // note 1

ACCESS name CLASS authors
   return name

ASSIGN name(cValue) CLASS authors
   if valtype(cValue) == "C" .and. !empty(cValue)
      name := cValue
   endif
   return name

METHOD init(cName) CLASS authors
   name := if(valtype(cName) == "C", cName, "")
   issue:= date()
   return self
*** eof file1 ***
*** file2.prg *******************
PROTOTYPE CLASS authors                          // note 3
   INSTANCE name
   INSTANCE first AS character
   PROTECT  title
   EXPORT   issue AS date
PROTOTYPE ACCESS name             CLASS authors  // note 2
PROTOTYPE ASSIGN name(cValue)     CLASS authors  // note 2
PROTOTYPE METHOD init(cName)      CLASS authors  // note 2
PROTOTYPE METHOD fillname(cInput) CLASS authors  // note 2

CLASS coAuthors INHERIT authors                  // note 3
   HIDDEN   coAutName AS character
* PROTOTYPE METHOD fillSubAuth() CLASS coAuthors // note 1

METHOD fillname(cInput) CLASS authors            // note 2
   cInput := trim(strtran(cInput, ",", " "))
   if " " $ cInput
      name := left(cInput, at(" ", cInput)-1)
      first:= substr(cInput,at(" ", cInput)+1)
   else
      name := cInput
      first:= ""
   endif
   return name
METHOD fillSubAuth(cAuth, cSubAuth) CLASS coAuthors
   self:fillname(cAuth) ; coAutName := cSubAuth
   return NIL

FUNCTION start()                      // program start
   LOCAL oAuth AS authors             // typed Locals for
   LOCAL oSub1, oSub2 AS coAuthors    //   speed-up only
   oAuth := AUTHORS {"Miller"}        // instantiate oAuth
   oSub1 := COAUTHORS {}              // instantiate oSub1
   oSub1:fillSubAuth("Smith", "Maier")
   * oSub2 := COAUTHORS {}    // instantiate new obj, or:
   * oSub2 := oSub1           // oSub2 points to oSub1 obj
   oSub2:fillName("Johnson")  // otherwise, RTE occurs here
   ? oSub1:name, oSub2:name, oSub1:issue
   ? oAuth:name, oAuth:issue
   quit
*** eof file2 ***
*** Compile: FlagShip file?.prg -na -m -Mstart
```

Note 1: it is a good programming style to prototype all the used class entities, even if declared later. Therefore, un-comment it.

Note 2: you have to PROTOTYPE the whole class and all its properties, to be able add/define its entities in another source file.

Note 3: the whole "parent" class must be known when inheriting it.

Example 2:

The same example, but the class declaration and its methods are specified in the same file. In the second (user) file, run-time evaluation takes place, since the class structure and/or object type is unknown at compile time. You may avoid it, and speed-up the execution by prototyping the CLASSes in file2.prg, e.g. by #include-ing the "reposit.fh" file.

```
*** file1.prg ***
CLASS authors
   INSTANCE name  := ""
   INSTANCE ...
```

```
ACCESS name CLASS authors
   return name
ASSIGN name(cValue) CLASS authors
   if ...
   return .T.
METHOD init(cName) CLASS authors
   name := ...
   return self
METHOD fillname(cInput) CLASS authors
   cInput := ...
   return NIL

CLASS coauthors INHERIT authors
   HIDDEN   coAutName AS character

METHOD fillSubAuth(cAuth, cSubAuth) CLASS coAuthors
   self:fillname(cAuth)coAutName := cSubAuth
   return NIL
*** eof file1 ***
*** file2.prg ***
FUNCTION start()                // program start
   LOCAL oAuth AS AUTHORS
   LOCAL oSub1, oSub2
   oAuth := AUTHORS {"Miller"}
   oSub1 := COAUTHORS {}
   ...
   quit
*** eof file2 ***
```

Example 3:

The same application, but the class is declared in an already compiled file 'file1.o', available for the user as a black box (or in an object library). The application (file2.prg) knows the class structure and uses prototyping for the compile-time address resolution.

```
*** file2.prg ***
#include "file1.fh"             // includes prototypes
FUNCTION start()                // program start
   LOCAL oAuth, oSub1, oSub2
   oAuth := AUTHORS {"Miller"}
   oSub1 := COAUTHORS {}
   ...
*** eof file2 ***
*** file1.fh *** (created e.g. from reposit.fh)
PROTOTYPE CLASS authors
   EXPORT   issue AS date
   INSTANCE name := ""        // assignment is ignored
   PROTECT  title
   INSTANCE first AS character
PROTOTYPE ACCESS name            CLASS authors
PROTOTYPE ASSIGN name(cValue)    CLASS authors
PROTOTYPE METHOD init(cName)     CLASS authors
PROTOTYPE METHOD fillname(cInput) CLASS authors
PROTOTYPE CLASS coauthors INHERIT authors
   HIDDEN   coAutName AS character
PROTOTYPE METHOD fillSubAuth()    CLASS coAuthors
```

```
 *** eof file1.fh ***
 *** Compile: FlagShip file2.prg file1.o -na -Mstart
```

Example 4:

For additional examples, see chapter LNG.2.11 and the METHOD declarator.

Classification:

programming

Compatibility:

Not available in Clipper, but compatible to CA/VO. The PROTOTYPE and ALIAS clause is available in FlagShip only.

Related:

[ACCESS, ASSIGN] METHOD, PROTOTYPE, LOCAL..AS, (OBJ)DBSERVER, LNG.2.11

CLEAR

Syntax:

```
CLEAR
```

Purpose:

Clears the screen and all active GET fields, homes the cursor.

Description:

CLEAR is a full-screen command that erases the screen using the current color setting and releases pending GET objects in the currently visible GETLIST array. When the screen is cleared, the cursor is set to the upper left corner (0,0).

When this command is used in a VALID or SET KEY routine while being in READ, the active READ aborts on returning from the UDF.

If you want only to clear the screen without releasing the GETs, use CLEAR SCREEN, CLS or @..CLEAR commands instead.

In GUI mode, the widgets are erased by CLS (or by @..CLEAR TO..) as well, see also LNG.5.3

Example:

```
CLEAR
USE authors
LIST Firstname, Lastname
```

Classification:

screen oriented output

Translation:

```
SCROLL() ; SETPOS(0,0)
__KILLREAD() ; GetList := {}
```

Related:

@...CLEAR, @...GET, CLEAR GETS, CLEAR SCREEN, CLS, SCROLL()

CLEAR ALL

Syntax:

```
CLEAR ALL
```

Purpose:

Closes all open databases, indices, format files, releases all PUBLIC and PRIVATE memory variables, clears all GETs and selects working area 1.

Description:

CLEAR ALL does not release LOCAL, STATIC or typed variables. This command is a superset of CLOSE DATABASES, CLOSE FORMAT, CLEAR MEMORY, CLEAR GETS and SET ALTERNATE TO.

Files associated with working areas can be explicitly closed with one of the various forms of the CLOSE command. Private and public variables can be released using the RELEASE command, although explicitly releasing variables is not generally recommended. For more information on the scope and lifetime of variables, refer to the LNG section.

Classification:

database, programming

Translation:

```
CLOSE DATABASES      =>   DBCLOSEALL()
CLOSE FORMAT         =>  __SET FORMAT TO; __SETFORMAT({|| })
CLEAR MEMORY         =>  __MCLEAR()
CLEAR GETS           =>  __KILLREAD() ; GetList := {}
SET ALTERNATE OFF    =>  SET (_SET_ALTERNATE, OFF)
SET ALTERNATE TO     =>  SET (_SET_ALTFILE, "" )
```

Related:

CLEAR MEMORY, CLEAR GETS, CLOSE, RELEASE, SET(), oRdd:Close()

CLEAR GETS

Syntax:

`CLEAR GETS`

Purpose:

Clears the active set of GETs.

Description:

This command explicitly releases all GET objects in the current and visible GETLIST array and terminates the current READ if executed within a UDF of the VALID clause or if invoked by a SET KEY procedure.

There are three other mechanisms that automatically release GET objects: the CLEAR command, and READ specified without the SAVE clause, and invoking the ReadKill(.T.) function. The last will not delete the GetList[] entries when executing ReadSave(.T.), so you may re-issue READ without filling the GETs anew.

Note that CLEAR GETS does not clear the GET fields on the screen, but objects in the Getlist buffer. To clear the display, use CLS or CLEAR SCREEN or CLEAR or @ row,col CLEAR TO row,col or READ CLEAR. This is also required, when you wish to overwrite inactive GETs by @..SAY in GUI mode.

Example 1:

```
LOCAL var1 := 10.5, var2 := "text text "
SET COLOR TO "W+/B, GR+/BG"
@ 1,2 GET var1                          // display with
@ 2,2 GET var2                          // "GR+/BG" color
? "Getlist{} length=", len(Getlist)     // 2
CLEAR GETS                              // GETs yet visible
? "Getlist{} length=", len(Getlist)     // 0
wait
@ 1,2 CLEAR TO 2,maxcol()               // un-display these GETs
```

Example 2: Emulate @..GET display

```
LOCAL var1 := 10.5, var2 := "text text "
SET COLOR TO "W+/B, GR+/BG"
SETENHANCED
@ 1,2 SAY var1                          // display with
@ 2,2 SAY var2                          // "GR+/BG" color
SETSTANDARD
```

Classification:

programming

Translation:

ReadKill(.T.) ; getlist := {}

Related:

@...CLEAR, @...GET, CLEAR, READ, ReadKill(), ENHANCED, STANDARD

CLEAR MEMORY

Syntax:

CLEAR MEMORY

Purpose:

Clears all PUBLIC and PRIVATE memory variables.

Description:

CLEAR MEMORY deletes all public and private variables from the internal memory variable table, unlike RELEASE ALL, which assigns NIL to PRIVATEs of the procedure where issued.

In FlagShip, there is no real need to RELEASE or CLEAR variables since the number of variables is not limited. Returning from the UDP or UDF automatically releases all the PRIVATE and autoPRIVATE variables declared or created there.

LOCAL, STATIC and typed variables are not affected by CLEAR MEMORY or RELEASE. See also (LNG) Variable scope and Visibility.

Example:

```
LOCAL locvar := 1
PUBLIC pub1, pub2, pub3 := .F.
PRIVATE priv1 := 1234, priv2 := 5678
STORE "test" TO pub1, pub2
priv3 = 9876                                && autoPRIVATE

? TYPE("pub1"), TYPE("priv1"), ;
  TYPE("pub3"), TYPE("priv3")               && C N L N
RELEASE ALL
? TYPE("pub1"), TYPE("priv1"), ;
  TYPE("pub3"), TYPE("priv3")               && C U L U
CLEAR MEMORY
? TYPE("pub1"), TYPE("priv1"), ;
  TYPE("pub3"), TYPE("priv3")               && U U U U
? VALTYPE(locvar)                           && N
```

Classification:

programming

Translation:

__MCLEAR( )

Related:

CLEAR ALL, RELEASE, RELEASE ALL

CLEAR MENU

Syntax:

`CLEAR MENU`

Purpose:

Clears all @..PROMPT items without user interaction.

Description:

CLEAR MENU deletes all menu items previously created by @..PROMPT and not yet processed by MENU TO. The next @..PROMPT command will then start a new menu item sequence.

This command does not clear the items/text displayed on the screen, but the menu items only. It is similar to executing MENU TO, but there is no user interaction with CLEAR MENU.

The Prompt class is used internally for @..PROMPT items and MENU TO processing, the object is hold in _oPrompt. See also menuclass.fh. The CLEAR MENU command is equivalent to _oPrompt:Clear()

Example:

```
LOCAL _oPrompt, lastCol

@ 1,0       PROMPT "Item 1"
@ 1,col()+3 PROMPT "Item 2"
@ 1,col()+3 PROMPT "Item 3"
lastCol := col()
...
if myConditionIsMet
   MENU TO myChoice          // perform user selection
else
   CLEAR MENU                // cancel it
   @ 1,0 CLEAR TO 1,lastCol  // and remove from screen
   myChoice := 0             // for later processing...
endif
```

Classification:

programming

Translation:

_oPrompt:Clear()

Related:

@...PROMPT, MENU TO

CLEAR SCREEN / CLS

Syntax:

`CLEAR SCREEN`

or:

`CLS`

Purpose:

Clears the screen and homes the cursor.

Description:

CLEAR SCREEN (or CLS) is a full-screen command that erases the screen using the current color setting. It is identical to the @ 0,0 CLEAR or @ 0,0 CLEAR TO MAXROW(),MAXCOL() command. When the screen is cleared, the cursor is set to upper left corner (0,0).

As opposed to CLEAR, the current GETs are not cleared/deleted by CLEAR SCREEN or CLS.

In Terminal i/o mode, the screen background corresponds to the standard color pair, set by SetColor() or SET COLOR TO command.

In GUI mode, the background color (assigned by SET COLOR) is set only when SET GUICOLOR is ON (default is OFF - according to GUI design specs). You may set the background also explicitly by invoking SetColorBackground(cColor) followed by CLS, CLEAR SCREEN, SCROLL() or @ ... CLEAR [TO..]

Example:

```
CLEAR SCREEN
USE authors
LIST Firstname, Lastname, Title
WAIT
CLS
```

Example:

```
SET GUICOLOR ON  // use colors also in GUI mode (default is OFF)
SET COLOR TO "W+/B,R+/GR,,,B/W"
CLS
? "hello world"  // white text on blue background
wait
```

Classification: *programming*

Compatibility:

Unlike DOS, the size of the screen and the color capability is not fixed in Unix, but depends on the terminal emulation chosen (environment variable TERM) and the terminal description in the terminfo file. Where possible, use one of the extended terminal descriptions FSxxx. See (REL) Predefined Terminals and LNG.2.1.

Translation: `SCROLL() ; SETPOS(0,0)`

Related: *@...CLEAR, @...CLEAR TO, CLEAR, COL(), ROW(), MAXCOL(), MAXROW(), SET GUICOLOR, SetColorBackgr()*

CLEAR TYPEAHEAD

Syntax:

`CLEAR TYPEAHEAD`

Purpose:

Clears the keyboard buffer.

Description:

CLEAR TYPEAHEAD is used to make sure that no keystrokes remain pending in the FlagShip buffer. This could happen if the user typed several keystrokes in advance, which were then stored in an internal type-ahead buffer, see LNG.5.2.1.

This command is often used prior to executing a @..GET/READ, @..PROMPT/MENU, DBEDIT(), ACHOICE() etc. or before setting up keyboard trapping using SET KEY TO, to avoid side effects from characters pending in the buffer.

Commands KEYBORD and SET TYPEAHEAD also clear the type-ahead buffer.

Note: some Clipper versions clears (undocumented-wise) the LastKey buffer by CLEAR TYPEAHEAD, some do not. If you wish to clear the LastKey buffer in FlagShip, use LastKey([pos],,.T.).

Example:

```
@...PROMPT...
? NEXTKEY()                             && 27
CLEAR TYPEAHEAD
? NEXTKEY()                             &&  0
MENU TO choice
```

Classification:

programming

Translation:

__KEYBOARD ()

Related:

KEYBOARD, SET TYPEAHEAD, NEXTKEY(), INKEY(). LASTKEY()

CLOSE

Syntax:

```
CLOSE [<Alias> | ALL | ALTERNATE | DATABASES |
           FORMAT | INDEXES]
```

Purpose:

Closes all files of the specified type.

Arguments:

CLOSE with no argument closes the current database file and its indices, producing the same effect as USE without an argument.

Options:

<**Alias**> does the same as CLOSE, but with a specified working area where the files given are closed rather than the default current working area.

ALL: Closes the database and index files in all working areas, as well as the format and alternate files, and releases active filters and relations.

ALTERNATE: Closes the currently open alternate file, with the same effect as SET ALTERNATE TO.

DATABASES: Closes database and index files in all working areas and releases active filters and relations.

FORMAT: Closes the active format file, with the same effect as SET FORMAT TO.

INDEXES: Closes all open indices in the current working area.

Description:

There are other commands besides CLOSE, which also close files. These are: QUIT/CANCEL, RETURN from the main procedure, CLEAR ALL and USE without an argument.

The "fatal error" runtime error or user termination via ^K also closes all files by using QUIT before exiting a program.

Multiuser:

If a record or the whole file is locked by RLOCK() or FLOCK(), all the locks are automatically removed when the database file is closed.

Example:

```
DO WHILE .T.
   CLOSE DATABASES
   choice = my_menu ()
   DO CASE
   CASE choice = 1
      CLOSE ALL
      USE personal
      DO pers_proc          // process task
   CASE choice = 2
      USE stock NEW
      DO stock_proc         // process task
   ENDCASE
ENDDO
QUIT
```

Classification:

programming, database

Compatibility:

The option <Alias> is new in FS4.

Translation:

```
CLOSE                =>   DBCLOSEAREA()
CLOSE ALIAS          =>   <alias>->(DBCLOSEAREA() )
CLOSE ALL            =>   CLOSE DATA ; SELE 1; CLOSE FORMAT
CLOSE ALTERNATE      =>   SET(_SET_ALTFILE, "")
CLOSE DATABASES      =>   DBCLOSEALL()
CLOSE FORMAT         =>   __SETFORMAT(NIL)
CLEAR INDEX          =>   DBCLEARINDEX()
```

Related:

CLEAR ALL, QUIT, RETURN, SET ALTERNATE TO, SET FORMAT TO, USE, SETCANCEL(), FS_SET("break"), oRdd:Close()

COMMIT

Syntax:

```
COMMIT [ALL]
```

Purpose:

Writes the internal Unix (or Windows) buffers of all used working areas to the hard disk.

Options:

COMMIT ALL will commit both SHARED and EXCLUSIVE open databases. If ALL is not given, only SHARED open databases are committed. You may set the global switch

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

whereby COMMIT behaves then same as COMMIT ALL The COMMIT ALL also commits text files (if any) for SET PRINTER, SET ALTERNATE and SET EXTRA to hard disk.

Description:

FlagShip stores the current .dbf record in internal working area buffers. These get flushed to the Unix (or Windows) buffer upon:

• SKIP 0 or DbSkip(0)	one SHARED
• SELECT or DbSelectArea(n)	one SHARED
• GOTO Recno() or DbGoto(Recno())	one SHARED
• CLOSE / USE	all SHARED/EXCL
• QUIT, user abort (Ctrl-K)	all SHARED/EXCL
• COMMIT or DbCommit() or DbCommitAll()	all SHARED
• COMMIT ALL or DbCommit(.T.) or DbCommitAll(.T.)	all SHARED/EXCL

When executing any of the above commands, the current (one) database or (all) database changes become visible to other users in a multi-user/multi-tasking environment.

COMMIT updates internal buffers for all working areas (i.e. dbf and if available also dbt, idx) writing them physically to the hard disk and reads them back again. The executable waits until the update is performed successfully.

See further tuning details in SET COMMIT

COMMIT is equivalent to DbCommitAll() function, COMMIT ALL is same as DbCommitAll(.T.). To commit only current database, use DbCommit() function.

Multiuser:

Use this command to make sure that the data is immediately physically written to the disc. The USE and CLOSE commands as well as above commands implicit the DBCOMMIT() which flushes the changes of the current database to the disc. Executing this COMMIT command or DbCommit*() functions will make the database and index changes available to other users. See also LNG.4.8.5.

You may also use COMMIT prior to outputting the database record (if not SKIPped before) to make sure the current data (which had been probably changed in the meantime by another user) will be read from the file and not from the internal buffer only.

COMMIT should be executed before you free the by Flock() or Rlock() locked records or database, especially on heavy loaded database. If SET AUTOLOCK is ON (the default), COMMIT is executed automatically in AutoUnlock(), see also *<FlagShip_dir>/system/autolock.prg*.

Performance:

The COMMIT may be **very** expensive (time consuming), especially in Windows and/or network environment. If you REPLACE or APPEND records within a loop, better is to COMMIT changes after finishing the loop instead of do it after each replacement within the loop. If only one database was changed, better is to use DbCommit() instead of COMMIT all of them.

Tuning:

See tuning details in SET COMMIT

Example 1:

```
USE stock SHARED                        // check by USED()
SET INDEX TO stockno                    // check by NETERR()
SEEK 12345                              // check by FOUND()
WHILE !FLOCK()                          // wait for file-lock
   SleepMs(50)                          //   with small delay to
ENDDO                                   //   avoid heavy CPU load
DO WHILE !EOF() .AND. stock_no = 12345
   REPLACE sold_out WITH .T., ;
           act_item WITH  0
   SKIP
ENDDO
COMMIT                                  // update all buffers
UNLOCK                                  // release file-lock
```

Example 2:

Display data, check the actuality every 5 seconds and redisplay new data, if changes detected:

```
LOCAL changed = .T., key, act_items
FIELD stock_no, text, item_avail
USE stock SHARED                        // check by USED()
SET INDEX TO stockno                    // check by NETERR()
SEEK 12345                              // check by FOUND()

WHILE .T.
   IF changed                           // avoid permanent display
      @ 1,0 say stock_no
      @ 2,0 say text
      @ 3,0 say item_avail
      act_items = item_avail
      changed = .F.
   ENDIF
   key = INKEY(5)                          // wait 5 sec or user key
   IF key # 0
EXIT
   ENDIF
   COMMIT                                  // or: SKIP 0
   IF act_items # item_avail               // data changed,
      ?? chr(7)                            // sound bell
      changed = .T.
   ENDIF
ENDDO
IF key = 27 ...                            // process pressed key
```

Classification:

database

Compatibility:

The following commands produce the same effect as COMMIT: SKIP 0 or act_rec := RECNO() ; GOTO act_rec

Translation:

DBCOMMITALL() or DBCOMMITALL(.T.)

Related:

SET COMMIT, GOTO, SKIP, REPLACE, UNLOCK, DBCOMMIT(), DBCOMMITALL(), oRdd:Commit()

CONTINUE

Syntax:

`CONTINUE`

Purpose:

Continues the pending LOCATE search in the current working area.

Description:

The search is continued from the current record. It terminates when the first record which meets the most recent LOCATE condition, is found, or, the end of LOCATE scope is reached.

If the search was successful, the matching record becomes the current record, and FOUND() returns .T. Else, FOUND() returns .F., and the record pointer is positioned on EOF or the next record outside the FOR scope.

Each working area may have an active LOCATE condition which remains pending until a new condition is issued or a new database file is used in that area. No other actions release the LOCATE condition.

The <scope> and WHILE conditions of the initial LOCATE are ignored; only the FOR condition is used with CONTINUE. If you are using a LOCATE with a WHILE condition and want to continue the search for a matching record, use SKIP and then repeat the original LOCATE statement adding REST as the scope.

Example:

```
USE employee
? RECCOUNT()                             && 100
LOCATE FOR Salary > 50000
? FOUND(), EOF(), RECNO()                && .T. .F. 21
CONTINUE
? FOUND(), EOF(), RECNO()                && .T. .F. 53
CONTINUE
? FOUND(), EOF(), RECNO()                && .F. .T. 101
```

Classification:

database

Translation:

__DBCONTINUE()

Related:

LOCATE, FOUND(), oRdd:Continue(), oRdd:GetLocate()

CONSTANT

Syntax:

```
CONSTANT <memvar> := <exp>
```

Purpose:

Creates and initializes the specified memory variable similar to PUBLIC but not re-assignable. If you wish to declare re-assignable public variable, use PROTECT PUBLIC instead.

Arguments:

<**memvar**> is the variable to be created as fix PUBLIC. The name may be of any length, but only the first 10 characters are significant (see more LNG.2.6). Variable names in the FlagShip language are not case sensitive.

Initializing:

<**exp**> is any valid FlagShip expression including a literal (constant) array to initialize the variable. Since the CONSTANT is not re-assignable, the initializer must be specified at the time of declaration.

Scope, Visibility:

CONSTANT variables have the same scope and visibility as the PUBLIC variables, i.e. are available (after the declaration) for the whole life-time of the application.

Classification:

programming

Compatibility:

New in FS5

Related:

PUBLIC, MEMVAR

COPY FILE ... TO

Syntax:

```
COPY FILE <file1>|(<expC1>)
     TO <file2>|(<expC2>)
        [ADDITIVE] [WITHMSG]
```

Purpose:

Duplicates a file regardless of its type.

Arguments:

<**file1**> is the name of the source file, including extension, according to Unix or Windows conventions; DOS filenames are also supported. Standard Unix/Windows wildcards are allowed.

<**file2**> is the name of the target file, including extension, according to Unix/Windows conventions. Standard Unix and Windows wildcards (or a path only, but not the period . alone) are allowed.

Option:

ADDITIVE appends the contents of <file2> to <file1>. <file2> must be explicitly specified. If a wildcard is given in <file1>, all files found are copied to <file2>. When the ADDITIVE option is omitted, <file2> is overwritten.

<**WITHMSG**> if specified, run-time warning message is displayed on failure. Default is no warning message.

Description:

COPY FILE copies files located in the SET DEFAULT TO path if set, or the current directory, unless a path is specified. The success or error may be checked using DOSERROR(). Both <file1> and <file2> (if such exists) must be closed before being copied. The file's permission of <file2> is set according to umask.

Example:

```
FS_SET ("lower", .T.)                       // automat. translation
COPY FILE TestFile.tmp TO test.dummy
? DOSERROR(), FILE("test.dummy")            // 0  .T.
COPY FILE [a-d]*x.p?g  /usr/smith/allfiles.prg ADDITIVE WITHMSG
TYPE /usr/smith/allfiles.prg
COPY FILE [a-d]*x.p?g  /usr/smith           // directory
```

Classification:

system, file access

Compatibility:

The COPY FILE command is equivalent to the Unix command "cp file1 file2" or "cat file1 >> file2" or Windows "copy file1 file2", if the ADDITIVE option is used. ADDITIVE and WITHMSG clauses, wildcard support, and DOSERROR() checking is available in FlagShip only.

Translation:

```
__COPYFILE ( "file1", "file2", .add., .msg.)
```

Related:

RENAME, COPY TO, SET DEFAULT, RUN, Unix:cp, Windows:Copy

COPY TO

Syntax:

```
COPY TO <file>|(<expC1>)
        [<scope>]
        [FIELDS <fieldList>]
        [FOR <condition>]
        [WHILE <condition>]
        [SDF | DELIMITED
                [WITH BLANK|<delimiter>|(<expC2>)]
           [FLDSEP (<expC4>)]
           [CHARSEP (<expC5>)]
           [LINESEP (<expC6>)]
           [HEADER (<expC7>)]
           [WITHMEMO [<expN8>]]  ]
        [VIA <expC3>]
```

Purpose:

Copies specified parts or the whole current database to a new file.

Arguments:

TO <**file**>|(<**expC1**>) is the name of the new file. If an extension is not specified, it is assumed to be .dbf when no type clause is given, or .txt otherwise. The <file> name may be omitted, with the SDF or DELIMITED clause, when an SET EXTRA file already opened is to be used ADDITIVEly. The given path or the SET DEFAULT is obeyed.

Options:

FIELDS: Specifies the list of fields to copy to the target file. The default is all fields. In text files the fields will appear in the order given by the FIELDS clause, if specified.

<**scope**> is the part of the current database file to COPY. The default scope is ALL.

<**condition**> specifies additional FOR and/or WHILE filtering of the copied records within the given <scope>. See the general command description.

SDF: Specifies the output data type to be a System Data Format ASCII file. Records are of a fixed length, separated by a line feed, without a field separator. Character fields are padded with trailing blanks, numeric fields are padded with leading blanks, date fields are written in the form "yyyymmdd", and logical fields are written in the form T/F.

SDF: file format	
Field separator	None or <expC4>
Record separator	LF or CR/LF = 0Ahex or <expC6>
End of file marker	file-end or the DOS eof = 1Ahex
Character fields	Delimited by <expC5>...<expC5>, padded with trailing blanks
Numeric fields	Padded with leading blanks for zeros
Date fields	YYYYMMDD
Logical fields	T or F

Memo fields	Ignored without WITHMEMO <expN8> clause

DELIMITED identifies an ASCII text file, where fields are separated by commas and character fields are bounded by double quotation marks, which are also the default delimiters. Fields and records are of variable length and end with a line feed. Leading and trailing spaces for numeric and character fields are truncated, date fields are written in the "yyyymmdd" form, and logical fields are written as T/F.

DELIMITED [WITH delimiter]:	file format
Field separator	Comma (,) or <expC4>
Record separator	LF or CR/LF = 0A hex or 0D+0A hex
End of file marker	None, file-end
Character fields	Delimited by quotas ("...") or by <expC2> or by <expC5>, trailing blanks truncated
Numeric fields	Leading zeros truncated
Date fields	YYYYMMDD
Logical fields	T or F
Memo fields	Ignored without WITHMEMO clause

DELIMITED WITH <**delimiter**>|(<**expC2**>) identifies a delimited ASCII text file, where character fields are delimited with the specified delimiter. Note: this clause, if given, must be the last one within the command. To avoid misinterpretation, it is better to enclose the delimiter in quotes. DELIMITED WITH `'"'` is the same as the clause DELIMITED only. This clause can be overwritten by FLDSEP clause.

DELIMITED WITH BLANK identifies an ASCII text file, where fields are separated by one space and character fields are not bounded by delimiters (except when FLDSEP and/or CHARSEP is specified).

DELIMITED WITH BLANK:	file format
Field separator	Single blank space or <expC4>
Record separator	LF or CR/LF = 0A hex or 0D+0A hex
End of file marker	None, file-end
Character fields	Not delimited (or delimited by <expC5>), trailing blanks are truncated
Numeric fields	Leading zeros truncated
Date fields	YYYYMMDD
Logical fields	T or F
Memo fields	Ignored without WITHMEMO clause

FLDSEP <**expC4**> is optional field separator. If given, overrides the comma or space field separator of DELIMITED WITH, or is added into SDF output.

CHARSEP <**expC5**> is optional separator/delimiter of character fields. If given, overrides the quota (") or <delimiter> of DELIMITED WITH, or is added in SDF output. If you wish different left and right delimiters, pass an array of two elements, where the first is left, and second is the right delimiter character/string.

LINESEP <**expC6**> is optional record separator. If given, overrides the default LF = chr(10) or CR+LF = chr(13,10) separator specified in `_aGlobSetting [GSET_C_COPY`

`_TO_NEWLINE]`. You may assign any other separator either globally by assigning value to this _aGlobSetting element, or temporary by this LINESEP clause.

HEADER <**expC7**> is an optional string which will be written as the first line in the text file, specifying e.g. the used field names. Apply only with SDF or DELIMITED clause.

WITHMEMO [<**expN8**>] includes also memo and variable length fields in the DELIMITED or SDF output. The field is TRIM()ed, soft-CR are replaced by space, and hard-CR by chr(20). You may re-define these replace characters by your own, see "Tuning" below. For DELIMITED output, the memo field is delimited same as character field by quotas or by <expC5> and the <expN8> value is ignored. For SDF output, the string is padded by space or trimmed to total <expN8> length. If <expN8> is not given for SDF or is less than or equal 0, memo field is not processed.

VIA <**expC3**> specifies the name of the RDD (replaceable database driver) to use to export the desired data, given as quoted string or character variable. The default FlagShip driver is "DBFIDX".

Description:

All records from the current database file are copied, unless limited by: scope, FOR or WHILE conditions, filter, or SET DELETED ON. Records are copied in controlling index order if such is set, otherwise in natural order. The file's permission of <file> is set according to the current database.

Since the DELIMITED [WITH] clause is ambiguous in dBase specification (it defines field separator or character delimiter), FlagShip allows you to specify explicitly the field separator by FLDSEP and the character delimiter by CHARSEP clauses, for both DELIMITED and SDF output format.

FlagShip supports three different Memo field structures: .DBT, .FPT and .DBV, see details in FUN:DbCreate(). By setting SET MEMOFILE TO DBT or FPT before COPY TO, you may convert .DBT to .FPT format (or vice versa), see example 2 below.

Multiuser:

If the output file is a database (i.e. neither SDF nor DELIMITED clause is used), it will be created and opened in EXCLUSIVE mode, and then closed again. If the target database or file exists, it will be deleted without notice before COPYing.

To avoid inconsistent target data when the source database is open in SHARED mode and SET AUTOLOCK is ON, the database may be automatically FLOCK()ed during the COPY TO operation, see Tuning below. Otherwise, you should lock it programatically by FLOCK() before COPY TO... (and UNLOCK thereafter), so it cannot be changed by others during the COPY process.

Tuning:

When SET AUTOLOCK is ON (the default), you may force the FLOCK() and UNLOCK automatically by assigning

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

The replace characters for memo field can be re-defined by

```
  _aGlobSetting[GSET_A_COPYDELIM_MEMO] := {" ", chr(20) }  // def
```

Example 1:

Prepare a mail-merging list

```
USE employee SHARED
local iCount := 0
while ! FLOCK()                  // on lock failure, retry
   if ++iCount > 20              // with msg every 2 seconds
      InfoBox("waiting for lock employee.dbf")
      iCount := 0
   endif
   sleepms(100)                  // wait 0.1 seconds
enddo
COPY TO address FIELDS Name, Lastname, Address SDF
UNLOCK
```

Example 2:

Convert database with Foxbase/FoxPro .FPT memo files to .DBT and vice versa.

```
USE olddata            // uses olddata.dbf and olddata.fpt
SET MEMOFILE TO DBT    // this is default setting
COPY TO newdata        // creates newdata.dbf and newdata.dbt

USE olddata            // uses olddata.dbf and olddata.fpt
SET MEMOFILE TO FPT    // forces to create .fpt memo file
SET DELETED ON         // copy only undeleted records
COPY TO otherdata      // creates otherdata.dbf and otherdata.fpt

SET MEMOFILE TO FPT    // forces to create .fpt memo file
USE dbtdata            // uses dbtdata.dbf and dbtdata.dbt
COPY TO fptdata        // creates fptdata.dbf and ftpdata.fpt

SET MEMOFILE TO        // reset to default = .dbt
```

Classification:

database and export to ASCII file

Compatibility:

The new file will be created with the current access rights of the database. The output text file is created using the Unix (or Windows) convention. To translate it to the DOS format (CR/LF), use the "unix2dos" utility. Omitting the <file> argument is possible in FlagShip only. The FLDSEP, CHARSEP and WITHMEMO clauses are available in FlagShip since VFS7, the HEADER since VFS8.

Translation:

```
__DBCOPY ("file", {"field1" [,"field2.."]}, ;
    {forCond}>, {whileCond}, [next], [record], [.rest.] )
__DBCOPYSDF ("file", {"field1" [,"field2.."]}, ;
    {forCond}>, {whileCond}, [next], [record], [.rest.] )
__DBCOPYDELIM ("file", "delim", {"field1" [,"field2.."]}, ;
    {forCond}>, {whileCond}, [next], [record], [.rest.] )
```

Related:

APPEND FROM, COPY FILE, COPY STRUCTURE, SET DELETED, SET MEMOFILE, DbCreate(), oRdd:CopyDB(), oRdd:CopySDF(), oRdd:CopyDelimited()

COPY STRUCTURE TO

Syntax:

```
COPY STRUCTURE TO <file>|(<expC>)
           [FIELDS <fieldList>]
```

Purpose:

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

Arguments:

<**file**>|(<**expC**>) is the name of the file to be created. The default extension is .dbf unless another extension is explicitly specified.

Options:

FIELDS <**fieldList**> is the set of fields to copy to the new database file in the order specified. The default is all fields.

Example:

```
USE employee
xx = "new_part"
? FCOUNT(), RECCOUNT()                 && 12  100
COPY STRUCTURE TO new_emp
COPY STRUCTURE TO (xx) FIELDS name,city,zip
USE new_emp
? FCOUNT(), RECCOUNT()                 && 12    0
USE (xx)
? FCOUNT(), LASTREC()                  && 3     0
```

Classification:

database

Translation:

__DBCOPYSTRUCT ("file", {"field1" [, "field2"...]})

Related:

COPY STRUCT EXTENDED, CREATE, CREATE FROM, DBCREATE(), oRdd:CopyStructure()

COPY TO...STRUCT EXTENDED

Syntax:

`COPY TO <file>|(<expC>) STRUCTURE EXTENDED`

Purpose:

Creates a structure extended database file containing the field definitions of the current database.

Arguments:

TO <**file**>|(<**expC**>) is the name of the structure extended database file.

Description:

COPY STRUCTURE EXTENDED creates a database file with four fields: FIELD_NAME, FIELD_TYPE, FIELD_LEN and FIELD_DEC, and fills it with field definitions of the current database file. Thereby, it is possible to create and modify structures of database files from within an application. CREATE FROM is used to create a new database file from a structure extended file. To create only an empty structure extended file, use the CREATE command.

	Field name	Type	Length,	deci
1	FIELD_NAME	Character	10	
2	FIELD_TYPE	Character	1	
3	FIELD_LEN	Numeric	3	0 see note
4	FIELD_DEC	Numeric	3	0

Note: character fields up to 64 KBytes are supported by FlagShip. Fields greater than 255 characters are defined with a combination of the FIELD_DEC and FIELD_LEN fields to remain compatible with other xBASE dialects. After copying STRUCTURE EXTENDED, you can use the following formula to determine the length of any character field:

```
act_len = IF (FIELD_TYPE = "C" .AND. FIELD_DEC != 0, ;
             (FIELD_DEC * 256) + FIELD_LEN, FIELD_LEN)
```

The structure database may be extended with additional fields for the user's own purposes.

Example:

```
USE Employee
xx = "new_part"
? FCOUNT(), RECCOUNT()                  && 12  100
COPY STRUCTURE EXTENDED TO New_stru
COPY STRUCTURE EXTENDED TO (xx) FIELDS name,city,zip, note
USE New_stru
? FCOUNT(), RECCOUNT()                  && 4   12

USE (xx) NEW
? FCOUNT(), LASTREC()                   && 4    3
LOCATE FOR UPPER(TRIM(field_name)) == "NOTE"
IF FOUND()
   REPLACE field_dec WITH 4, ;          // 4 * 256 = 1024
           field_len WITH 76            // + 76    = 1100
ENDIF
CLOSE
CREATE new_name FROM (xx)
USE new_name
? LEN(note)                             // 1100
```

Classification:

database

Translation:

__DBCOPYXSTRUCT ("file")

Related:

CREATE, CREATE FROM, FIELD(), TYPE(), DBCREATE()

COUNT ... TO

Syntax:

```
COUNT [<scope>]
        [FOR <condition>] [WHILE <condition>]
        TO <memvar>
```

Purpose:

Counts records in the current working area, which fall into the given scope and fulfill the specified conditions. The result is stored to the specified memory variable.

Arguments:

<**memvar**> is the memory variable where the result of counting is stored. If the variable does not exist, a new autoPRIVATE is created as numeric.

Options:

<**scope**> is the part of the current database file to be counted. The default scope is ALL.

<**condition**>**:** The FOR clause specifies that the set of records meeting the condition within the given scope, are to be counted. The WHILE clause stops counting when the first record not fulfilling the condition is reached.

Example:

```
USE magazine
? RECCOUNT()                           &&   100
COUNT FOR Price > 2 TO Exp             &&    12
COUNT FOR Price <= 2 TO Cheap          &&    88
```

Classification:

database

Translation:

```
<var> := 0
DBEVAL({|| <var> := <var> + 1}, ;
      {forCond}>, {whileCond}, [next], [record], [.rest.])
```

Related:

AVERAGE, SUM, TOTAL, DBEVAL(), oRdd:Count()

CREATE

Syntax:

```
CREATE <file>|(<expC>)
        [ALIAS <alias>]
        [NEW]
        [VIA <driver>]
```

Purpose:

Creates an empty structure extended database file, and leaves it open in the selected working area.

Arguments:

<**file**>|(<**expC**>) is the name of the empty structure extended file.

Options:

ALIAS <**alias**> is the name to be associated with the working area. If not specified, the main part of the <file> name is assigned to <alias>.

NEW selects an unused working area making it the current one and opens the database <file> there. The clause is equivalent to SELECT 0 prior to the CREATE... command. If this clause is not given, the database is opened in the current SELECTed working area.

VIA <**driver**> defines the replaceable database driver (RDD) to process the current working area. The default driver is "DBFIDX".

Description:

The empty structure extended file consists of four fields: FIELD_NAME, FIELD_TYPE, FIELD_LEN and FIELD_DEC, see CREATE FROM. To form a new database file, use CREATE FROM.

	Field name	Type	Length,	deci
1	FIELD_NAME	Character	10	
2	FIELD_TYPE	Character	1	
3	FIELD_LEN	Numeric	3	0
4	FIELD_DEC	Numeric	3	0

Example:

see example of CREATE FROM... and DBCREATE()

Classification:

database

Translation:

```
__DBCREATE ("file")
```

Related:

DBCREATE(), CREATE FROM, COPY STRUCTURE EXTENDED

CREATE ... FROM

Syntax:

```
CREATE <file1> | (<expC1>) FROM <file2> | (<expC2>)
        [ALIAS <alias>]
        [NEW]
        [VIA <driver>]
```

Purpose:

Creates a new database file from a structure extended file, and leaves it open in the selected working area.

Arguments:

<**file1**>|(<**expC1**>) is the name of the new database file to be created.

<**file2**>|(<**expC2**>) is the name of a structure extended file, from which the field definitions for <file1> will be used during the creation process.

Options:

ALIAS <**alias**> is the name to be associated with the working area. If not specified, the main part of the <file> name is assigned to <alias>.

NEW selects an unused working area making it the current one and opens the database <file> there. The clause is equivalent to SELECT 0 prior to the CREATE... command. If this clause is not given, the database is opened in the current SELECTed working area.

VIA <**driver**> defines the replaceable database driver (RDD) to process the current working area. The default driver is "DBFIDX".

Description:

CREATE FROM creates a new database file according to the information contained in a structure extended file. A database file is regarded as a structure extended file, if it contains the following four fields:

	Field name	Type	Length,	deci
1	FIELD_NAME	Character	10	
2	FIELD_TYPE	Character	1	
3	FIELD_LEN	Numeric	3	0
4	FIELD_DEC	Numeric	3	0

A structure extended file can contain any number of fields, providing that these four fields exist. The order in which the fields appear is of no importance. Only the four fields are used when creating a new dbf file.

To create a character field longer than 256 characters, specify the FIELD_DEC equal to the INT() of the required length divided by 256, and the FIELD_LEN equal to the remainder of the length divided by 256. The formula is

```
act_len = IF (FIELD_TYPE = "C" .AND. FIELD_DEC != 0, ;
             (FIELD_DEC * 256) + FIELD_LEN, FIELD_LEN)
```

The file's permission of <file2> is set for <file1>.

Note, that the function DBCREATE() performs the same functionality, but is easier to handle. See additional description there.

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 set the field containing glyphs correspondingly (e.g. to 30 or more characters to accept 10 Japanese or Chinese glyphs).

Example 1:

```
CREATE New_stru
USE New_stru
APPEND BLANK
REPLACE Field_name WITH "Id", ;
   Field_type WITH "N",;
   Field_len  WITH 5,  ;
   Field_dec  WITH 0
APPEND BLANK
REPLACE Field_name WITH "Lastname", ;
   Field_type WITH "C", ;
   Field_len  WITH 20,  ;
   Field_dec  WITH 0
APPEND BLANK
REPLACE Field_name WITH "Birthdate", ;
   Field_type WITH "D", ;
   Field_len  WITH 8,   ;
   Field_dec  WITH 0
APPEND BLANK
REPLACE Field_name WITH "Longfield", ;
   Field_type WITH "C", ;
   Field_len  WITH 160, ;                // 4000 % 256
   Field_dec  WITH 15                    // int(4000/256)
USE
CREATE New_file FROM New_stru

** use the new database

USE New_file
? FCOUNT(), FIELD(1), RECCOUNT()         //  4  ID  0
? LEN(longfield)                         //  4000
```

Example 2:

This example is equivalent to Example 1:

```
aDbStru := {{"Id", "N", 5, 0}, ;
            {"Lastname","C",20,0}, ;
            {"Birthdate","D",8,0}, ;
            {"Longfield","C", 4000, 0}}
DbCreate ("New_file", aDbStru)
```

Classification:

database

Compatibility

FlagShip supports character field length up to 64534 bytes (FIELD_DEC = 252, FILED_LEN = 85), Clipper up to 32 or 64 KBytes (release dependent), dBASE III up to 256 Bytes. To remain compatible to DOS, the maximal record length (the sum of field lengths) is in all cases 64534 Bytes.

Translation:

```
 __DBCREATE ("file1", "file2")
```

Related:

DBCREATE(), COPY STRUCTURE EXTENDED, CREATE, SET MEMOFILE, oRdd:CreateDB()

DECLARE

Syntax:

```
DECLARE <array> [<dim>]
DECLARE <array> [<dim1>,<dim2>,<dimn>]
DECLARE <array> [<dim1>][<dim2>][<dimn>]
DECLARE <array> := {<initializer>}
```

Purpose:

Creates the specified one-dimensional or multi-dimensional array(s) of class type PRIVATE.

Arguments:

In this case, the square brackets around <dim> do **not** specify an optional argument, but are a required part of the syntax.

<**array**> is the name of the array to be created.

<**dim**> is the dimension of the array. With one-dimensional arrays, its syntax is [<expN>]. With multi-dimensional arrays, the dimensions may be given together in the [ ] bracket separated by commas or each dimension separately in [ ] brackets without commas. You may declare more than one array in one DECLARE statement.

Array elements can be handled like ordinary memory variables. Different elements of the same array can have different types. Each element may contain another sub-array (non-symmetric structure), see LNG.2.6.4.

Initializing:

Array elements can be declared and initialized with a starting value using an array (literal) constant (see LNG.2.7) which includes any valid expression, and the assign := operator, e.g.:

```
DECLARE arr1 := {}                        // creates   arr1[0]
DECLARE arr2 := {0,date(),"test",.T.}  // creates   arr2[4]
DECLARE arr3 := {{1,2},{3,4}}          // creates arr3[2,2]
DECLARE arr4:= {1, {2,3}, {"test",.T.,NIL,4, {5,DATE()}},6}
```

The above arrays arr1, arr2 and arr3 are symmetric, while the declaration of array4 specifies a non-symmetric array. If no explicit <initializer> is specified, the variable is given an initial value of NIL. The exception is the zero length literal array { }.

Description:

DECLARE creates private arrays. This hides all the private arrays or variables with the same name created in higher level procedures. Declaring an array LOCAL, STATIC or PUBLIC is another way of specifying the visibility scope. DECLARE and PRIVATE are equivalent statements.

FlagShip uses one variable slot per array. The maximum number of array elements is 65535 per dimension, up to 65535 dimensions will be handled. The theoretical size of a symmetric array is therefore 4 billion (* 28 bytes), if non-symmetric, even more.

Arrays can be declared or used from within macro variables, see LNG.2.10. As parameters to functions and procedures, arrays are passed by reference, while array elements are passed by the usual (variable) convention. See PROCEDURE and FUNCTION.

Example:

```
name = "arr1"
len = 20
DECLARE &name.[len]                         && arr1[20]
DECLARE arr2[15], arr3[5,6]
AFILL(arr1, "John")

? &name.[5]                                 && "John"
? LEN(arr1), arr3[5,2]                      && 20 NIL
? TYPE("name"), TYPE(name)                  && "arr1"  "A"
? TYPE("arr1"), TYPE("arr1[1]")             && "A" "C"

DECLARE uarr:= {1, {2,3}, {"test",.T.,NIL,4, {5,DATE()}},6}
? VALTYPE(uarr), LEN(uarr)                  // A  4
? VALTYPE(uarr[2]), LEN(uarr[2])            // A  2
? VALTYPE(uarr[3,5]), LEN(uarr[3,5])        // A  2
? uarr[1], uarr[3][4], uarr[3,5,1]          // 1  4  5
```

Classification:

programming

Compatibility:

Multi-dimensional and non-symmetric arrays are new in FS4 and C5. Clipper allows arrays with a maximum of 4096 elements, dBASE IV two-dimensional arrays with a maximum of 1170 elements. Unlike Clipper, FlagShip supports saving and restoring arrays to .mem files, see SAVE TO.

Related:

PRIVATE, PUBLIC, LOCAL, STATIC, AADD(), ARRAY(), ACOPY(), ACLONE(), ADEL(), ACHOICE(), ADIR(), AFILL(), AINS(), ASCAN(), ASORT(), DBEDIT()

DELETE

Syntax:

```
DELETE [<scope>]
        [FOR <condition>]
        [WHILE <condition>]
```

Purpose:

Marks records in the current working area for deleting.

Options:

<**scope**> is the part of the current database file to be deleted. The default scope is the current record if a condition is not specified, or ALL if a condition is specified.

<**condition**>**:** The FOR clause specifies that the set of records meeting the condition within the given scope is to be deleted. The WHILE clause stops deletion when the first record not fulfilling the condition is reached.

Description:

After deletion, the records remain in the database until removed by PACK or reinstated by RECALL. They may be queried with DELETED(), and filtered out with SET DELETED ON. Removing all records from a database file is done more easily with ZAP than with DELETE ALL and PACK. If SET DELETED is ON, the record stays visible until the record pointer is moved.

Multiuser:

In a multiuser / multitasking environment, DELETE requires that the records be locked with RLOCK() if deleting a single record, or by FLOCK() or an EXCLUSIVE open, to delete multiple records. Otherwise, AUTORLOCK() is used automatically, if SET AUTOLOCK is active. See LNG.4.8.

Example:

```
SET DELETED ON
USE employee
WHILE NETERR() ; USE employee ; END
? RECCOUNT()                              &&   100
COUNT TO Sick_no FOR Sick_days > 30       &&    12
WHILE !FLOCK() ; END
DELETE FOR Sick_days > 30
UNLOCK
COUNT TO Healthy_no                       &&    88
```

Classification:

database

Translation:

```
DBDELETE ()
DBEVAL ({|| DBDELETE()}, for, while...)
```

Related:

RECALL, DELETED(), SET DELETED, PACK, ZAP, oRdd:Delete()

DELETE FILE

Syntax:

`DELETE FILE <file>|(<expC>) [WITHMSG]`

or

`ERASE <file>|(<expC>) [WITHMSG]`

Purpose:

Removes a file from disk.

Arguments:

<**file**> is the name of the file (including extension) to be deleted. A full path may be specified. If omitted, only the current directory is searched; the SET PATH or SET DEFAULT path is ignored. Standard Unix wildcards using ?, *, [..] are supported.

Option:

<**WITHMSG**> if specified, run-time warning message is displayed on failure. Default is no warning message on failure.

Description:

The file will be deleted without any warning. The consequences are not recoverable. The user must have at least "w" access rights for the file and "x" for the directory.

The success can be checked using DOSERROR().

Example:

```
? FILE ("data.tmp")                        && .T.
DELETE FILE dat?.t*p
? FILE ("data.t*p")                        && .F.
? DOSERROR()                               && 0
```

Classification:

system, file access

Compatibility:

Wildcard support, WITHMSG clause and the DOSERROR() checking is available in FlagShip only.

The ERASE or DELETE FILE command is equivalent to the Unix command "rm" or similar to the DOS/Windows command "DEL".

The command 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.

Translation:

FERASE ("file", [.msg.])

Related:

CLOSE, USE, CURDIR(), FILE(), FS_SET ()

DELETE TAG

Syntax:

```
DELETE TAG <expC1>
        [IN|OF <file1>]
        [ , <expC2> [IN|OF <file2>]... ]
```

Purpose:

Deletes a tag (subindex) or the whole index file.

Arguments:

<**expC1**> is a literal string or parenthesized character expression that represents the subindex (tag) name, within the index file. For the default "DBFIDX" replaceable driver RDD, the <expC1> is equivalent to <file1>.

IN <**file1**> (or **OF** <**file1**>) is a literal string or parenthesized character expression that represents the index file name containing the <expC1> subindex (tag). For the default "DBFIDX" replaceable driver RDD, which contains only one subindex (tag), the <file1> entry is ignored. If the <file1> is omitted, all active indices in the current working area are searched for the subindex name <expC1>.

Description:

This command is designed to delete one tag (subindex) in a multiple index file supplied by other RDDs. With single index files, like the default "DBFIDX", the whole index file is deleted, equivalent to the DELETE FILE command.

If the removed <expC1> is the active index, the next tag of the index file is selected. If the removed <expC1> is the last, or the only one subindex (tag) in the index file <file1>, the index is deselected, equivalent to the CLOSE INDEXES command or SET INDEX TO without arguments.

Multiuser:

With some RDD drivers, the database must be used exclusively, but is not required for the default "DBFIDX" driver.

Example 1:

```
USE employee VIA "dbfmdx" NEW
SET INDEX TO employee
DELETE TAG persno OF employee
```

Example 2:

```
USE employee NEW
SET INDEX TO persno, persname
? INDEXORD(), pers_num, name            // 1  101  Smith
DELETE TAG persname
? INDEXORD(), pers_num, name            // 0  295  Miller
? FILE("persname" + INDEXEXT())         // .F.
```

Classification:

database

Compatibility:

Available in FS4, C5, DB4. The clause OF is not available in C5, but IN only.

Translation:

```
ORDERDESTROY (exp1, file1)
```

Related:

USE, ORDDESTROY(), DBSETDRIVER(), oRdd:DeleteOrder()

DIR

Syntax:

`DIR [<skeleton>]`

Purpose:

Displays a listing of files from the specified path.

Options:

<**skeleton**> is the standard wildcard notation for files (* and ?) used to select files for display. It may include a directory path to specify the tree structure from the current (relative path) or the root (absolute path) directory to the desired files. If it is omitted, the current directory is assumed. The directory names can be separated by a slash ("/") or by the back slash ("\") character.

If <skeleton> is not specified, only database files .dbf will be displayed. Otherwise, all the files that match the skeleton will be displayed.

Description:

The .dbf list includes file name, date of the last update and number of records. Specifying a <skeleton> displays the files that match a file name pattern. The list includes file names, attributes, their size and date of the last update, in a format similar to the Unix command "ls -l" or Windows "DIR".

Note, that the information about specific file is also available via the DIRECTORY() or ADIR() functions.

Example:

```
DIR *.*                      // same as: ls -l *.*
DIR ("./[a-d]*.dbf")         // same as: ls -l ./[a-d]*.dbf
DIR                          // standard header (1)
#ifdef FlagShip
   FS_SET ("load", 1, "FSsortab.ger")
   FS_SET ("set", 1)
   DIR                       // german header (2)
#endif
```

```
------------------- Output using DIR *.* ------------------

-rwxrwxr-x 1 jan   program 459198 Sep 23 14:55 a.out
-rw-rw-r-- 1 peter program   1845 Apr 27 19:03 adress.dbf
-rw-rw-r-- 1 hugo  user       657 Jul 15 16:08 adress1.dbf
-rw-rw-r-- 1 sven  program     30 Sep 01 18:42 dummy.prg
-rw-rw-r-- 1 guest guest     1253 Jun 29 11:52 dummy.txt
-rw-rw-r-- 1 peter program    115 Sep 11 14:28 tvarmac.prg
------------------- Output using DIR *.dbf ----------------

-rw-rw-r-- 1 peter program   1845 Apr 27 19:03 adress.dbf

------------------- Output (1) using DIR ------------------

Database Files    # Records    Last Update     Size
adress.dbf               15    01/27/94        1845
adress1.dbf               4    07/15/93         657

------------------- Output (2) using DIR and FS_SET()------

Datenbanken          Saetze  Letzt.Aender   Groesse
adress.dbf               15    27.01.94        1845
adress1.dbf               4    15.07.93         657
```

Classification:

system, file access

Compatibility:

The header for displaying the databases is user-definable in FlagShip via FS_SET("load"). The <skeleton> output on Unix is the same as in "ls -l" and differs from the output on DOS.

Translation:

__DIR ("skeleton")

Related:

DIRECTORY(), ADIR(), PUBLIC FlagShip, #ifdef FlagShip, FS_SET()

DISPLAY

Syntax:

```
DISPLAY [OFF] [<scope>] <expList>
        [FOR <condition>]
        [WHILE <condition>]
        [TO PRINTER]
        [TO FILE <file>|(<expC>) [ADDITIVE]]
```

Purpose:

Displays the result of one or more expressions for each processed record.

Arguments:

<**expList**> is the list of values displayed for each processed record.

Options:

<**scope**> is the part of the current database file to display. The default scope is the current record. If a condition is specified, the scope becomes ALL.

<**condition**> specifies additional FOR or/and WHILE filtering. See the general command description.

OFF: Suppresses the display of the record number.

TO PRINTER: echoes output to a printer file. To disable the screen output, use SET CONSOLE OFF.

TO FILE: echoes output (ADDITIVE) to the specified file. See also the general command description.

Description:

DISPLAY sends the results of the <expList> to screen in a tabular format, each column being separated by a space. DISPLAY is similar to LIST, with the difference that its default scope is NEXT 1, rather than ALL as in LIST.

Example:

```
Esc interrupts DISPLAY:

USE Employee
DISPLAY Lastname, Firstname, Birthdate FOR INKEY() <> 27
```

Classification: sequential output

Compatibility:

The ADDITIVE option is available in FlagShip only. C5 will accept but ignores it, if "/ustd.fh" is used.

Translation:

```
__DBLIST (.off., {exp1 [,exp2...]}, .all., {for}, {while},;
          next, rec, .rest., .toPrint., "file")
```

Related:

LIST, SET EXTRA

DO

Syntax:

```
DO <procname> [WITH <parameterList>]
```

Purpose:

Executes a user-defined-procedure (UDP).

Arguments:

<**procname**> is the name of the procedure to be executed. It can be written either in FlagShip or in the C language using the Extend System.

Options:

WITH <**parameterList**> allows to pass any number of arguments, separated by commas, to the UDP, which receives them as parameters. Each argument may be a single variable, field, array, array element, expression, or an object. Before branching to the UDP, the arguments in the <parameterList> are evaluated. When the argument is an expression, macro-evaluation, constant, or function call, it is passed as a reference to a temporary variable. Field variables have to be preceded by an alias`->` or `FIELD->`, or enclosed in parentheses.

Arguments can be skipped or left off the end of the list. The number of arguments specified does not have to match the number of parameters specified in the called procedure. If the number of arguments is less than the number of parameters, the parameter variables with no corresponding arguments are initialized with a NIL value when the procedure is called.

A skipped argument, given a comma only, also initializes the corresponding parameter to NIL. To detect the position of the last argument passed in the <parameterList>, use PCOUNT(). To detect a skipped argument, compare the receiving parameter to NIL or TYPE() / VALTYPE() to "U".

Parameter passing using the WITH clause is done by reference by default. This means, the formal parameter receives the address of the current argument. Changes made to the parameter within the UDP called will be reflected automatically in the argument; only constants, expressions and database fields arguments remain unchanged. Closing an argument in parentheses, passes it "by value" instead.

Description:

The DO statement calls a procedure (UDP), optionally passing arguments to the called routine. It performs the same action as a user-defined function (UDF) except that DO passes parameters by reference as a default, and that a UDP has no return value.

Compilation:

When the FlagShip compiler is invoked without the -m option, it searches the current directory for a source file with the same name in order to compile it, every time it finds a DO statement and the name of the procedure is unknown. If it is not found, the compiler considers the procedure externally. At link-time, the linker looks for such unresolved externals in other object files or libraries given. If the external had not been found, the linker prints an error message like "unresolved external _bb_<procname>".

Example:

The FlagShip will also compile the file xchange.prg automatically (because of the DO Xchange... statement) when compiled by: "FlagShip test.prg" (i.e. without -m switch)

```
*** File TEST.PRG
DO DispWork WITH "Dept", "Markt"
number1 := 10
number2 := 20
DO Xchange WITH number1, number2
? number1, number2                     && 20   10
DO Xchange WITH number1, (number2)
? number1, number2                     && 10   10 (unchanged)
QUIT

PROCEDURE DispWork                     && list fields
PARAMETERS field1, field2
LIST &field1, &field2
RETURN
*** eof TEST.PRG
*** File XCHANGE.PRG          && ┐ do not declare PROCEDURE
* PROCEDURE Xchange           && ┘ for the same file name
PARAMETERS num1, num2
PRIVATE dummy
dummy = num1
num1  = num2
num2  = dummy
RETURN
*** eof XCHANGE.PRG
```

Classification:

sequential output

Compatibility:

FlagShip supports any number of parameters, Clipper up to 128, dBASE up to 50. The file name for additional compilation is searched in lower case only.

Related:

PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, RETURN, SET PROCEDURE

DO CASE..CASE ... ENDCASE

Syntax:

```
DO CASE
CASE <condition>
     <statements> ...
[CASE <condition>
     <statements> ...]
[OTHERWISE
     <statements> ...]
ENDCASE | END
```

Purpose:

A control structure to execute a set of statements according to the associated conditions.

Arguments:

DO CASE defines the structure beginning.

ENDCASE specifies the end of the structure. ENDCASE may be abbreviated with END.

Options:

CASE <**condition**>**:** The condition is a logical value or any expression resulting in logical. If the condition given is met, the statements which follow will be executed until the next CASE, OTHERWISE, or ENDCASE command is encountered; and the program control is passed to the next statement following the ENDCASE.

When the condition is not met, the control branches to check the next CASE condition, the OTHERWISE or ENDCASE command.

OTHERWISE: If all CASE conditions are false, the statements following the OTHERWISE command up to the ENDCASE are executed. If this option is not specified, and all the CASE conditions are false, no statements inside the CASE are executed.

Description:

The DO CASE ...CASE .. .ENDCASE is equivalent to the IF...ELSEIF ...ELSE ... ENDIF control structure, see also LNG.2.5.

There is no limit to the number of CASEs inside the structure. This structure can be nested to any depth with other control structures.

Example:

```
* this structure:                  * is equivalent to:
hour = VAL(TIME())                | hour = VAL(TIME())
DO CASE                           |
   CASE hour < 10                 | IF hour < 10
      str="morning"               |    str="morning"
   CASE hour < 15                 | ELSEIF hour < 15
      str="day"                   |    str="day"
   CASE hour < 18                 | ELSEIF hour < 18
      str="afternoon"             |    str="afternoon"
   CASE hour < 20                 | ELSEIF hour < 20
      str="evening"               |    str="evening"
   OTHERWISE                      | ELSE
      str="night"                 |     str="night"
ENDCASE                           | ENDIF
@ 10,30 SAY "good " + str + " !" | @ 10,30 SAY "good " + str + " !"
```

Classification:

programming

Related:

IF, IF() / IIF()

DO WHILE ... ENDDO

Syntax:

```
[DO] WHILE <condition>
     <statements>...
[EXIT]
     <statements>...
[LOOP]
     <statements>...
ENDDO | END
```

Purpose:

A control structure to execute a looping when the <condition> is true (.T.).

Arguments:

WHILE <**condition**> is the controlling condition that is evaluated every time the DO WHILE or WHILE statement executes. The <condition> is a logical value or any expression resulting in logical.

ENDDO specifies the end of the structure. If encountered, the program control is passed back for the next DO WHILE condition check. ENDDO may be abbreviated to END.

Options:

EXIT: The EXIT statement terminates the looping, and branches unconditionally to the statement following the ENDDO. Any number of EXITs within the structure are accepted.

LOOP: The LOOP statement repeats the loop by immediately branching back to the DO WHILE condition check. Any number of LOOPs within the structure are accepted.

Description:

The DO WHILE structure executes a block of statements repetitively, as long as the specified condition evaluates to true (.T.). The control is passed into the structure and proceeds until an EXIT, LOOP or ENDDO is encountered. ENDDO and LOOP pass control back to the beginning of the DO WHILE statement for a new iteration.

The DO WHILE construct terminates or is not processed at all, when the condition evaluates to false (.F.). Control is then passed to the statement immediately following the ENDDO.

Example 1:

Repeat until construct: You can also use the DO WHILE to create a repeat until looping construct as follows:

```
more = .T.                                  * or:
DO WHILE more                               DO WHILE .T.
   IF <end condition>                          IF <end condition>
      more = .F.                                  EXIT
   ENDIF                                       ENDIF
ENDDO                                       ENDDO
```

Example 2:

Traversing a database file: The DO WHILE looping construct enables you to move sequentially through a database file, as you can see in the following two examples:

```
DO WHILE .NOT. EOF()
   <statements>...
   IF <repeat the same record>
      LOOP
   ENDIF
   SKIP
ENDDO
```

Example 3:

This example sequentially scans a database file, processing records that match a condition:

```
LOCATE FOR <condition>
DO WHILE FOUND()
   <statements>...
   CONTINUE
   <statements>...
ENDDO
```

Example 4:

Macros on the DO WHILE command line: Macro variables can be used without any limitations in the DO WHILE condition, partially or entirely.

```
var = "upper(trim(Name)) == 'SMITH'"
DO WHILE &var .and. !EOF()
   ? name, city
   var = "zip = " + STR(zip)
   SKIP
ENDDO
```

Classification:

programming

Compatibility:

Optionally shortening DO WHILE to WHILE is new in FS4, according to Clipper 5.x.

Related:

FOR, IF, RETURN

EJECT

Syntax:

`EJECT`

Purpose:

Causes an advance to a new page while printing.

Description:

In FlagShip, printer output normally goes to an internal print file, except when SET PRINTER TO <device> is specified (or with SET GUIPRINT ON). This avoids printer output being garbaged in multiuser mode.

EJECT sends a form-feed character [chr(12)] to the active SET PRINTER TO file, if specified, or else to the default spooling file (see SET PRINTER). The form feed is sent only when SET PRINT is ON (see also Tuning) or SET GUIPRINT is ON or with PrintGui(.T.).

EJECT also resets the internal printer row and column tracers of PROW() and PCOL() to zero. You may also reset the tracers only with SETPRC().

You may tune the printer driver by FS_SET("prset").

When printing via GUI/GDI driver (SET GUIPRINT ON, PrintGui(.T.)), EJECT command sends form-feed to printer buffer (i.e. creates new page) and increases the page number. If SET CONSOLE is ON or SET DEVICE TO is SCREEN, it also invokes CLEAR SCREEN (see tuning). For this mode, EJECT is equivalent to _oPrinter:GuiNewPage() and optional CLEAR SCREEN.

Tuning:

In FlagShip 6.x and earlier, the EJECT has sent FormFeed to printer spooler file regardless SET PRINT ON or OFF was set. You may force this behavior by assigning

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

To avoid CLEAR SCREEN when printing via GUI/GDI driver (see above), assign

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

EJECT with PrintGui(.T.) however always set the cursor position to top = SetPos(0,0).

Example 1: *print traditional*

```
USE stock INDEX stockno
LIST stockno, article, price TO PRINT
EJECT
LIST stockno, article, retail TO PRINT
EJECT                                      // new page
prn_file = FS_SET("printfile")             // get file name
SET PRINT TO                               // flush file
RUN ("lp -dlaser -s " + prn_file)          // spool it (Linux)
```

Example 2: *Print & EJECT via GUI*

```
local nPage := 1
_aGlobSetting[GSET_L_EJECT_CLEARSCREEN] := .F. // see above
? "Printing, please wait..."
SET CONSOLE OFF              // screen output is not required
PrintGui(.T.)                // select printer, start printout
for ii := 1 to 10
   ? "Line",ltrim(ii),"on page",ltrim(nPage)   // output to printer
next
eject                        // printer: new page
nPage++
for ii := 1 to 10
   ? "Line",ltrim(ii),"on page",ltrim(nPage)   // output to printer
next
PrintGui()                   // end printout, send to printer
SET CONSOLE ON               // enable screen output
wait "done, any key..."
```

Classification:

sequential and GUI printer output

Compatibility:

Spooled printer output is supported only in FlagShip.

Translation:

__EJECT ()

Related:

SET PRINTER, SET EJECT, PRINTGUI(), PCOL(), PROW(), SETPRC(), FS_SET("print"), FS_SET("prset"), OBJ.Printer class

ERASE

Syntax:

`ERASE <file>|(<expC>) [WITHMSG]`

or

`DELETE FILE <file>|(<expC>) [WITHMSG]`

Purpose:

Removes a file from disk.

Arguments:

<**file**> is the name of the file (including extension) to be deleted. A full path may be specified. If omitted, only the current directory is searched, the SET PATH or SET DEFAULT path is ignored. Standard Unix wildcards using ?, *, [..] are supported.

Option:

<**WITHMSG**> if specified, run-time warning message is displayed on failure. Default is no warning message.

Description:

The file will be deleted without any warning. The consequences are not recoverable. The user must have at least "w" access rights for the file and "x" for the directory.

The success can be checked using DOSERROR().

Example:

```
? FILE ("data.tmp")                        && .T.
ERASE data.tmp
? FILE ("data.tmp")                        && .F.
ERASE "[k-m]d*.tm*"
? DISERROR()                               && 0
? FILE ("[k-m]d*.tm*")                     && .F.
```

Classification:

system, file access

Compatibility:

Wildcard support, WITHMSG clause and the DOSERROR() checking is available in FlagShip only.

The ERASE or DELETE FILE command is equivalent to the Unix command "`rm`" or similar to the DOS command "DEL".

The command 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.

Translation:

FERASE ("file", [.msg.])

Related:

CLOSE, USE, CURDIR(), FILE(), FS_SET ()

EXPORT INSTANCE

Syntax 1:

```
[STATIC] CLASS <ClassName> [INHERIT <SuperClass>]
```

and optional:

```
INSTANCE <Name> [:= <exp>] [AS <type>]
EXPORT [INSTANCE] <Name> ...
HIDDEN [INSTANCE] <Name> ...
PROTECT [INSTANCE] <Name> ...
```

Syntax 2:

```
PROTOTYPE [STATIC] CLASS <ClassName>
        [INHERIT <SuperClass>]
```

and optional:

```
INSTANCE <Name> [AS <type>]
EXPORT|HIDDEN|PROTECT [INSTANCE] <Name> [AS <type>]
```

See detailed description in the CLASS command.

EXTERNAL

Syntax:

`EXTERNAL <nameList>`

Purpose:

Explicitly requests the procedures (UDP) or functions (UDF) to be linked into the application.

Arguments:

<**nameList**> is a comma separated list of UDP/UDF names, Extend C functions and format file names which should be added to the symbol table.

Description:

EXTERNAL is used in case where user-defined-procedures or standard functions are called only from within macro statements, included in the INDEX key, or passed as a character variable to ACHOICE(), DBEDIT() or MEMOEDIT(). Such procedures/ functions might not be linked-in at all if not specified in an EXTERNAL statement. The same may apply for standard FlagShip functions not explicitly used in the application.

Generally: when you are using a UDP, UDF or standard function for the above purposes, and are not sure about calling it also by name elsewhere in the application, use EXTERNAL to ensure the name is known to the linker. Otherwise, a run-time error "unresolved external" may occur during application execution.

Example:

```
* file test.prg
EXTERNAL my_proc

var = "my_proc"
DO &var WITH 1, 2
* eof test.prg
* File my_proc.prg
* automatically declared: PROCEDURE my_proc
PARAMETERS p1, p2
:
RETURN
* eof my_proc.prg
```

Classification:

compiler/linker

Compatibility:

FlagShip does not support Clipper's division of pre-linked functions vs. libraries.

Related:

SET PROCEDURE TO, REQUEST

FIELD

Syntax:

`FIELD <fieldList> [IN <alias>]`

Purpose:

Declares database field names to be used as if implicitly aliased.

Arguments:

<**fieldList**> is a list of names to be declared as fields to the compiler. The fields from the <fieldList> are accessed as `FIELD->`fieldname or <alias>->fieldname.

Options:

IN <**alias**> specifies an alias to assume when there are unaliased references to the names specified in the <fieldList>. The fields will be accessed in the same manner as if <alias>**-**>fieldname is given.

If the IN.. clause is not specified, unaliased references to <fieldList> are treated as if they are preceded by `FIELD->`fieldname alias.

Scope:

The scope of the FIELD declaration is normally the procedure or function in which it occurs. If the declaration is given prior to the first PROCEDURE or FUNCTION **and** the compiler switch `-na` is used, the scope becomes the entire .prg file.

In FlagShip, the declarator may be placed anywhere in the code; the compiler starts the aliasing of the FIELD variables after encountering the declaration and continues to the end of the corresponding procedure/ function (or the .prg file).

Description:

The FIELD declaration allows the compiler to resolve references to variables in the <fieldList> without explicit aliases. The FIELD statement has no effect on variable/field references within macro expressions.

The FIELD statement neither opens a database file nor verifies the existence of the specified fields. It is useful primarily to ensure correct references to fields to which accessibility is known to be guaranteed at runtime. At runtime, the field variables are made accessible with the USE command. Attempting to access the fields when the associated database is not in USE will cause a run-time error.

When **accessing** an ambiguous variable, which was not specified by FIELD, MEMVAR or <alias>, fields of the current working area have precedence over the PRIVATE, autoPRIVATE or PUBLIC variables with the same name. The same is true for accesses/replaces in the @...GET/READ command.

To **replace** a field value, the REPLACE command, an aliased field name, or the FIELD declarator have to be used; otherwise a memory variable will be used or a new autoPRIVATE created.

To check for and/or prevent from ambiguous occurrences of variables, the -w or -am option of the FlagShip compiler may be used.

Example:

Without the -w compiler option, the missing "FIELD persno" declaration may pass unnoticed; e.g. the field "persno" below remains unchanged instead of replaced.

```
FUNCTION output (first, last)
FIELD name, lastname, zip, address, printed
LOCAL record
GOTO (first)
WHILE !EOF() .AND. RECNO() <= last
   record = RECNO()
   printed := .T.      // same as: REPLACE printed WITH .T.
   persno  := record   // PRIVATE persno is created/updated
** REPLACE persno WITH record     // field will be replaced
** FIELD->persno := record        // field will be replaced
** (ALIAS())->persno := record    // field will be replaced

** FIELD persno        // the ABOVE persno is not affected
   ? record, name, lastname, zip, address, ;
     persno                       // output: database field
   SKIP
END
RETURN NIL
```

Classification:

programming, database

Compatibility:

In FlagShip, the declarator may be placed anywhere in the code; in C5, the declarator position is fixed.

Related:

LOCAL, MEMVAR, PRIVATE, PUBLIC, STATIC, @..GET

FIND

Syntax:

`FIND <keyC>|(<expC>)|&<memvar>`

Purpose:

Searches through an index to find the first key matching the specified character string and positions the record pointer onto the corresponding record.

Arguments:

<**keyC**> is part of or the entire index key to be found. If (<expC>) is specified, FIND behaves similar to SEEK.

Description:

A search of the master index starts from the first key. If a match is found, the record pointer is positioned to the record number found in the index, FOUND() returns TRUE, EOF() returns FALSE.

If the searched for value is not found, the current state of SET SOFTSEEK affects the values returned from FOUND(), EOF() and the position of the record pointer:

- If SOFTSEEK is OFF (the default), FOUND() returns FALSE, EOF() returns TRUE, and the database is positioned at eof = LASTREC() +1.
- If SOFTSEEK is ON, and there are keys with values greater than the search argument, the database pointer is positioned to the first record with a key value greater than the searched argument, FOUND() returns FALSE and EOF() returns FALSE.
- If SOFTSEEK is ON, and there is no key greater than the search argument, the database is positioned at eof = LASTREC() +1, FOUND() returns FALSE and EOF() returns TRUE.

The SET DELETED switch and SET FILTER condition are considered. The current state of SET EXACT does not affect the search; the comparison is done as with SET EXACT OFF.

FIND is identical to SEEK, but has a slightly different syntax: FIND &<var> has the same effect as SEEK <var>, FIND (<var>) is identical to SEEK <var>.

Example:

```
USE employee INDEX idnumber, name
* SET ORDER TO 1                          && key: idnum (numeric 3)
seek_id = 100
find_id = "100"
FIND 100                                  && found
FIND 005                                  && found
FIND 5                                    && found
FIND (STRZERO(10,3))                      && found
FIND find_id                              && not found
FIND &find_id                             && found
SEEK seek_id                              && found
SEEK &seek_id                                   && run-time-error

SET ORDER TO 2                                  && key: UPPER(name)
find_name = upper("Smith")
FIND SMITH                                      && found
FIND Smith                                      && not found
FIND (upper("Smith"))                           && found
SEEK upper("Smith")                             && found
SEEK "SMITH"                                    && found
find_name = upper("Smith")
FIND &find_name                                 && found
SEEK find_name                                  && found
```

Classification:

database

Translation:

DBSEEK (&("key"))

Related:

INDEX, LOCATE, SEEK, SET DELETED, SET EXACT, SET INDEX, SET SOFTSEEK, EOF(), FOUND(), RECNO(), oRdd:SEEK()

FOR ... NEXT

Syntax:

```
FOR <memvar> = <expN1> TO <expN2> [STEP <expN3>]
     <statements>...
[EXIT]
     <statements>...
[LOOP]
     <statements>...
NEXT | ENDFOR
```

Purpose:

A control structure for executing a loop a specified number of times while either incrementing or decrementing a counter expression.

Arguments:

<**memvar**> is the variable that controls the loop. If <memvar> is out of the boundary <expN1>..<expN2>, control is passed to the program statement following the NEXT command.

<**expN1**> is the initial value assigned to <memvar> and the lower (upper) boundary of the looping range.

<**expN2**> is the upper (lower) boundary of the looping range, see also STEP.

NEXT | ENDFOR determines the end of the loop structure. When this command is encountered, the <memvar> is increased (decreased) by <expN3> (or by 1) and the program control is passed to the check boundary against <expN2>. If the check is fulfilled, execution continues with the next statement following the FOR... command.

Options:

STEP <**expN3**> sets the increment value. If not specified, the default value is (plus) one.

Looping stops or is not executed at all when <memvar> is greater than <expN2>. If <expN3> is negative, the <memvar> is reduced and the looping stopped when <memvar> becomes lower than <expN2>.

EXIT: The EXIT statement terminates the looping, branching unconditionally to the statement following the NEXT command. Any number of EXITs within the structure are accepted.

LOOP: The LOOP statement repeats the looping by branching on to complete the increment/decrement and then back to the ...TO <expN2> condition check. Any number of LOOPs within the structure are accepted.

Description:

The FOR...NEXT structure iterates the statements within from an initial value of the control variable to a specified boundary. The control variable sweeps this range of values for a increment specified in the STEP clause. In contrast to some other programming languages, FlagShip evaluates the entire termination and increment condition each time it is encountered. This means that the upper boundary and increment are dynamic - they can be changed as the loop operates.

Hint: Using TYPED variables and/or numeric constants for the control variable, step and the end value increases loop speed significantly, see example LOCAL..AS.

Example:

The FOR...NEXT construct is useful when dealing with arrays.

```
LOCAL_INT i, len
LOCAL array[1000], count := 1
len = LEN(array)

FOR i = 1 TO len   // Runs forward through an entire array
   array[i] := i   // 1..1000
NEXT

* Runs backwards through an entire array
FOR i = len TO 1 STEP -1
   array[i] = count++                  // 1000...1
NEXT
```

Classification:

programming

Compatibility:

The ENDFOR statement is not available in Clipper, but in FoxPro.

Related:

DO WHILE, BEGIN SEQUENCE

FUNCTION

Syntax 1:

```
FUNCTION <udfname> [AS <type>]
[PARAMETERS <paramList>]
     <statements>...
RETURN <exp>
```

Syntax 2:

```
FUNCTION <udfname> (<paramList>) [AS <type>]
     <statements>...
RETURN <exp>
```

Syntax 3:

```
[STATIC|INIT|EXIT] FUNCTION <udfname> (<paramList>)
        [AS <type>]
     <statements>...
RETURN <exp>
```

Purpose:

Declares a user-defined function (UDF) written in the FlagShip language and, optionally, its formal parameters.

Arguments:

<**udfname**> is the declared name of the user-defined function. The function name can be of any length, but only the first 10 characters are significant. Upper or lower case makes no difference. The names can contain any combination of characters A..Z, numbers, or underscores, but names with a leading underscore are reserved for internal FlagShip functions.

RETURN <**exp**> terminates the execution of the UDF and passes control back to the calling program returning the value of <exp> to the program module called. Any number of RETURNs, even when they have different types, may be placed within the UDF. The returned <exp> can be of any type, including array, code block or object. If <exp> is not given or a RETURN command is not encountered, NIL is returned. For typed function, the type of <exp> has to match to the declared function <type>. Where the function returns different types, prototype it AS USUAL.

Options:

STATIC FUNCTION declares a UDF which is visible in the current .prg file only. Several STATIC UDFs and UDPs (and only one public UDP/UDF) may be defined with the same name in different .prg files.

Because the references to a STATIC function are resolved at compile-time, they will hide public UDF carrying the same name. STATIC functions are not generally visible and therefore cannot be used during a macro evaluation or as UDFs for ACHOICE(), MEMOEDIT() etc.

When the keyword STATIC is omitted, the UDF becomes public and the name visible to the entire application.

INIT FUNCTION declares a module, executed at program startup; see description of INIT PROCEDURE.

EXIT FUNCTION declares a module, executed at program termination; see description of EXIT PROCEDURE.

PARAMETERS <**paramList**> specifies one or more comma separated PRIVATE variables which receive the calling arguments. See more in the PARAMETERS command.

(<**paramList**>**)** is an alternative syntax for the PARAMETERS command, but the variables in <paramList> have LOCAL type and may optionally be typed, see below.

AS <**type**> (proto)types the function declaring it to return the specified <type> value only, see below. The specified type has to correspond to the RETURNed value type. If different value types (or NIL) is returned, (proto)type the function AS USUAL. If the AS <type> is omitted, the implicit USUAL type is assumed. Note: only explicitly typed functions are added to the repository file (e.g. reposit.fh) with the -ru compiler switch.

Prototyping of parameters and return value:

The local parameters specified in brackets (according to syntax 2) may optionally be typed (with all usual <type>s according to LOCAL..AS), and/or prototyped as optional. The syntax is equivalent to **(**<paramList>**)** of the PROTOTYPE declarator, e.g.

```
FUNCT myUdf (p1 AS CHAR, [p2 as NUMER], p3, [p4]) AS LOGIC
```

If the <type> is not given (e.g. parameters p3 and p4 in this example), AS USUAL is assumed. The parameter name enclosed in square brackets `[ ]` (visually) signals an optional parameter, used also in (and passed to) UDF prototypes. It does not change the behavior of parameter passing, nor the parameter order in any way.

Also, the return function <type> may be prototyped by using the syntax 1 or 2.

Purpose of the prototyping:

Declaring a type of the return value allows to check the RETURN statement of a UDF and its usage (e.g. in assignments) already at compiler time. Giving the parameters a type allows a compile-time check of the parameters (arguments) passed to the function at places where it is invoked. Both of these compile-time checks will help you to avoid unexpected RTEs (run-time errors) and simplify parameter validation in the function body. See also "parameter passing" below.

Use the PROTOTYPE declarator (e.g. in an #include file), when the UDF is invoked in other than the current file (prototyping); or when the UDF is specified in the same file, but is invoked before its declaration (forward prototyping) to take advantage of the compile-time checking.

Note: the PROTOTYPE statement is automatically created in the repository file (for typed UDFs only) by using the -ru compiler switch, see FSC.1.3. All standard FlagShip functions are prototyped in the stdfunct.fh file.

Description:

Functions and procedures increase both readability and modularity, and standardize a block of frequently used statements.

A user-defined function (UDF) is called using the same notation as when calling a standard FlagShip function:

```
  [value :=]  udfname (parameters)
```

The UDF may be called within an expression or on a line by itself, ignoring the return value.

A user-defined function may also be called as an aliased expression by preceding it with an alias and enclosing it in parentheses, like:

```
  [value :=]  alias->(udfname (parameters))
  [value :=]  ("xyz"+var)->(udfname (parameters))
  [value :=]  (SELECT()+1)->(udfname (parameters))
```

Functions called in this way will select the associated working area prior to execution and re-select the original one on return.

Assigning the UDF return value to a typed variable is checked at compile-time and/or run-time. If the function type is known at compile-time (see prototyping), an incorrect assignment is already reported by the FlagShip compiler. Otherwise, if the declared function type does not match the fixed variable type in the return statement, a run-time error occurs. Of course, a possible numeric conversion (e.g. AS NUMER to INTVAR etc.) is accepted and performed automatically.

A UDF may call itself **recursively**. The number of recursions in FlagShip is limited only by the available RAM + swap disk space to store the local data of each recursion.

Parameter passing:

The calling arguments are passed to a user-defined function by value by default, except for array names and objects, which are always passed by reference. Variables other than field variables preceded by the @ operator are passed by reference.

The UDF receives the passed arguments into predefined PRIVATE or LOCAL variables in the <paramList>. The number of arguments passed and parameters received does not need to match. Arguments may be skipped or left off the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. Refer to LNG.2.3.2 and (CMD) PARAMETERS for a more detailed discussion.

On typed parameters, only arguments of the specified parameter type are accepted. If the prototype of the UDF is known at compile time (see prototyping), an incorrect argument passing is reported by the FlagShip compiler. If the prototype or the argument type is unknown at compile time, and an incorrect argument type is passed, a run-time error occurs. On optional parameters (i.e. enclosed in square brackets), only the specified type or NIL is accepted.

UDF vs. UDP

In FlagShip, the only difference between the call to a function (UDF) or procedure (UDP) is the convention of default parameter passing. Both UDFs and UDPs may be used interchangeably. Hence, if a function (UDF) is called using the procedure's DO...WITH invocation, the parameters are passed per default by reference, instead of by value as with a standard UDF call.

Example 1:

The example centers a string using a user-defined function:

```
center (20, "user message")
text = "Hello!"
@ 30, cent_col(text) SAY text
FUNCTION cent_col (string)
RETURN INT((MAX_COL() - LEN(string)) /2)

FUNCTION center
PARAMETERS row, string
@ row, cent_col(string) SAY string
RETURN NIL
```

Example 2:

Usage of typed parameters and typed function. The first parameter is optional:

```
PROTO FUNCT centOut ([par1 AS nume], par2 AS char) AS NUMER
LOCAL xx AS logical, yy AS numeric
LOCAL tt := "Hello world!" as character

centOut (5, "Text in line 5")             // ok
devpos(10,0)
centOut ( , "centered text at line 10")   // ok
yy := centOut (NIL, tt)                   // ok
xx := centOut (NIL, tt)                   // compiler error
centOut (6, xx)                           // compiler error

FUNCTION centOut ([row AS numer], string AS char) AS NUMER
LOCAL col := INT((MAXCOL() - LEN(string)) /2) // [as numer]
row := min (if (row == NIL, row(), abs(row)), maxrow())
@ row, col SAY string
RETURN col
```

Example 3:

In the following example a variable is passed to a user-defined function by value and then by reference. Note that the second case changes the original variable as well.

```
value = 10
? change(value),  value                     // 20  10
? change(@value), value                     // 20  20

STATIC FUNCTION change (par)
par *= 2                                    // par = par * 2
RETURN par
```

Example 4:

The next example demonstrates how to validate a data entry using a user-defined function:

```
x = 0
@ 1,0 SAY "Enter number: " GET x VALID checkit(x, 10, 20)
READ
RETURN

FUNCTION checkit (numb, toolow, toohigh)
RETURN (numb > toolow .AND. numb < toohigh)
```

Example 5:

Usage of aliases:

```
USE address NEW ALIAS addr                  // field adr_no
? TYPE("adr_no"), TYPE("cust_no")           // N  U
USE custom NEW INDEX custno                 // field cust_no
? TYPE("adr_no"), TYPE("cust_no")           // U  N
? addr->(TYPE("adr_no")), TYPE("cust_no")  // N  N

SELECT addr
IF custom->(my_replace (adr_no, 55))
   ? "replacing o.k."
ENDIF
? ALIAS ()                                  // addr

FUNCTION my_replace (number, value)
? ALIAS()                                   // custom
SEEK number                                 // search for adr_no
IF FOUND()                                  //   in cust_no
   REPLACE cust_no WITH value
   RETURN .T.
ENDIF
RETURN .F.
```

Classification:

programming

Compatibility:

The STATIC, INIT and EXIT clause and the use of formal LOCAL parameters is compatible to C5. FlagShip accepts returning from a UDF by RETURN or when the end-of-file or next UDF declaration is reached, NIL is returned. In Clipper, the <exp> value must be specified.

Typed parameters and typed functions are supported by FlagShip and VO. The definition of optional parameters by using square brackets is available in FlagShip only.

Related:

PROCEDURE, PARAMETERS, PROTOTYPE, RETURN, SELECT

GLOBAL ... AS

Syntax 1:

```
GLOBAL <tvarList> [:= <constN>] AS <C-type>
```

Syntax 2:

```
GLOBAL_<C-type> <tvarList> [:= <expN>]
```

Purpose:

Declares and initializes C-TYPED GLOBAL variables.

Arguments:

<**tvarList**> is a comma separated list specifying the names of variables, to be declared as TYPED GLOBAL. The same naming convention (10 significant characters, no case dependence, conversion to lower case) is valid as for the other typed variables. See LOCAL..AS.

AS <**C-type**> is the alternate syntax to GLOBAL_<type> where <C- type> is one of the C-like type keywords listed in LOCAL...AS.

Example of valid syntax:

```
   GLOBAL iVar := 4, ipos := 0, iCount AS INT
   CLOBAL_LONG iOther := 5, myCount
```

Options:

<**constN**> is a numeric constant within the <type> range to initialize the variable at program start. If not given, the TYPED GLOBAL variables will be initialized with zero.

Scope, Visibility:

The TYPED GLOBAL variables may be described also as "STATICs with an application-wide visibility". They are very similar to global C variables and have a lifetime of the entire program. They are visible within the entity that defines them; other entities have to enable the visibility, if needed, using the GLOBAL_EXTERN declaration.

- **UDF wide scope**: if the declaration is given within the procedure or function body, the variables are visible in this module only; all other modules may enable the visibility using the GLOBAL_EXTERN declaration.
- **File-wide scope**: if the declaration is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the variable is visible for all UDFs or UDPs within these .prg files. Modules in all other .prg files can enable the visibility using GLOBAL_EXTERN.

Description:

Using TYPED GLOBAL variables is identical to other typed variables, like LOCAL..AS. They may be used directly in any expression, command or #Cinline program part.

Like LOCAL and STATIC, typed GLOBAL variables are invisible within a macro evaluation and will hide PRIVATE or PUBLIC variables having the same name. The TYPED variables will always be passed to UDF and UDP by value, regardless of the calling convention used (@ prefix or using the DO...WITH procedure call).

Normally, only one GLOBAL...AS variable of the same name is allowed for the whole application. Some linkers accept a multiple declaration, but do not define which initializing value is used.

Example 1:

Using typed variables:

```
*** file test1.prg, calls --> test2.prg ***
#ifndef FlagShip                       // Clipper compatib.
   #define LOCAL_INT           LOCAL
   #define GLOBAL_EXTERN_LONG  MEMVAR
#endif
LOCAL_INT          aa, bb := 15
GLOBAL_EXTERN_LONG gg                  // enable visibility

? VALTYPE(aa), VALTYPE(gg)                        // N N
? aa, bb, gg                                      // 0 15 1
aa = 3
my_funct (aa, bb)
? aa, bb, gg                                      // 3 15 2
bb *= 2
gg := 5
my_funct (3, 0)
? aa, bb, gg                                      // 3 30 6
*** file test2.prg ***
#ifndef FlagShip                                  // Clipper ?
   #define LOCAL_LONG   LOCAL
   #define GLOBAL_LONG  PUBLIC
#endif

function my_funct (p1, p2)
LOCAL_LONG  aa, bb
GLOBAL_LONG gg := 1                               // declare it
? VALTYPE(aa), VALTYPE(gg), aa, bb, gg            // I I 0 0 1
? VALTYPE(p1), VALTYPE(p2), p1, p2                // N N 3 0
p1 := p2 := aa := 5
gg++
? aa, bb, gg, p1, p2                              // 5 0 2 5 5
RETURN NIL
```

Example 2:

see more examples in GLOBAL_EXTERN, LOCAL...AS, STATIC...AS, CALL, #Cinline.

Classification:

programming

Compatibility:

Typed variables are available in FlagShip only. To be compatible to Clipper 5, use PUBLICs:

```
#ifndef FlagShip
# define GLOBAL_INT PUBLIC
# define EXTERN_INT MEMVAR
#endif
```

Related:

GLOBAL_EXTERN, LOCAL, LOCAL...AS, STATIC, STATIC...AS, PRIVATE, PUBLIC, CALL, FIELDS, DO, FUNCTION, TYPE(), VALTYPE(), #define, #ifdef, #Cinline

GLOBAL_EXTERN ... AS

Syntax 1:

`GLOBAL_EXTERN <tvarList> AS <C-type>`

Syntax 2:

`GLOBAL_EXTERN_<C-type> <tvarList>`

Purpose:

Enables access to a C-TYPED GLOBAL variable from other program modules.

Arguments:

<**tvarList**> is a comma separated list specifying the names of variables, declared in other procedure or .prg file as TYPED GLOBAL, see GLOBAL...AS.

AS <**C-type**> is the alternate syntax to EXTERN_GLOBAL_<type> where <type> is one of the C-like type keywords listed in LOCAL...AS.

Example of valid syntax:

```
   GLOBAL_EXTERN_INT  iVar, ipos, iCount AS INT
   CLOBAL_EXTERN_LONG iOther, myCount
```

The variable's <**C-type**> specifies the storage range, which **must be** identical with the one declared by GLOBAL...AS.

Scope, Visibility:

The GLOBAL_EXTERN scope is similar to that of other typed or local/static variables:

- UDF wide scope: if the declaration is given within the procedure or function body, the variable visibility is enabled in this module only.
- File-wide scope: if the declaration is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the variable is visible for all UDFs or UDPs within these .prg files.

Description:

The GLOBAL_EXTERN enables the visibility to a GLOBAL...AS variable of the same name, declared elsewhere in the application.

The <C-type> of GLOBAL_EXTERN will be not checked against the GLOBAL...AS declaration; if a different <C-type> is used, unpredictable results will occur. If the variable is not declared at all, the linker error "unresolved external" occurs.

When the visibility is enabled, the GLOBAL variable can be modified by the current procedure (or by all UDF/ UDPs of the .prg file, see scope).

Example:

The variable "byte" may be used/accessed also in other .prg files using GLOBAL_EXTERN. No naming conflicts occurs with other variable types, if GLOBAL_EXTERN is not used.

```
FUNCTION my_udf1 (par1)
GLOBAL_BYTE byte
? byte                                  // 0
my_udf2 ()
? byte                                  // 2
my_udf3 ()
? byte                                  // 2
my_udf2 ()
? byte                                  // 4
RETURN NIL

FUNCTION my_udf2
EXTERN_GLOBAL_BYTE byte
byte += 2
RETURN

FUNCTION my_udf3
LOCAL byte
byte := 5
RETURN
```

Classification:

programming

Compatibility:

Typed variables are available in FlagShip only. To be compatible to Clipper 5, use PUBLICs:

```
#ifndef FlagShip
# define GLOBAL_INT        PUBLIC
# define GLOBAL_EXTERN_INT MEMVAR
#endif
```

Related:

GLOBAL, LOCAL, STATIC, PRIVATE, PUBLIC, FIELDS, DO, FUNCTION, TYPE(), VALTYPE(), #define, #ifdef, #Cinline

GO | GOTO

Syntax:

```
GO <expN>|TOP|BOTTOM
```

or:

```
GOTO <expN>|TOP|BOTTOM
```

or:

```
GO TO <expN>|TOP|BOTTOM
```

Purpose:

Moves the record pointer to a specific record in the current working area.

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 also be accessed.

TOP: GOTO TOP 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, GOTO TOP moves to the first record of the scope, as in the command LOCATE. At the time of opening a database and/or index by USE, USE...INDEX and SET INDEX TO... or associated functions, GO TOP is executed automatically when SET GOTOP is ON. The default setting is OFF to enable programmable integrity check.

BOTTOM: GOTO BOTTOM moves to the last record of the controlling index, if there is one, or to LASTREC(), if there is no index in use. If there is a filter scope, GOTO BOTTOM moves to the last record of the scope.

Description:

GO and the synonym GOTO are database commands which position the record pointer in the current working area to a specified physical record or at the (logical) top or bottom of the file. Hint: in most cases it is easier to search in your sources for GOTO instead of GO when maintained later.

Using the TOP or BOTTOM criteria also obeys the SET FILTER and SET DELETED condition. This may be time consuming on large database, because the fulfilling criteria has to be looked for, by skipping forward or backwards from the first/last record.

Using GOTO <expN> is a fast database access. If the required record number <expN> is out of range, no run time error is generated, but the database pointer is positioned to LASTREC()+1, EOF() and BOF() are both set to TRUE.

Multiuser:

In multiuser environment, the internal and Unix/Windows buffers can also be refreshed using GOTO RECNO() (or SKIP 0, COMMIT). See more in section LNG.4.8. The GOTO may perform flushing of changed record to database, modifiable by SET COMMIT.

Example:

```
USE employee
? RECCOUNT()                                && 100
GO 1 + 2 * 3
? RECNO()                                   && 7
GO BOTTOM
? RECNO()                                   && 100
SET INDEX TO Name
GO BOTTOM
? RECNO()                                   && 17
? Lastname                                  && Smith
GO TOP
? RECNO()                                   && 59
? Lastname                                  && Aaron
```

Classification:

database

Translation:

DBGOTO (expN)
DBGOTOP ()
DBGOBOTTOM ()

Related:

SET GOTOP, SKIP, LOCATE, COMMIT, LASTREC(), RECNO(), SET COMMIT, oRdd:GOTO(), LNG.4.8

HIDDEN INSTANCE

Syntax 1:

```
      [STATIC] CLASS <ClassName> [INHERIT <SuperClass>]
```

and optional:

```
      INSTANCE <Name> [:= <exp>] [AS <type>]
      EXPORT [INSTANCE] <Name> ...
      HIDDEN [INSTANCE] <Name> ...
      PROTECT [INSTANCE] <Name> ...
```

Syntax 2:

```
      PROTOTYPE [STATIC] CLASS <ClassName>
              [INHERIT <SuperClass>]
```

and optional:

```
      INSTANCE <Name> [AS <type>]
      EXPORT|HIDDEN|PROTECT [INSTANCE] <Name> [AS <type>]
```

See detailed description in the CLASS command.

IF ... ENDIF

Syntax:

```
IF <condition>
     <statements>...
[ELSEIF <condition>]
     <statements>...
[ELSE]
     <statements>...
ENDIF | END
```

Purpose:

A control structure to conditionally execute a block of commands.

Arguments:

<**condition**> is the control expression. If <condition> evaluates to true (.T.) all the commands following are executed until an ELSEIF, ELSE or ENDIF is encountered. Otherwise, the control is passed to the next ELSEIF condition if given, or the first command following the ELSE statement. If there is no ELSE statement, the control is passed to the next program statement following the ENDIF.

ENDIF may be shortened to END.

Options:

ELSEIF <**condition**>**:** The ELSEIF clause will be evaluated if the previous IF and/or ELSEIF conditions are returned false. If the <condition> evaluates to true (.T.), the following commands are executed until an ELSEIF, ELSE or ENDIF is encountered. The control structure may contain any number of ELSEIF clauses.

ELSE: The ELSE clause is used to identify the commands that are to be executed if neither the previous control expression <condition> was successful.

Description:

IF...ENDIF structures may be nested with other structured programming commands, and also within other IF...ENDIF structures. This structure is equivalent to the DO CASE... CASE... END sequence.

Example:

```
* the structure:                  * is equivalent to:
DO CASE                      |
   CASE value < 10           |    IF value < 10
      ? "up to 10"           |       ? "up to 10"
   CASE value <= 100         |    ELSEIF value <= 100
      ? "10 to 100"          |       ? "10 to 100"
   CASE value <= 1000        |    ELSEIF value <= 1000
      ? "101 to 1000"        |       ? "101 to 1000"
   OTHERWISE                 |    ELSE
      ? "1000 and up"        |        ? "1000 and up"
ENDCASE                      |    ENDIF
```

```
IF value < 10                       |    IF value < 10
   ? "up to 10"                     |       ? "up to 10"
                                    |    ELSE
ELSEIF value <= 100                 |       IF value <= 100
   ? "10 to 100"                    |          ? "10 to 100"
                                    |       ELSE
ELSEIF value <= 1000                |          IF value <= 1000
   ? "101 to 1000"                  |             ? "101 to 1000"
ELSE                                |          ELSE
   ? "1000 and up"                  |             ? "1000 and up"
ENDIF                               |          ENDIF
                                    |       ENDIF
                                    |    ENDIF
```

Classification:

programming

Related:

DO CASE, IF(), IIF()

INDEX ON...TO

Syntax 1:

```
INDEX ON <exp> TO <file>|(<expC1>)
        [UNIQUE]
        [NOLOCK]
```

Syntax 2:

```
INDEX ON <exp> TO <file>|(<expC1>)
        [TAG <tagName>]
        [NOLOCK]
        [FOR <condition> [<scope>]
           [WHILE <condition>] ]
        [EVAL <expL2> [EVERY <expN3>]]
        [UNIQUE]
        [ASCENDING | DESCENDING]
```

Purpose:

Creates a file that contains an index to records in the current database file.

Arguments:

<**exp**> is an expression that returns, for each record in the current database file, the key value to be placed in the index. <exp> can be character, numeric, date, or logical type.

In the default DBFIDX driver, the maximum length of the given <**exp**> string, stored in the index header, is 420 bytes, of the evaluated expression (index key) up to 238 bytes.

<**file**> is the name of the index file to be created. The default file name extension with the default DBFIDX driver is .idx if none is specified.

Options:

UNIQUE specifies that index <file> includes only unique key values. The result is identical to SET UNIQUE ON, but the UNIQUE clause has precedence over the SET switch.

NOLOCK avoids check for the FLOCK() or EXCLUSIVE open, being in SHARED mode. See "Multiuser" below.

TAG <**tagName**> is the name of the order to be created in the <file> bag. Supported only by RDD drivers with multiple order capability, ignored by RDDs with single-order bags (like the default DBFIDX).

FOR <**condition**> creates an index including only a subset of records met by the <condition>. If the clause is not specified, the index file includes all records of the database. The <condition> is stored in the .idx header and therefore is considered when updating or REINDEXing the index file. The maximum length of the FOR <condition> is 198 bytes.

WHILE <**condition**> specifies an additional index filter. Applied during indexing, and together with the FOR clause only; not used for other index operations.

<**scope**> is the part of the current database file to be indexed. Applied during indexing, and together with the FOR clause only; not used for other index operations. The default scope is ALL records.

EVAL <**expL2**> is similar to the WHILE <condition> but it may be executed at a specific record interval given by the **EVERY** <**expN3**> clause. The <expL2> must return TRUE to continue the indexing. The EVAL clause may be used, for example to monitor the progress of the indexing, using an UDF. If <expN3> is not specified, the default value is one (each record).

ASCENDING | DESCENDING specifies that the index keys are sorted in increasing or decreasing order. The default is ASCENDING.

Note: the INDEX ON command stores the following data in the header of the .idx file (the sizes may vary with other RDDs):

- the <exp> string as given (max. 430 bytes),
- the FOR <condition> string (max. 230 bytes),
- the UNIQUE (or current SET UNIQUE) status,
- the ASCEND/DESCEND status.

The REINDEX command takes all these stored parameters into consideration. All other arguments, like the scope, WHILE, EVAL, EVERY clause are used only during the INDEX ON process. Additionally the following data is also stored in the .idx file header (with the DBFIDX driver):

- the name of the active database, and
- the current update counter of the .dbf to synchronize integrity checking (see LNG.4.5)

Description:

If records are required to appear in a specific order in the database file they could be SORTed. However, this would cause physical reordering, which is rather time consuming. If the application later requires ordering according to some other criterion, it would be very inefficient.

To solve this problem, index files were designed. The command INDEX ON will build a file consisting of values of key expressions evaluated on the records of the database and pointers to their physical location in the file. In this manner, manipulation is much quicker and as many different index files as needed can be built. Changing the database contents will automatically update all assigned indices, so re-indexing is not necessary at all.

When INDEX ON is invoked, all open index files in the current working area are closed and the new index file is created. After the command has been 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 SET INDEX, USE or CLOSE releases it for sharing.

Indexing may be aborted by invoking oRdd:Abort() or by returning .F. value from optional codeblock(s) supplied via oRdd:BlockStart() or oRdd:BlockDone() or oRdd:BlockEval() = DbObject():Block*()

Index key size:

All index keys within the same index file (or tag) **must always** have the same size. If TRIM() is used, the key must be extended with spaces to resolve the default key length:

```
INDEX ON TRIM(name) + STR(zip,6) + ;
      SPACE (LEN(name) - LEN(TRIM(name)))  TO ...
INDEX ON PADR(TRIM(name) + STR(zip,6), 50) TO ...
```

Sorting order:

INDEX orders character keys according to the ASCII value of each character within the string, numeric values in numeric order, date values chronological order with blank dates treated as low values, and logical values sorted with the order FALSE...TRUE. Memo fields cannot be INDEXed.

To create **descending** order indices use the DESCEND() function or the DESCENDING clause. The 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. Using the DESCENDING clause, use SEEK without the **DESCEND**() function.

Deleted records:

Deleted and filtered records are **included** in the index until PACK is executed. To omit them, the `FOR !DELETED()` clause can be used.

Using variables:

LOCAL, STATIC and typed variables cannot be used in index key expressions, because the stored <exp> string is later evaluated as macro to produce the current key values. For the same reason, the compile-time declarations using MEMVAR or FIELD, are not valid within an index key expression; but explicit aliasing may be used instead, if required.

With a numeric key **expression** (i.e. not a simple database field but a calculation or an UDF call) returning floating number, a RTE (run-time error) 311 may occur during adding/replacing the database record because of possible numeric inaccuracy. To avoid this, you should use INT(expr) or ROUND(expr,n) for the index key. Otherwise FlagShip changes the index key automatically during INDEX ON ... to fixed decimals by ROUND(expr,deci), where <expr> is the supplied expression and <deci> is the current SET DECIMAL value. This manual adjustment of the decimal precision is not required nor is done automatically when indexing a simple numeric **field** (i.e. with the usual INDEX ON operation), since the decimals are always fix in the database.

Unique and conditional indexes:

If the clause UNIQUE is used, or SET UNIQUE is set to ON during INDEX ON creation, only the first occurrence of a key will be stored in the index file. Subsequent REPLACE, PACK and REINDEX commands do not add a new key, if the same one is already available. Since the unique status is stored in the index file header, SET UNIQUE only takes effect during the index creation using the INDEX ON command.

Using the FOR, WHILE or EVAL clause may create an empty index. If such an index is used or selected thereafter, both BOF() and EOF() return TRUE and the record pointer is set beyond the end of database file (LASTREC()+1).

Hint: The usage of a filtered index using the FOR, WHILE or EVAL clause is similar to SET FILTER TO... but may speed-up the searching significantly compared to e.g. LOCATE, SET FILTER ... GOTO TOP etc. especially on large database. Also, the usage of filtered index in BROWSE(), DBEVAL() or Tbrowse is almost much faster then SET FILTER.

Multiuser:

During the INDEX ON, the required database must be exclusively opened using USE...EXCLUSIVE or SET EXCLUSIVE ON. If the index file is not being used by other users, 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 NOLOCK clause.

Example 1:

Creates /usr/data1/pers_titl.idx and /usr/data2/pers_bd.idx

```
#ifdef FlagShip
   FS_SET ("lower", .T.)
   FS_SET ("pathlower", .T.)
   FS_SET ("translext", "ntx", "idx")
#endif

ind1 = "\usr\Data1/pers_titl"
SET DEFAULT TO "\usr\Data2"
USE Adres NEW
if !FILE("Pers_BD.ntx") .or. !FILE(ind1+INDEXEXT())
   INDEX ON birth_date TO Pers_bd
   INDEX ON title + DESCEND(DTOS(birth_date)) TO &ind1
endif
SET INDEX TO Pers_bd, &ind1
```

Example 2:

The same example for multiuser/multitasking:

```
SET EXCLUSIVE OFF
ind1 = "/usr/data1/pers_titl"
SET DEFAULT TO "/usr/data2"
IF !FILE("Pers_bd.idx") .or. !FILE(ind1+INDEXEXT())
   USE adres EXCLUSIVE NEW              && open dbf exclusive
   DO WHILE NETERR()                    && if no success:
      INKEY(3)                          && wait and
      USE adres EXCLUSIVE               && try again
   ENDDO
   INDEX ON birth_date TO pers_bd
   INDEX ON title + DESCEND(DTOS(birth_date)) TO &ind1
   USE                                  && close from exclusive
ENDIF

USE adres                               && open dbf shareable
DO WHILE NETERR()                       && if no success:
   USE adres                            && try again
ENDDO
SET INDEX TO pers_bd, &ind1
```

Example 3:

Report the percentage of the index processed. See other examples in DBCREATEINDEX().

```
LOCAL count, perc := 0
USE address NEW EXCLUSIVE
count := LASTREC()
INDEX ON UPPER(name) + STR(zipcode,6) TO namezip ;
      EVAL mydisplay(perc++) EVERY INT(count/100) ASCENDING

FUNCTION mydisplay (out)
@ 20,10 say "Indexing, " + STR(out,3) + "% ready"
RETURN .T.
```

Example 4:

Filtered database, similar to SET FILTER TO but the access is much faster, specially on large database. Of course, the special index may be build once only and assigned thereafter with SET INDEX TO ... SET ORDER.

```
USE address NEW SHARED
IF !file("special" + indexext())
   WHILE !FLOCK(); END                  // at least Flock required
   INDEX ON UPPER(name) + zip ;
         FOR TRIM(country)=="D" TO special
   UNLOCK                               // free lock
ENDIF
SET INDEX TO name, special              // two indices used
SET ORDER TO 2
GOTO TOP                                // only indexed records
DBEDIT (1,1, maxrow()-1,78)             // are visible now
SET ORDER TO 1                          // all records are visible

- which is similar to -

USE address NEW SHARED
SET INDEX TO name
SET FILTER TO TRIM(country) == "D"      // the filter is slower
GOTO TOP                                // only filtered records
DBEDIT (1,1, maxrow()-1,78)             // are visible now
SET FILTER TO                           // all records visible
```

Example 5:

Check database/index integrity:

```
USE address INDEX name EXCLUSIVE
? "Integrity: ", INDEXCHECK()           // .T.
REPLACE name WITH "nobody"
SET INDEX TO name, zipcodes             // index integrity unknown
? INDEXCHECK(1), INDEXCHECK(2)          // .T.  .F.

FS_SET ("develop", .T.)                 // set "developer" mode
SET ORDER TO 2
SKIP                                    // RTE warning occurs

FOR ii = 1 TO INDEXCOUNT()              // rebuild indices
   IF .not. INDEXCHECK(ii)              // when integrity violated
      REINDEX
      EXIT
   ENDIF
NEXT
```

Classification:

database

Compatibility:

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.

Programs ported from DOS or other Unix systems with different hardware have to create new indices using INDEX ON. FlagShip indices (and databases) are however cross-compatible to different operating systems like Windows, Unix, Linux, Mac etc. To check the index file existence using FILE(), either INDEXEXT() or FS_SET("transl") can be used for portable applications.

The integrity check and the NOLOCK clause is available in FlagShip only.
FS support an unlimited number (65000) of indices for each working area/database

The index structure of FS4.x and FS6 is **not compatible**. On attempt to access the FS4.x index by application compiled by FS4, run-time error message occurs. You will need INDEX ON..TO.. on the first use by FS6. The new index structure is significantly faster, support automatic PC8/ANSI conversion and the key size is increased.

Translation:

Syntax 1: DBCREATEINDEX ("file","exp", {||exp}, .unique., .nolock.)
Syntax 2: ORDCONDSET ("for", {||for}, .all., {||while}, ;
{||eval}, every, RECNO(), next, rec, .rest., .descend.)
ORDCREATE ("file", "tag", "exp", {||exp},
.unique., .nolock.)

Related:

REINDEX, DBCREATEINDEX(), SET INDEX, SET ORDER, SET UNIQUE, USE, SEEK, FIND, SET EXCLUSIVE, CLOSE, DTOS(), INDEXCHECK(), INDEXEXT(), INDEXKEY(), INDEXORD(), INDEXCOUNT(), INDEXNAMES(), INDEXDBF(), NETERR(), FS_SET(), SET AUTOLOCK, AUTOFLOCK(), oRdd:SetOrderCondition(), oRdd:CreateOrder()

INPUT ... TO

Syntax:

`INPUT [<exp>] TO <memvar>`

Purpose:

Waits for an expression to be typed in from the keyboard. The result is placed in a memory variable.

Arguments:

<**memvar**> is the memory variable where the entered user input is stored. If the variable is not visible or does not exist, a new autoPRIVATE variable is created.

Options:

<**exp**> is the optional prompt which is displayed in front of the entry area. It can be an expression of any data type. If not given, no prompt is displayed.

Description:

INPUT is a console command with wait state. First, a NEW LINE and the prompt (or "") is displayed. The user input is evaluated using the macro (&) operator, and the result is stored in <memvar>. The type of expression entered determines the type of memory variable which is set.

The entry from the keyboard is terminated by the ENTER <┘ key. Among special keys, only BACKSPACE is supported. If nothing was entered, the variable is not changed (or created).

If the response should be a character type, it has to be enclosed in single or double quotes; or the alternate commands ACCEPT, WAIT or @...GET are used. Unlike these, INPUT allows a complex expression to be entered using variables, functions etc. If the result of date type is required, the entry must be placed in curly brackets or evaluated by the program using CTOD().

Example:

This function returns .T. if the user wants to quit

```
FUNCTION Quitfun (text)
LOCAL   answ
PRIVATE yes := y := ja := oui := .T.
PRIVATE no := nein := n := .F.
INPUT text + " ? " TO answ
IF VALTYPE(answ) = "L"
   RETURN answ
ENDIF
RETURN .F.
```

Classification:

sequential screen output, waiting keyboard input

Translation:

```
IF ( !EMPTY (__ACCEPT ("exp")) )
   <memvar> := &( __ACCEPTSTR () )
END
```

Related:

@...SAY...GET, ACCEPT, WAIT

INSTANCE

Syntax 1:
```
     [STATIC] CLASS <ClassName> [INHERIT <SuperClass>]
```
and optional:
```
             INSTANCE <Name> [:= <exp>] [AS <type>]
             EXPORT [INSTANCE] <Name> ...
             HIDDEN [INSTANCE] <Name> ...
             PROTECT [INSTANCE] <Name> ...
```

Syntax 2:
```
     PROTOTYPE [STATIC] CLASS <ClassName>
             [INHERIT <SuperClass>]
```
and optional:
```
             INSTANCE <Name> [AS <type>]
             EXPORT|HIDDEN|PROTECT [INSTANCE] <Name>
                [AS <type>]
```

See detailed description in the CLASS command.

JOIN WITH...TO...

Syntax:

```
JOIN WITH <alias>|(<expC1>) TO <file>|(<expC2>)
        FOR <condition>
        [FIELDS <fieldList>]
```

Purpose:

Creates a new database by merging specified records and fields from two open database files.

Arguments:

WITH <**alias**> specifies the file to merge with the database in the current working area to create the <file>.

TO <**file**> is the name of the target database file.

FOR <**condition**> selects only records meeting the <condition>.

Options:

FIELDS <**fieldList**> specifies comma separated fields from both areas, which will be the structure of the new file. Fields not from the primary working area have to be referenced by an alias. Note that, if the primary or secondary areas have relations set to some other working areas, those relations will be taken care of, so fields from other working areas can also be specified.

Description:

For each record in the primary area, JOIN analyzes all the records from the secondary area, and creates a new target record each time the <condition> is fulfilled. This means, he operation will take time and can create a lot of new records: the product of both databases (records_curr * records_alias) if no filter is used. JOIN should therefore be used with care.

Example:

```
USE article
USE authors ALIAS aut NEW
SELECT article
JOIN WITH aut TO artauth FOR name = Authors->name
```

Classification:

database

Translation:

```
__DBJOIN("alias", "file", {"field1",... }, {for} )
```

Related:

APPEND FROM, REPLACE, SET RELATION, oRdd:JOIN()

KEYBOARD

Syntax 1:

```
KEYBOARD <expC> [ADDITIVE]
```

Syntax 2:

```
KEYBOARD <expN> [, <expN>, ...] [ADDITIVE]
```

Purpose:

Places a string into the keyboard buffer.

Arguments:

<**expC**> is the string to place into the keyboard buffer. Any INKEY() valid code can be used, e.g. "Abc" + CHR(13) + CHR(-5). If the character is out of range CHR(1...255), a two byte character code is placed into the keyboard buffer to be removed afterwards by any wait status command.

<**expN**> is the inkey value to place into the keyboard buffer. Any INKEY() valid code can be used, e.g. K_F5, K_ALT_X, K_RBUTTONDOWN, ASC("X"). Note: if you add mouse movement or buttons in keyboard buffer and retrieving it later by Inkey(), it does neither affect the reported Mrow() and Mcol() position, nor the automatic mouse movement or button trapping.

Options:

ADDITIVE: If the clause is specified, the string is added at the end of the keyboard buffer. Otherwise, the current contents of the buffer will be overwritten by the new contents.

Description:

Normally, FlagShip stores all keystrokes typed on the terminal in an internal buffer of variable length, cf. LNG.5.2.1. The characters remain in the keyboard buffer until fetched by a wait state command/function such as ACCEPT, INPUT, READ, WAIT, ACHOICE(), MEMOEDIT(), DBEDIT(), or INKEY().

KEYBOARD can be used to fill the internal buffer by simulating user input. For example, filling the buffer from a DBEDIT() user defined function may cause it to perform more actions at once. It can also be used to position the cursor within a READ, if it is to be positioned at a specific GET which is not the first one.

Example:

Positions the cursor initially at the fifth GET (city):

```
USE authors
GO TOP
@ 1,2 GET adrtype
@ 2,2 GET name
@ 3,2 GET firstname
@ 4,2 GET zip PICTURE "99999"
@ 4,9 GET city

KEYBOARD REPLICATE(CHR(13), 4)
READ
```

Classification:

programming

Compatibility:

The ADDITIVE clause is not supported by Clipper but is the default in dBASE. Clipper allows characters with INKEY() codes between zero and 255 only. Numeric parameters are available in FS only.

Translation:

```
__KEYBOARD ("expC", .add.)
```

Related:

SET TYPEAHEAD, INKEY(), READ, ACCEPT, INPUT, WAIT, CHR(), LASTKEY(), NEXTKEY(), REPLICATE(), FS_SET("zerobyte")

LABEL EDIT

Syntax:

```
LABEL EDIT <file>|(<expC>)
           [SIZES <expA1>]
           [MESSAGES <expA2>]
```

Purpose:

Creates/modifies labels in interactive mode for later use by the LABEL FORM command.

Arguments:

<**file**> is the file which holds the definition of the report. If the file exists, it is modified, otherwise a new report is created. The default extension is `.lbl`.

Options:

SIZES <**expA1**> specifies the array of available sizes. Per default <expA1> is set to:

```
   {{'3.5x15", 16x1', 35,5,1, 1,0,0},        ;
    {'3.5x15", 16x2', 35,5,2, 1,0,0},        ;
    {'3.5x15", 16x3', 35,5,3, 1,0,0},        ;
    {'4x17",   16x1', 40,8,1, 1,0,0},        ;
    {'3.2x11", 12x3', 32,5,3, 1,2,0},        ;
    {'user defined ', 35,5,1, 1,0,0}}
```

MESSAGES <**expA2**> specifies the array of query and error messages. It may be changed for another local human language. Per default <expA2> is set to:

```
{"LABEL F2:size F3:specify F5:fields F10:save ESC:quit", ;
 "CREATE",                                              ;
 "MODIFY",                                              ;
 "Size, remark         :",                              ;
 "Label width  (chars) :",                              ;
 "Label hight  (lines) :",                              ;
 "Label columns        :",                              ;
 "Lines between labels :",                              ;
 "Spaces between cols  :",                              ;
 "Left margin  (chars) :",                              ;
 "(F2) PgUp/PgDn: select size CursUp/CursDn: move ...",  ;
 "Line ",                                               ;
 "(F3) enter field or expression. CursUp/ ...",         ;
 "(F5) PgUp/PgDn, CursUp/CursDn: move, scroll ...",     ;
 "<->",                                                 ;
 "Wrong file name, press any key to return",            ;
 "Check the correct entry: TYPE() not character, ...",   ;
 "No LABEL data specified. Use F3 to specify, ESC ...",  ;
 "F6:next dbf"}
```

Description:

If the <file> does not exists, a new `.lbl` file is created, otherwise the available one is modified.

If one or more databases are open in the current working area, the user may view or alter the field names by pressing the F5/F6 key.

By executing the LABEL EDIT command, a full screen label design form appears:

```
CREATE LABEL persname.lbl F2:size F3:specify F5:fields  F10:save ESC:quit
┌F2----------------------------------------┐ ┌F5:test2.dbf---------┐
│  Size, remark         : 3.2x11", 12x3      <->│ │  NAME        C  25 0 │
│  Label width  (chars) :  32                   │ │  FIRST       C  20 0 │
│  Label high   (lines) :   5                   │ │  ZIP         N   5 0 │
│  Rows                 :   3                   │ │  CITY        C  25 0 │
│  Lines between labels :   1                   │ │  BIRTHDATE   D   8 0 │
│  Spaces between rows  :   2                   │ │  EARNING     N   8 2 │
│  Left margin  (chars) :   0                #3│ │  OK          L   1 0 │
└----------------------------------------┘ │                      │
╓F3══════════════════════════════════════╖ │                      │
║  Line  1: trim(PERSONAL->NAME)+" "+FIRST+if<->║ │                      │
║  Line  2: PERSONAL->TITLE                 <->║ │                      │
║  Line  3: if(!empty(ADDRESS),ADDRESS,"")   <->║ │                      │
║  Line  4: PERSONAL->ZIP + TEST2->CITY      <->║ │                      │
║  Line  5: trim(CITY)+", "+STATE+str(ZIP)   <->║ │                      │
║                                              ║ │                      │
║                                              ║ │                      │
╙══════════════════════════════════════╜ └----------F6:next dbf┘
(F3) enter field or express.  CursUp/CursDn: select,scroll ENTER: confirm
```

Pressing PgUp/PgDn in the F2 window, five pre-defined and one user defined label forms may be selected and/or modified. The first line is used for user comment only, the rest specifies label size. To be compatible to the .lbl format, the label height may be up to 16 lines, all other data may contain data in the range 0/1 to 999.

To specify or modify the data to be printed in each label, the F3 key is pressed. Now, moving the light bar up and down, the label line is selected. Enter <┘ to specify/modify the expression (e.g. field, visible variable, function) in these label lines. Any expression may contain up to 60 characters. The contents of these expressions are evaluated by the LABEL FORM command using a macro operator, so the result should be a character, number, date or logical. When the line command is finished, press <┘ , the expression will be validated. If the TYPE() results in unknown or illegal data, a warning appears; it can be safely ignored, if some databases, variables or functions containing the unknown data are not open or available yet.

Pressing F5 in the active F3 window, an entry of available database fields may be accepted into the current expression entry. If more than one data÷ base is opened, the contents of the next one will be displayed using the F6 key.

To save the entry, press F10 key; to abort the LABEL EDIT command without saving it, press the ESC key.

Example:

Create and prints labels

```
USE test2
USE personal INDEX name NEW
IF !FILE(persname.lbl")
   LABEL EDIT persname                  // create label
ENDIF
SEEK UPPER("smith")
IF FOUND()
   LABEL FORM personal TO PRINT ;
         FOR UPPER(TRIM(name)) = "SMITH" NOCONS
ENDIF
```

Classification:

programming

Compatibility:

The command is available in FlagShip only. To create/modify labels in dBASE III+, use CREATE LABEL, in Clipper the program RL.EXE can be used.

Source:

The file <FlagShip_dir>/system/labedit.prg is user modifiable, e.g. to support other languages or to create context sensitive help.

Translation:

__LABELEDIT ("file")

Related:

LABEL FORM, REPORT EDIT

LABEL FORM

Syntax:

```
LABEL FORM <file1>|(<expC1>)
        [<scope>]
        [FOR <condition>]
        [WHILE <condition>]
        [TO PRINTER]
        [TO FILE <file2>|(<expC2>) [ADDITIVE]]
        [SAMPLE [<expA>]]
        [NOCONSOLE]
```

Purpose:

Displays labels defined in a .lbl file.

Arguments:

<**file1**> is the file which holds the definition of the report. The default extension is .lbl.

Options:

<**scope**> is the part of the current database file to traverse. The default scope is ALL.

<**condition**> The FOR clause specifies that the set of records meeting the condition within the given scope is to be displayed. The WHILE clause stops displaying labels when the first record not meeting the condition is reached.

TO PRINTER echoes output to a printer (spool file).

TO FILE <**file2**> echoes output to the specified file. If extension is not specified, .txt is assumed. The **ADDITIVE** clause stops from truncating <file2> if it exists. See also general command description.

NOCONSOLE suppresses the output to the screen, as when SET CONSOLE OFF was set.

SAMPLE <**expA**>**:** If the SAMPLE clause is given, LABEL FORM displays labels as rows of asterisks, allowing the correct positioning of the printer paper. After each row of samples, the program prompts for more samples using the text of <expA> or the default texts {"Do you want more samples?", "Yy"}, if <expA> is not given. When no more samples are requested, the printing of normal labels starts. For direct printer output, use SET PRINTER TO <device>.

Description:

LABEL FORM displays the labels using the definitions stored in a .lbl file. The label file is created either by the FlagShip's command LABEL EDIT or by using the output from dBASE command CREATE/MODIFY LABEL or the Clipper's utility RL.EXE.

Example:

Prints labels for given condition

```
USE article INDEX name
SEEK "RISC Machines"
IF FOUND()
   SELECT 2
   USE Authors
   LABEL FORM authors TO PRINT FOR Id = Article->Id NOCONS
ENDIF
```

Classification:

programming

Compatibility:

The ADDITIVE clause and the query text <expC3> in the SAMPLE clause is supported by FlagShip only.

Translation:

```
__LABELFORM ("file1", .print., "file2", .noconsole., ;
            {for}, {while}, next, rec, .rest., ;
            .sample., "expC3")
```

Related:

LABEL EDIT, REPORT FORM, REPORT EDIT

LIST

Syntax:

```
LIST [OFF] [<scope>] <expList>
        [FOR <condition>]
        [WHILE <condition>]
        [TO PRINTER]
        [TO FILE <file>|(<expC>) [ADDITIVE]]
```

Purpose:

Prints the result of one or more expressions for each processed record to the console and/or printer or file.

Arguments:

<**expList**> is the list of values or expressions (e.g. list of file names) to be evaluated and displayed for each record processed.

Options:

<**scope**> is the part of the current database file to LIST. The default scope is ALL.

<**condition**> specifies additional FOR or/and WHILE filtering, see the general command description.

OFF: Suppresses the display of record numbers.

TO PRINTER: echoes output to a printer file. To disable the screen output, use SET CONSOLE OFF.

TO FILE: echoes output (ADDITIVE) to the specified file. See also general command description.

Description:

LIST displays the record number and the results of the <expList> on screen (with optional echo to printer and/or file) in a tabular format, where each column is separated by a space. LIST is similar to DISPLAY, except that its default scope is ALL, rather than NEXT 1.

Deleted records (see the DELETE command) are marked with a star (*). Deleted records are **not** displayed when SET DELETED is ON, and/or when the current index has "`FOR !Deleted()`" condition, and/or when the "`FOR !Deleted()`" clause was specified with the LIST command.

Example:

Esc will interrupt the LIST output:

```
SET FONT "courier",10
SET CENTURY ON
USE employee
LIST Lastname, Firstname, Birthdate FOR INKEY() <> 27

// output:
  1  Maier      John       05/07/1985
* 2  Miller     Paul       12/29/1994
  3  Smith      Dorothy    11/18/1972
```

Classification:

programming

Compatibility:

The ADDITIVE option is available in FlagShip only.

Translation:

```
 __DBLIST (.off., {exp1 [,exp2...]}, .T., {for}, {while},;
         next, <rec>, .rest., .toPrint., "file")
```

Related:

DISPLAY, SET CONSOLE, SET ALTERNATE, SET EXTRA, SET PRINTER, DBEVAL()

LOCAL

Syntax:

`LOCAL <memvar> [:= <exp>] [, ...]`

Purpose:

Declares and optionally initializes LOCAL variables and arrays.

Arguments:

<**memvar**> is the name of a FlagShip variable or array, to be declared in the (lexically scoped) LOCAL class. The name may be of any length, but only the first 10 character are significant (see more LNG.2.6). Variable names in the FlagShip language are not case sensitive.

If the <memvar> is followed by square brackets [ ], an array is created. The number of elements for each array dimension can be specified as `[dim1,dim2, ..,dimN]` or `[dim1][dim2][dimN]`. The maximum number of dimensions and of the elements per dimension in FlagShip is 65535.

Options, Initializing:

<**exp**> is any valid FlagShip expression including a literal (constant) array to initialize the variable. If the initializer (:= <exp>) is not given, the variable (or all array elements) will be set to NIL.

The LOCAL variable will be created and initialized on each entry into the program module (procedure or function).

Scope, Visibility:

The scope, visibility and lifetime of LOCAL variables is always restricted to one function or procedure only. The variables are created and initialized upon entering a UDF or UDP (exactly when reaching the LOCAL statement) and are destroyed when returning from that module. If a procedure or UDF is invoked recursively (calls itself), each recursive activation creates a new set of local variables.

The local variables can be passed by value or by reference to other UDFs or UDPs called at the same level. In code blocks, only LOCAL variables of the module where the block is declared are visible; see LNG.2.3.3.

LOCAL variable declarations hide all inherited PRIVATE, PARAMETERS, PUBLIC or FIELD variables having the same name. If the variable name is already declared in the same module by using another declarator (STATIC, GLOBAL, MEMVAR, FIELD), or by trying to re-declare such a variable from the file-wide scope, a compiler error is generated.

For more information, refer to the section LNG.2.6.

Description:

LOCAL is a declaration statement that declares one or more variables or arrays local to the current procedure or user-defined function. A parameter list, following the

FUNCTION or PROCEDURE declaration, enclosed in parentheses, is treated as a LOCAL declaration.

In FlagShip, the LOCAL declarator may be placed anywhere in the function body; the scope and visibility of the corresponding local for the compiler start from this declaration.

Short notation: if the declarator is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the declaration (and initialization) is placed at the beginning of every module in the .prg file. The scope, visibility and lifetime is equivalent to explicitly placed LOCAL declarations in each one of these entities.

The variable names are known at compile-time only. Therefore, a LOCAL variable can be evaluated by simple macros, but it cannot be used as a composed macro or **within** the macro string; see also LNG.2.10. Local variables cannot be SAVEd and RESTOREd from .mem files, nor released by CLEAR or RELEASE.

To determine the type of a LOCAL variable, only the standard function VALTYPE(varname) can be used; since the TYPE("varname") tries to evaluate the string using a macro and the variable is invisible during string evaluation.

Example 1:

Declaration and initializing of LOCAL variables:

```
LOCAL var1 := 1, var2 := "xyz", var3 := date()
LOCAL arr1 := {}                       // creates   arr1[0]
LOCAL arr2 := {0,date(),"test",.T.}    // creates   arr2[4]
LOCAL arr3 := {{1,2},{3,4}}            // creates arr3[2,2]
LOCAL arr4[3,2], arr5[0]               // creates arr4[3,2]
LOCAL arr6 := {NIL, arr4, NIL}         // non-symmetr. [3]
```

Example 2:

Parameter passing to UDF (invoke it e.g.: a.out xxx):

```
LOCAL a1, a2 := 0
PARAMETERS cmd1, cmd2
? a1, a2, cmd1, cmd2                    && NIL 0 xxx NIL
start (cmd1, cmd2, a1, @a2)
? a1, a2, cmd1, cmd2                    && NIL 5 xxx NIL
quit

FUNCTION start (p1, p2, p3, p4)
LOCAL xyz
? p1, p2, p3, p4                        && NIL 0 xxx NIL
p4 = 5
xyz = p4 * 10
RETURN NIL
```

Classification:

programming

Compatibility:

The lexical scope is new in FS4, and is compatible to Clipper 5.x. Clipper has a fixed order of the declaration and does not support the short notation (declaration on start of .prg).

Related:

LOCAL..AS, STATIC, GLOBAL, PRIVATE, PUBLIC, FIELDS, DO, FUNCTION, TYPE(), VALTYPE()

LOCAL ... AS

Syntax 1:

```
LOCAL <tvarList> [:= <exp>] AS <type>
```

Syntax 2:

```
LOCAL_<C-type> <tvarList> [:= <expN>]
```

Purpose:

Declares and initializes TYPED LOCAL variables.

Arguments:

<**tvarList**> is a comma separated list specifying the names of variables, to be declared as TYPED LOCAL. The name may be of any length, but only the first 10 characters are significant (see more LNG.2.6). The variable names in the FlagShip language are not case sensitive; when accessing them from the #Cinline statements, use lowercase.

AS <**type**> is an alternate syntax for LOCAL_<type> where <type> is one of the keywords (all of them may be abbreviated to four characters, except objects)

Local...AS <C-type>

C-like <type>	Description
BYTE	one byte char or unsigned num in the range 0..255
DOUBLE	double floating point, in the range +/- 4.94*10-324 ... 1.79*10308 with at least 15 significant digits.
DWORD	unsigned long integer, in the range 0 ... 4 294 967 295.
FLOAT	floating point in the range +/- 1.40*10^-45... 3.40*10^38 with at least 7 significant digits
INT	signed integer, in the range +/- 2 147 483 647 in Unix and Windows32 (or +/- 32 767 in DOS)
LONG	signed long in the range +/- 2 147 483 647 for 32bit OS or signed long in the range +/- 9 223 372 036 854 775 807 for 64bit operating systems
REAL4	equivalent to FLOAT
REAL8	equivalent to DOUBLE
SHORT	signed short integer, in the range +/- 32 767
WORD	unsigned short integer, in the range 0 ... 65 535

The above C-like types do not create an overhead and are therefore much faster than usual variables, see additional description below.

The valid syntax is e.g.

```
LOCAL rVar := 1.0, rVar2, rVar3 := 5.5 AS FLOAT
LOCAL_LONG iVar := 1, iVar2, iVar3 := 5
```

Local...AS <usual type>

Usual <type>	The variable contains:
ARRAY	single or multidimensional array
CHARACTER	string (may include binary 0)
CODEBLOCK	address of a code block
DATE	date values in the range 1/1/1 to 12/31/9999
INTVAR	long integer numbers in the LONG range.
LOGICAL	logical true/false status
NUMERIC	floating point numbers if the DOUBLE range.
OBJECT	any object variable. It does not specify the object and does therefore not force the compile-time address resolution.
PSZ	same as CHARACTER
SCREEN	screen contents from SAVESCREEN()
SPECIAL	user defined, eg. pointer to a C structure
STRING	same as CHARACTER
USUAL	any usual variable type of this table

The above usual variable types allow to check assignments already at compile-time (except for the USUAL type), instead of causing run-time error later.

The valid syntax is e.g.
```
   LOCAL getList := {}, aVar := {1,2,{3,4}} AS ARRAY
   LOCAL iVar1 := 1, iVar2, iVar3 := 5.2 AS NUMERIC
```

Local...AS <object type>

Object <type>	Description, the variable contains
GET	Object variable of the Get class
TBROWSE	Object variable of the TBrowse class
TBCOLUMN	Object variable of the TBcolumn class
ERROR	Object variable of the Error class
DATASERVER	Object variable of the DataServer class
DBSERVER	Object variable of the DBserver or DbfIdx class
<UserClass	> Object variable of the user defined class

The above object types (along with class declaration or prototyping) allow address resolving already at compile-time, which speeds-up execution significantly.

The valid syntax is e.g.
```
   LOCAL getElem AS GET
   LOCAL oMyBrow AS TBROWSE
   LOCAL oDbf1, oDbf2 AS DBFIDX
```

Options, Initializing:

<**exp**> is any valid expression returning a value of the same <type> (or a number for C-like types) to initialize the variable at declaration time.

If the initializer (:= <exp>) is not given, the TYPED LOCAL C-like variables will be initialized with zero, all others to NIL or the empty type, respectively.

The TYPED LOCAL variable will be created and initialized on every entry into the program module (procedure or function), just like a lexical, untyped LOCAL variable.

Scope, Visibility:

The scope, visibility and lifetime of TYPED LOCAL variables is identical to the usual, untyped lexical LOCAL variables. The only difference is the fixed storage type of C-like variables, which allows faster runtime access and their direct use in #Cinline statements, see below.

The variables are created and initialized upon entering a UDF, UDP or method (exactly when reaching the LOCAL..AS statement) and are destroyed when returning from that module. If a procedure or UDF is invoked recursively (calls itself), each recursive activation creates a new set of local variables.

TYPED LOCAL C-type variables can also be passed by value to other UDFs or UDPs called at the same level; passing them by reference is not supported. They may be used in code blocks with the same restriction as LOCAL vars.

Because TYPED variables have the same scope as lexical variables (LOCAL, STATIC), they hide all inherited PRIVATE, PARAMETERS, PUBLIC or FIELD variables with the same name.

For more information, refer to the section LNG.2.6.

Description:

LOCAL..AS is a declaration statement that declares TYPED lexical variables, very similar to untyped LOCAL variables. The advantages are:

- The type and storage range is fixed during compile-time and cannot be changed at runtime.
- The correct usage is already checked at compile time, which avoids possible run-time errors later during the execution.

Additional advantages of typed Object variables:

- The use of typed OBJECT variables increases program execution speed significantly, refer to chapter LNG.2.11.6 for additional information.

Notes about and restrictions of C-like types (BYTE, INT, LONG, DOUBLE etc):

- Since additional runtime type checking of C-like types may be omitted, their use results in faster program execution (up to 5 times), compared to usual typed or untyped variables. They are preferably used for large loops, calculations etc.
- The C-like typed variables can also be accessed directly in #Cinline statements, by giving the name (up to 10 significant chars) in lowercase.
- The variables occupy only 1, 4 or 8 bytes, compared to approx. 28 bytes for standard FlagShip variables.

- The programmer has to consider the maximum storage range of the variable's <type>. Otherwise, the resulting value will be truncated to the (lowest) available bytes.
- If the C-like typed variables are intermixed with usual untyped variables within an operation or command, they will be internally, temporarily converted to usual NUMERIC (or INTVAR) ones. Therefore, use only C-typed variables or constants within the e.g. FOR... declaration to maintain the speed advantage.
- C-like typed variables will always be passed to a standard function, UDF and UDP by value, regardless of the calling convention used (@ prefix or using the DO...WITH procedure call). The argument is automatically converted to a usual NUMERIC (or INTVAR) variable and received by the UDF as such. The same conversion occurs when assigning them to an array element.
- These variables cannot be used for any macro evaluation, as CLASS instances, and in code blocks. The function VALTYPE(varname) will return "N" (or "I" depending on FS_SET("intvar")), the TYPE("varname") function cannot be used.

The visibility is local to the current procedure or user-defined function. In FlagShip, the LOCAL..AS declarator may be placed anywhere within the function body; the scope and visibility of the corresponding local for the compiler starts from this declaration.

Short notation:

If the declarator is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the declaration (and initialization) is placed at the beginning of every module in the .prg file. The scope, visibility and lifetime is equivalent to an explicitly placed LOCAL..AS declarations in each one of these entities.

Example 1:

This example shows the usage and speed advantages of C-typed variables (remaining compatible to Clipper 5):

```
 *** file test.prg ***
 PARAMETERS cmd1, cmd2
 #ifndef FlagShip                   // Clipper redefinition
    #define LOCAL_INT    LOCAL      // not needed, if the
    #define LOCAL_DOUBLE LOCAL      // clipper switch
    #define SECONDSCPU   SECONDS    // Clipper test /uSTD.FH
 #endif                             // is used
 #define MAXLOOP 100000
 start (cmd1, cmd2)                 // passes cmd-line param
 QUIT
 FUNCTION start (cmd1, cmd2)
 LOCAL_INT    li_loop
 LOCAL        timestart := SECONDSCPU(), loc_loop
 LOCAL_DOUBLE ld_resulting
 ? "Input param.:", cmd1, cmd2

/* --- standard LOCAL variables --- */

 FOR loc_loop = 1 to MAXLOOP                // 100.000 times
```

```
    ld_resulting = loc_loop / 3
 NEXT
 ? "standard LOCAL: ", ;
   SECONDSCPU() - timestart, ld_resulting   // ca. 4.75 sec.

/* --- typed LOCAL variables ------ */
 timestart = SECONDSCPU()
 FOR li_loop = 1 to MAXLOOP
    ld_resulting = li_loop / 3.0
 NEXT
 ? "typed LOCAL: ", ;
   SECONDSCPU() - timestart, ld_resulting   // ca. 0.25sec.

/* --- The same using inline C code-- */

 #ifdef FlagShip
 timestart = SECONDSCPU()
 #Cinline
   #define MAXLOOP 100000           /* #define for C Code */
   for (li_loop = 1; li_loop <= MAXLOOP; li_loop++)
      ld_resulti = li_loop / 3.0;   /* use 10 chars only! */
 #endCinline
 ? "Inline C: ", ;
   SECONDSCPU() - timestart, ld_resulting   // ca. 0.18 sec.
 #endif

 RETURN NIL
 *** eof test.prg ***
```

Example 2:

This example shows the usage of typed variables and objects. See also chapter LNG.2.11.5 and LNG.2.11.6 for additional examples:

```
 LOCAL oDbf         AS DBSERVER
 LOCAL Getsys := {} AS GET              // local, nested GET
 LOCAL iIntVar := 0, iCount AS IntVar
 LOCAL cText := space(20)   AS CHARACTER
 LOCAL dMyDate := date()    AS DATE
 LOCAL lOk := .F., lResult := .T. AS LOGICAL
```

Classification:

programming

Compatibility:

The C-like types are available in FS only. CA/VO uses the same syntax (1). To remain compatibility to Clipper 5, use syntax 2 and #defines like:

```
#ifndef FlagShip
  #define LOCAL_BYTE   LOCAL
  #define LOCAL_LONG   LOCAL
  #define LOCAL_DOUBLE LOCAL
#endif
```

The usual types and object types are compatible to CA/VO (except IntVar), but not available in Clipper. For Clipper 5, you may specify e.g.

```
#ifndef FlagShip
```

```
   #command LOCAL <xx,...> AS <yy> => LOCAL <xx>
#endif
```

Related:

LOCAL, STATIC, GLOBAL, PRIVATE, PUBLIC, CLASS, FIELDS, PROTOTYPE, DO, FUNCTION, TYPE(), VALTYPE(), FS_SET("intvar"), INT2NUM(), NUM2INT(), #Cinline, #define, #ifdef

LOCATE ... FOR

Syntax:

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

Purpose:

Searches the working area for the first record meeting the specified condition.

Arguments:

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

Options:

<**scope**> is the portion of the current database file in which to perform the LOCATE. The default scope is ALL. The <scope> has no effect on CONTINUE.

WHILE <**condition**> specifies the set of records to be searched. These are the records meeting the condition. The WHILE condition is operational only until the first match is found, it has no effect on CONTINUE.

Description:

LOCATE searches the current database file sequentially from the beginning of the scope for the first record matching the condition. The search is terminated when a match is found, or the end of LOCATE scope is reached. After a successful LOCATE, FOUND() returns .T. and the matching record becomes the current record. Otherwise, FOUND() returns .F. and the record pointer is set to EOF or the first record outside the scope.

If you frequently search for a database key, SEEK on index will be the more effective, much faster alternative. If an index for a part of the <condition> is available, use SEEK and LOCATE...REST to skip unneeded records.

The LOCATE search can be initiated later by means of CONTINUE, which utilizes the FOR condition only. For a subsequent searching scope or WHILE, use SKIP and then LOCATE REST WHILE <condition> instead of CONTINUE.

Each working area can have its own LOCATE condition, which remains active until you execute another LOCATE command in that working area or close the database.

Example 1:

```
USE employee
? RECCOUNT()                           &&  98
LOCATE FOR Lastname = "Clifton"
? FOUND(), EOF(), RECNO()              &&  .T. .F. 43
CONTINUE
? FOUND(), EOF(), RECNO()              &&  .T. .F. 61
CONTINUE
? FOUND(), EOF(), RECNO()              &&  .F. .T. 99
LOCATE FOR Lastname = "Batman"
? FOUND(), EOF(), RECNO()              &&  .T. .F. 55
CONTINUE
? FOUND(), EOF(), RECNO()              &&  .F. .T. 99
```

Example 2:

```
USE address INDEX name
SEEK UPPER("smith")
DO WHILE FOUND()
   ? name, address, zip, city
   SKIP
   LOCATE REST FOR zipcode >= 1234 ;
          WHILE SUBSTR(UPPER(name),1,5) = "SMITH"
ENDDO
```

Classification:

database

Translation:

__DBLOCATE ({for}, {while}, next, rec, .rest.)

Related:

CONTINUE, FIND, SEEK, FOUND(), oRdd:LOcate(), oRdd:Locate(), oRdd:GetLocate()

MEMVAR

Syntax:

`MEMVAR <varList>`

Purpose:

Declares a list of identifiers to be used as PRIVATE and PUBLIC memory variables or arrays whenever encountered.

Arguments:

<**varList**> is a comma separated list of visible or declared PRIVATE and PUBLIC memory variables or arrays.

Scope:

The scope of the MEMVAR declaration depends on the location of the declaration statement:

- UDF wide scope: if the declaration is given within the procedure or function body, the scope is the UDF or UDP only.
- File-wide scope: if the declaration is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the scope is the entire .prg file (all UDFs or UDPs within these file).

Description:

MEMVAR is a declaration statement that causes the compiler to resolve references to variables specified without an explicit alias by implicitly assuming the memory alias `MEMVAR->`. Like all declaration statements, it has no effect on references made within macro expressions or variables.

The MEMVAR statement neither creates the variables nor verifies their existence. The variables may already exist (e.g. on a higher level) or will be created in the program body as autoPRIVATE or using the declarators PRIVATE, PARAMETERS, DECLARE or PUBLIC.

There is no fixed declaration order in FlagShip. The compiler starts the implicit aliasing from this declaration on.

In conjunction with the compiler switch -w (and e.g. also the FIELD declarator), unknown or ambiguous variable references will be printed (as warnings) at compile time.

Example:

With the compiler switch -w, a warning for "name" and "first" will be printed (main module and "first" in test2). Note the preference of dbf fields when accessing, or the memory variables when assigning the ambiguous variable names.

```
PRIVATE name
SELECT 5
USE address
first := "Peter"                        // autoPRIVATE
? name, first                           // Smith   John
DO test1
DO test2
RETURN

PROCEDURE test1
MEMVAR name, first
name := "Miller"                        // = assignment
? name, first                           // Miller  Peter
? address->name, address->first         // Smith   John
RETURN

PROCEDURE test2
FIELD name
? name, first                           // Smith     John
name := "NewMiller"                     // = REPLACE
? name, first                           // NewMiller John
? address->name, M->first               // NewMiller Peter
RETURN
```

Classification:

programming

Compatibility:

The MEMVAR declarator is new in FS4. This statement is compatible to C5, which has a fixed order of declaration statements (prior to the first executable statement).

Related:

FIELD, PRIVATE, DECLARE, PUBLIC, PARAMETERS, LOCAL, STATIC

MENU TO

Syntax:

```
MENU TO <memvar>
```

Purpose:

Executes a lightbar menu on currently defined prompts (@..PROMPT).

Arguments:

<**memvar**> is a memory variable where the choice will be placed after exiting from the menu. If the variable does not exist or is not visible, a new autoPRIVATE one is created.

Description:

Before executing a MENU TO lightbar menu, the item texts, positions on the screen and the order in which the lightbar will navigate through them need to be specified using the @...PROMPT command. Messages associated with PROMPTs, and their positions on the screen (SET MESSAGE) can also be defined.

Menu items and help texts are painted in the current "standard" color pair, the highlighted menu item appears in the "enhanced" color (see SET COLOR).

If the <memvar> contains a numeric value, the light bar is set to the corresponding item, or otherwise on the first one. MENU TO will then begin the selection process. To navigate through the PROMPTs, use the arrow keys to move the light bar to the next or previous menu item and display the associated (@..PROMPT..) MESSAGE, if any.

With SET WRAP ON, the downarrow key at the last prompt moves the lightbar to the first menu choice. The same happens with the uparrow key on the first choice.

By pressing the first menu character, the light bar is positioned on the first or next item, which starts with the pressed character, if any. If SET CONFIRM is OFF (the default) and the item is found, the choice is terminated and the current item position is stored in <memvar>. With SET CONFIRM ON, the user has to confirm the choice by pressing the enter key to leave the menu. You also may specify hot-key(accelerator) by prefacing the selected character by `"&", "\&"` or "\<", see details in @..PROMPT. This hotkey has then preference over the search by first character.

The following navigation keys can be used:

Key	Description	
Leftarrow <- (Cursor left)	Up one PROMPT	
Rightarrow -> (Cursor right)	Down one PROMPT	
Uparrow (Cursor up)	Up one PROMPT	
Downarrow (Cursor down)	Down one PROMPT	
Home	First menu item	
End	Last menu item	
Enter, Return	Exit, return PROMPT position	
PgUp, PgDn	Exit, return PROMPT position	
Esc	Abort, return zero	
Space	Select or search	(*) (**)
Hot-key	Go to corresp. item	(**)
First menu character	First/next PROMPT beginning with the same letter	(**)
Left-mouse double-click	Select item in GUI	(**)
Left-mouse click	Select item in GUI	(**)

(*) The space key usually handles same as Enter, i.e. it selects the current item. You may change this behavior by assigning .F. to the global variable `_aGlobSetting [GSET_L_PROMPTSPACESEL]`, it default is .T. When space select is enabled, the search for leading space in text is disabled.

(**) By this selection, the choice is terminated and the currently selected item position is stored in <memvar> when SET CONFIRM is OFF (the default). Otherwise, with SET CONFIRM ON, the user has to press the enter key to leave the menu.

Redirection:

When one of the navigation key is redirected via SET KEY or ON KEY or SET FUNCTION, the redirection is executed instead of the default behavior. The ESC key makes an exception: if ESC is re-directed, and you don't want to terminate MENU TO, pass anything else to LastKey() buffer e.g. KEYBOARD "x" ; Inkey() within the redirected UDF; otherwise MENU TO is terminated after returning from the redirected UDF.

Nesting:

The maximum number of PROMPTs per MENU TO is unlimited in FlagShip. MENU TO may be nested to any level when LOCAL or PRIVATE _oPrompt variable is declared to hold the nested PROMPTs, see details in the @..PROMPT command description.

If you wish to clear all @..PROMPT items without invoking MENU TO, use the CLEAR MENU command or _oPrompt:Clear()

The Prompt class is used internally for @..PROMPT items and MENU TO processing, the object is hold in _oPrompt. See also menuclass.fh

Tuning:

In GUI mode, you also may use Left/Middle/Right Mouse Button for the selection and wheel for positioning. You may enable/disable this action by assigning

```
_aGlobSetting[GSET_G_L_MENUTO_LMB  ] := .T. // LMB, default on
_aGlobSetting[GSET_G_L_MENUTO_MMB  ] := .T. // MMB, default on
_aGlobSetting[GSET_G_L_MENUTO_RMB  ] := .T. // RMB, default on
_aGlobSetting[GSET_G_L_MENUTO_DLMB ] := .T. // dblLMB, default on
_aGlobSetting[GSET_G_L_MENUTO_DMMB ] := .F. // dblMMB, default off
_aGlobSetting[GSET_G_L_MENUTO_DRMB ] := .F. // dblRMB, default off
_aGlobSetting[GSET_G_L_MENUTO_WHEEL] := .T. // wheel, default on
```

where the defaults are set in <FlagShip_dir>/system/initio.prg API. The LMB/MMB/RMB are left/mid/right mouse button, DLMB/DMMB/DRMB are double click the left/mid/right mouse button.

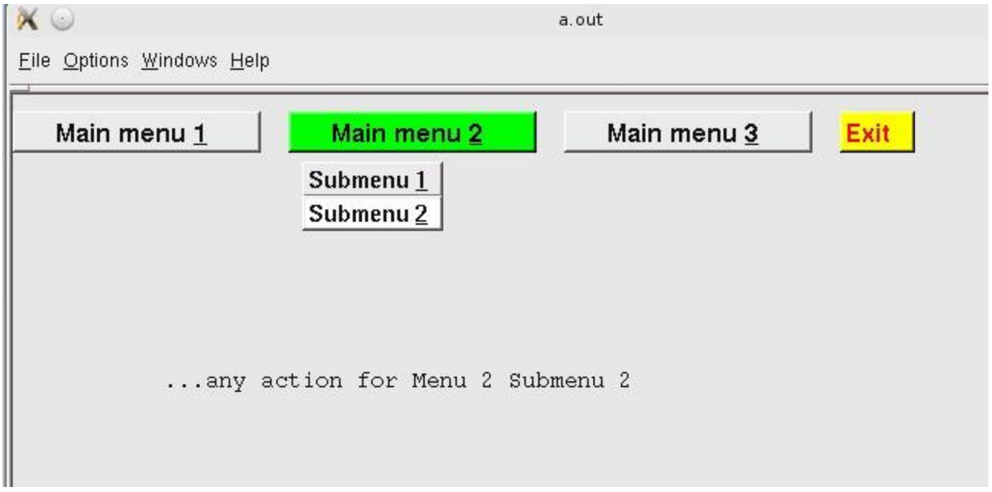
Example 1:

```
LOCAL choice := 3                           && light bar on "Append"
SET WRAP ON
@ 10,0  PROMPT "Help"
@ 11,0  PROMPT "Edit"
@ 12,0  PROMPT "Append"
@ 13,0  PROMPT "Delete"
@ 15,0  PROMPT "Quit"
MENU TO choice
SET WRAP OFF
IF choice = 0                               && ESC pressed
   RETURN
ENDIF
```

Example 2:

see more examples in @...PROMPT

Output:



```
...any action for Menu 2 Submenu 2
```

Classification:

programming

Compatibility:

The SET CONFIRM choice and the positioning to the first/next item is available in FlagShip only. Clipper supports only 32 PROMPTs per MENU TO, FlagShip unlimited.

Translation:

```
 memvar := __MENUTO ({|par| if(PCOUNT() == 0,  ;
                  memvar, memvar := par)}, "memvar")
```

Related:

@...PROMPT, CLEAR MENU, SET MESSAGE, SET WRAP, ACHOICE(), DBEDIT()

METHOD

Syntax 1:

```
ACCESS [METHOD] <methName> [( )]
        CLASS <className> [AS <type>]
```

Syntax 2:

```
ASSIGN [METHOD] <methName> (<par>)
        CLASS <className> [AS <type>]
```

Syntax 3:

```
[PROTECT] METHOD <methName> ([<paramList>])
        CLASS <className> [AS <type>]
```

Syntax 4:

```
PROTOTYPE ACCESS [METHOD] <methName> [()]
        CLASS <className> [AS <type>]

PROTOTYPE ASSIGN [METHOD] <methName> (<par>)
        CLASS <className> [AS <type>]

PROTOTYPE [PROTECT] METHOD
        <methName> [(<paramList>)]
        CLASS <className> [AS <type>]
```

Purpose:

Declares an access, assign or usual method, associated to the specified class.

Arguments:

<**methName**> is the declared name of the access, assign or usual method. The name may be of any length; only the first 10 characters are significant for access and assign, but significant in the full length for the usual method. Upper or lower case makes no difference. The names can contain any combination of characters A..Z, numbers, or underscores. The METHOD name must be unique within the class, but does not need to be unique within the application. The ACCESS or ASSIGN names may hide (or make accessible) same named instances of the same class, except EXPORTed ones.

<**className**> specifies the CLASS to which the access, assign or usual method belongs. The class has to be declared or prototyped already.

<**par**> **[AS** <**type**>**]** (in syntax 2, ASSIGN) specifies the value which should be assigned by the obj:methName := par syntax. It is passed to the ASSIGN method as a local variable. Optionally, you may give the variable a usual or an object <type> according to LOCAL..AS. The typed parameter, together with prototyping, allows additional type checking at compile and run-time.

<**paramList**> specifies optional parameters passed to the METHOD by the obj:methName(par1,par2...) syntax, same as parameters of a user defined FUNCTION. Optionally, you may give any parameter a usual or an object <type> according to LOCAL..AS by using the AS <type> syntax.

PROTECT METHOD is a usual METHOD, visible for the class entities (Access, Assign and Methods) only, but hidden from the usual application. It is used mainly as a class internal UDF, similar to a STATIC function. The important difference is, that the class instances are visible also within the PROTECT METHOD body. You would otherwise have to pass all the required instances via a parameter list to a usual (or static) UDF. Additionally, the protect method is available also for methods of the same class specified in other files.

PROTOTYPE (using syntax 4) informs the compiler about the CLASS entity, specified elsewhere later in the application. Knowing the method name, the FlagShip compiler is able to resolve the addresses during compile-time (early binding). Otherwise, the method address will be resolved during the run-time phase (late binding), see also chapter LNG.2.11.6. You will not need to prototype a method declared formerly in the same source, but should prototype methods used, but specified later. See also the PROTOTYPE statement in section CMD.

Description:

A **METHOD** is very similar to a usual user defined function (UDF). The only visible difference is, that the name is associated to the specified class. Therefore, it may only be invoked together with the object name and the send operator, e.g. oMyObj:MyMethod() as opposed to invoking a usual UDF by the name (and parentheses) only. An instance of any type and/or an access/assign method of the same name may be specified in the same class. A PROTECT METHOD is the same, but visible within the class entities only.

The **ACCESS** method is a special kind of method, which receives no parameters. It acts as a "read-only virtual export instance". Therefore, the access method is invoked from the application in the same way as an exported instance, e.g. [resultVar :=] oMyObj:MyVar. A same named instance (of any type except EXPORT) may, but need not be specified in the CLASS declaration. A same named ASSIGN and usual METHOD may coexist.

The **ASSIGN** method is also a special kind of method, whereby the assigned value is passed as a local parameter. The ASSIGN acts as a "write-only virtual export instance with optional validation". Therefore, the assign method is invoked from the application in the same way as an exported instance, e.g. oMyObj:MyVar := assignedValue. A same named instance (of any type except EXPORT) may, but need not be specified in the CLASS declaration. A same named ACCESS and usual METHOD may coexist.

Method programming:

The ACCESS, ASSIGN and METHOD are programmed in the same way, as usual UDFs. The method starts with the declarator according to syntax 1, 2 or 3, which is similar to the FUNCTION declarator of a UDF. The method body includes any number of valid statements and ends with the next UDF, UDP or method declarator or by end-of- file. The ACCESS method should always return the (virtual) object instance value (or an empty value on error), while the ASSIGN and METHOD may return any value. Usually, ASSIGN also returns the newly set (virtual) object instance value.

Within the ACCESS, ASSIGN and METHOD program body, you have direct access to all class instances. Their visibility for the method is similar to "private" variables, they may therefore be hidden by same named LOCAL or STATIC variables. If so, you may explicitly access the instance by using the `SELF:` keyword. If a same named instance and PRIVATE or PUBLIC variable exists, the instance is preferred. Generally, you may always use the `SELF:` keyword when referring to any instance variable of the same class.

Within the method body, the ACCESS and ASSIGN method is invoked instead of the instance, if such an assign or access method exists, and if it overloads a usual INSTANCE variable. All other instance types (EXPORT, PROTECT, HIDDEN) are accessed directly. Of course, in the ACCESS and ASSIGN body, the same named instance variable is accessed directly, regardless of its type.

In the access, assign and method body, invoking METHODs of the same class is always performed with the `SELF:` keyword. If the class is inherited from a superclass, and a local (redefined) method exists, the `SELF:` keyword calls the locally redefined method, while `SUPER:` invokes the original, inherited one.

Example 1:

See examples in the INSTANCE description, demonstrating the declaration of the CLASS and its methods in the same, or in different source files. See also the <FlagShip_dir>/system/smallrdd/smallrdd.prg file for a practical example of the OOP programming.

Example 2:

Appends a new assign method to the standard GET class for a controlled modification of the get:BUFFER. Note: the prototypes of the GET class are included in the "getclass.fh" or the general "stdclass.fh" file, which may be #included in your source (or automatically from the "std.fh" preprocessor file).

```
#include "getclass.fh"
ASSIGN FillBuff(cValue) CLASS get AS character
if valtype(cValue) == "C"
   cValue := PADR(cValue, LEN(self:buffer))
   self:buffer := cValue      // the SELF: keyword is not
endif                         // required here, but makes
return self:buffer            // the code better readable
```

Example 3:

Redefines some methods of the standard DBFIDX class. Note: the prototypes of the DBSERVER class are included in the "dbfidx.fh" or the general "stdclass.fh" file. The 'MyCrea' method is not visible for the rest of the application.

```
#include "dbfidx.fh"
CLASS DbfIdxExcl INHERIT DbfIdx     // my DBserver RDD

PROTECT METHOD MyCrea(p1 AS char) CLASS DbfIdxExcl AS logic
   if ALERT("Database "+p1+" does not exist. Create ?", ;
            {"No", "Yes"}) == 2
      if ! MyCreateUdf (p1, self:info(DBI_FULLPATH)) // UDF
         return .F.
      endif
   endif
   return .T.

METHOD Init(p1,p2,p3,p4,p5,p5) CLASS DbfIdxExcl
  WHILE .T.
   super:INIT (p1,.F.,p3,p4,p5,p6)  // invoke dbfidx:init()
   if self:used .or. file(self:info(DBI_FULLPATH))
      exit
   endif
   if self:MyCrea(p1)               // protected method
      exit
   endif
  ENDDO
  return self                       // returns object

ACCESS Shared CLASS DbfIdxExcl      // local redefinition
  return .F.

STATIC FUNCTION MyCreateUdf (cName, cFullName)
   // create the dbf, e.g. by using dbcreate()
return .T.

FUNCTION myApplic ()                // main program entry
LOCAL oMyDbf := DbfIdxExcl {"myData"} AS DbfIdxExcl
if ! oMyDbf:Used
   ? "cannot open myData.dbf exclusive"
   QUIT
endif
// process, e.g.
? EOF(),   oMyDbf:EOF
? ALIAS(), oMyDbf:ALIAS
```

Classification:

programming

Compatibility:

Not available in Clipper, but compatible to CA/VO. The PROTOTYPE clause is available in FlagShip only (managed by the repository in VO). PROTECTed METHODs are available in FlagShip only.

Related:

INSTANCE, PROTOTYPE, LOCAL..AS, (OBJ)DBSERVER, (LNG)2.11

NOTE

Syntax:

`NOTE [<text>]`

or:

`* [<text>]`

Purpose:

Puts a comment at the beginning of a line.

Arguments:

<**text**> is a character string ending with a new line.

Description:

The NOTE command is equivalent to the asterisk `"*"` comment. `NOTE` and `*` at beginning of the source line (leading spaces and TABs are not significant) marks the whole line as a (full-line) comment.

For more program comments, see (CMD) * Comments.

Example:

```
*************
*  Comment  *
*************
a = b                                   && Inline comment,
a = b + ;                               // usable also for
    c + d                               && continued statement
NOTE That is an comment line,
NOTE   same as these
*      or these line.
```

Classification:

programming

Related:

* and // and && and /*..*/

ON ANY KEY
ON KEY

Syntax 1:

```
ON KEY [DO <udfName>]
ON ANY KEY [DO <udfName>]
```

Syntax 2:

```
ON KEY CLEAR
```

Syntax 3:

```
ON KEY <expN1> [EVAL <expB2>]
```

Syntax 4:

```
ON KEY <expN1> [DO <udfName> [WITH param]]
```

Purpose:

ON KEY is very similar to SET KEY with some additional features.

Arguments:

<**expN1**> is numeric key value corresponding to Inkey()

<**udfName**> is name of any standard or user defined function

<**expB2**> is a code block to be evaluated

Description:

Syntax 1 set/clear action performed on any key press. It is a special case of syntax 4 and equivalent to ON KEY 0 [DO ...] or the invocation of OnKey(0, {|| udfName() })

Syntax 2 clear all previously set ON KEY actions and is same as OnKey(NIL, NIL)

Syntax 3 set/clear a codeblock, evaluated on key-press of a specific key, same as SetKey(expN1, [expB2]) or to the invocation of OnKey(expN1, [expB2])

Syntax 4 is similar to syntax 3 and equivalent to SET KEY nKey [TO udf]. In fact, the DO udfName WITH ... clause is translated by FlagShip preprocessor to OnKey(expN1, {|| udfName(param1,param2,param3,...)})

Classification:

programming

Compatibility:

New in FS5, compatible to FoxPro

Related:

OnKey(), SET KEY, SetKey(), PUSH KEY, POP KEY

ON ERROR

Syntax:

```
ON ERROR [DO]
ON ERROR [DO <udfName> [WITH param]]
```

Purpose:

ON ERROR is provided mainly for FoxPro compatibility

Description:

Set/clear action executed on RTE, same as ErrorBlock(...). In fact, the command "ON ERROR [DO]" is translated by the FlagShip preprocessor to

```
if type("_OnError") == "B"
   ErrorBlock(_OnError)
   _OnError  := NIL
   _OnErrObj := NIL
endif
```

and the command "ON ERROR [DO <udfName> [WITH param]]" creates a public variable with assigned ErrorBlock() to it:

```
PUBLIC _OnErrObj       // holds the error object
PUBLIC _OnError := ErrorBlock({|_err| _OnErrObj := _err, ;
                                udfName(par1,par2..) } )
```

When compiled with the -fox switch, also FoxPro's functions Error() and Message() are available, see foxpro_api.prg

Example: *compile with -fox switch for FoxPro's functions*

```
/* force RTE 501 on failure in USE... and SET INDEX...
 * for FoxPro compatibility
 */
_aGlobSetting[GSET_L_DBUSEAREA_ERR ] := .T. // default = .F.
_aGlobSetting[GSET_L_DBSETINDEX_ERR] := .T. // default = .F.

set font "courier"
on error do errhand          // activate own error handler
do testUdf
on error                     // de-activate own error handler
?
wait "now standard error will be raised..."
x = yz                       // error, variable 'yz' n/a
wait

procedure testUdf
use unknown index unknown    // RTE 501, see above settings
x = abc                      // error, variable 'abc' n/a
return

procedure errhand            // own error handler
? "*** FoxPro error  =", error()
? "    FlagShip error=", if(type("_OnErrObj") == "O", ;
                            _OnErrObj:GenCode, 0)
? "    message() =", message()                    // default
? "    message() =", strtran(message(), ";", " ") // no NL
? "    message(1)=", message(1)
? "    message(2)=", message(2)  // extended, n/a in Fox
? "    sys(16)   =", sys(16)
wait "(in " + procname() + ") press any key..."
/* If ON ERROR ... is active, and (in this example) other
 * than FoxPro error 1 occurred, display error by standard
 * FlagShip popup, then select this own error handler back
 */
if type("_OnError") == "B" .and. type("_OnErrObj") == "O"
   if error() <> 1
      local bSaveErr := ErrorBlock(_OnError)    // save
      _rt_error(_OnErrObj:GenCode, _OnErrObj:Description)
      ErrorBlock(bSaveErr)                      // restore
   endif
endif
return
```

Classification:

programming

Compatibility:

New in FS5, compatible to FoxPro

Related:

OnKey(), ErrorBlock()

ON ESCAPE

Syntax:

```
ON ESCAPE [DO <udfName> [WITH param]]
```

Purpose:

ON ESCAPE is provided mainly for FoxPro compatibility

Description:

Set/clear a UDF, evaluated on ESC key press. It is in fact a special case of "ON KEY 27 [DO <udfName> [WITH param]]" or of "SET KEY 27 TO <udfName>" and is hence evaluated same as

```
OnKey (27, {|a,b,c,d| udfName (par1, par2, ...) })
```

Classification:

programming

Compatibility:

New in FS5, compatible to FoxPro

Related:

ON KEY, OnKey(), SetKey(), PUSH KEY, POP KEY

PACK

Syntax:

`PACK`

Purpose:

Removes all records marked for deletion from the current database (and the associated .dbt) file.

Description:

PACK physically removes all records marked for deletion, REINDEXes all open indices in that working area, and restores the space previously occupied by removed records and index keys.

PACK only acts on the selected database file and its associated memo-files, the currently set RELATIONs and FILTERs are ignored.

PACK does not create any backup files (except during its execution); so if one is required, COPY TO should be invoked prior to PACK. After the command is finished (including REINDEXing of all open indices), the record pointer is reset to the first logical record and the original FILTER and RELATION are restored.

Performance:

On large databases, PACK can be a time-consuming process, very uncomfortable in a multiuser environment. In such a case, the PACK can be easily omitted using the records already deleted for new ones; see example below.

Multiuser:

PACK requires an exclusively opened database using SET EXCLUSIVE ON or USE...EXCLUSIVE. If not so, RTE (Run-Time-Error) displays and PACK is not performed. See also LNG.4.8.

Tuning:

PACK creates temporary database newNNNNN.dbf (and .dbt, dbv, .fpt if required) in the same directory where the database resides. The NNNNN is the current process ID number of the executable. If such a file exist, newHHMMSSUUUU.dbf is created, the format is of Time(1). These files are deleted after completing the PACK. You may assign any other directory for these temporary files by environment variable FSPACKDIR, e.g. `SET FSPACKDIR=[drive:]\path` in Windows, or `export FSPACKDIR=/path` in Linux.

Example 1:

```
USE employee
COUNT FOR Hair_len > 30 .AND. Sex = "M" TO Freaks
? RECCOUNT(), Freaks                             && 100   7
DELETE FOR Hair_len > 30 .AND. Sex = "M"
? RECCOUNT()                                     && 100
PACK
? RECCOUNT()                                     && 93
```

Example 2:

Omitting the PACK by re-using the deleted records. All deleted records are contained in the index at the logical top. Then, use MYDELETE() instead of DELETE + PACK and MYAPPEND() instead of APPEND BLANK.

```
USE address
WHILE NETERR()
   INKEY (2)                               // if busy,
   USE address                             // retry
END
SET INDEX TO name, zip
SET DELETED ON
GOTO TOP                                   // first valid data
choice := my_menu()
IF choice == 1
   MYDELETE()                              // deleting required
ELSEIF choice == 2
   MYAPPEND()                              // appending required
   REPLACE name with xName, zip with xZip
   UNLOCK
ENDIF
:

FUNCTION MYDELETE()                        // repl. DELETE+PACK
WHILE !RLOCK(); END                        // wait for lock
DELETE
REPLACE name with " ", zip with 0          // reset index order
UNLOCK
RETURN

FUNCTION MYAPPEND()                        // repl. APPEND BLANK
SET DELETED OFF
GOTO TOP
if DELETED()                               // deleted available?
   WHILE !RLOCK(); END                     // yes, remove the
   RECALL                                  //   "delete" mark
else
   APPEND BLANK                            // no, append new one
   WHILE NETERR(); APPEND BLANK; END
end
** UNLOCK     // record remains locked, simil.to APPE BLANK
SET DELETED ON
RETURN                                     // RLOCK() is set
```

Classification: *database*

Compatibility:

FlagShip also PACKs the associated memo-files (.dbt), whilst Clipper packs the .dbf file only.

Translation:

__DBPACK( )

Related:

DELETE, RECALL, REINDEX, SET EXCLUSIVE, SET DELETED, USE, ZAP, DELETED(), FLOCK(), RLOCK, oRdd:PACK(), ISDBEXCL()

PARAMETERS

Syntax:

```
PARAMETERS <paramList>
```

Purpose:

Specifies PRIVATE memory variables as FUNCTION or PROCEDURE parameters which will receive the arguments of the call (passed by value or by reference).

Arguments:

<**paramList**> is a comma separated list of receiving variables. The variables will be created in the PRIVATE class. The number of receiving variables does not have to match the number of arguments passed by the calling procedure UDP or user-defined function UDF.

Description:

The values or references actually passed by a call of UDP or UDF are referred to as **arguments**. The variables in the UDP or UDF, which receives them, are named **parameters**. Receiving parameters can be created by using the PARAMETERS command or alternatively as LOCAL variables (named **formal parameters**) if specified as a part of the PROCEDURE or FUNCTION declaration statement (i.e. included in parentheses).

When a PARAMETERS statement executes, all variables in the parameter list are created as PRIVATE class variables and all previous public or private variables with the same names are hidden until the current procedure or UDF terminates. The scope, visibility and lifetime of such variables is equivalent to those of the PRIVATE declaration.

In FlagShip, the number of arguments and parameters do not have to match. If there are more arguments than parameters, the rest of the arguments are ignored. Conversely, when there are more parameters than arguments actually passed, the rest of the parameters remain undefined (contain NIL). To find out how many arguments were passed, use PCOUNT(). If some arguments are omitted, the corresponding parameters become NIL.

Arguments can be passed in one of two ways: by value or by reference. To pass a parameter by value means that its value is copied to the receiving variable. When a parameter is passed by reference, the argument is just given the parameter name and the parameter contains this variable. In the former case, the value of the argument cannot be altered in the procedure, while in the latter, all changes to the parameter also happen to the corresponding argument.

When parameters are passed to procedures using DO.. ..WITH, memory variables and arrays are passed by reference, while array elements, expressions, fields and memory variables enclosed in round parenthesis ( ) are passed by value. When parameters are passed to user defined functions (UDFs), all parameters except for arrays are passed by value. Memory variables however, can be passed by reference if preceded by @. Parameters from the Unix or Windows shell may also be passed to the main program module e.g. [./a.out param1 param2] or [myapplic.exe param1 "second param"]

For more information, see section LNG.2.3.2.

Example 1:

```
* call it e.g.: a.out  -or-  myapplic xxx "yyy zzz" -etc-
PARAMETERS cmd1, cmd2, cmd3           // received from command-line
FOR i = 1 to PCOUNT()
   ii = LTRIM(STR(i))
   ? "Command-Line-Parameter " + ii + " = " + cmd&ii
NEXT
DECLARE arr [2]
arr[1] := 1 ; arr[2] := 1
v1=1 ; v2=1
DO chg WITH arr,arr[2],v1,(v2)
? arr[1], arr[2], v1, v2                // 2 3 4 1
RETURN

PROCEDURE chg
PARAMETERS p1,p2,p3,p4                  // procedure params
p1[1] := 2 ; p1[2] := 5                 // change array elem.
p2 := 3                                 // redeclares p1[2]
p3 := 4                                 // changes v1
p4 := 9                                 // v2 remains unchanged
RETURN
```

Example 2:

```
additional examples are in FUN.Pcount() and FUN.Param()
```

Classification:

programming

Related:

DO, FUNCTION, PRIVATE, PUBLIC, LOCAL, STATIC, SET PROCEDURE TO, Pcount(), Param()

PRIVATE

Syntax 1:

```
PRIVATE <memvar> [:= <exp>] [, ... ]
```

Syntax 2:

```
PRIVATE <array> [<dim>]
PRIVATE <array> [<dim1>,<dim2>,<dimN>]
PRIVATE <array> [<dim1>][<dim2>][<dimN>]
PRIVATE <array> := {<exp>,... }
```

Purpose:

Creates and initializes the specified memory variables or arrays in the PRIVATE class.

Arguments:

<**memvar**> is the list of variables or arrays to be created as PRIVATEs. In this list, arrays and other variables can be interchanged. The name may be of any length, but only the first 10 characters are significant (see more LNG.2.6). Variable names in the FlagShip language are not case sensitive.

<**array**> is the name of the array to be created. The naming convention is the same as in <memvar>. The square brackets `[ ]` behind the <array> name do not in this case specify an optional argument, but are a required part of the syntax. The number of elements for each array dimension can be specified as `[dim1, dim2,..,dimN]` or `[dim1][dim2][dimN]` . The maximum number of dimensions and of elements per dimension in FlagShip is 65535.

The PRIVATE <array> statement is equivalent to DECLARE <array>. Array elements can be handled like ordinary memory variables. Different elements of the same array can be of different types. Each element may contain another sub-array (non-symmetric structure), see LNG.2.6.4.

Options, Initializing:

<**exp**> is any valid FlagShip expression including a literal (constant) array to initialize the variable. If the initializer (:= <exp>) is not given, the variable (or all array elements) will be set to NIL. Initializing of a variable created by composed macro (e.g. PRIVATE var&macro := value) is not supported but the sequence PRIVATE var&macro ; var&macro := value is ok.

The array elements can be declared and initialized with a starting value using an array (literal) constant (see LNG.2.7) including any valid expression and the assign := operator. Initialization is performed at the time of the variable creation, that is, when executing the PRIVATE (or DECLARE, PARAMETERS) statement.

Scope, Visibility:

PRIVATE variables have dynamic scope. These variables are visible within the current and all UDFs and UDPs called from within. When a private variable or array is created, existing and visible PRIVATE and PUBLIC variables of the same name are hidden until the current procedure or user-defined function terminates. Private variables exist for the duration of the active procedure or until explicitly released with CLEAR ALL, CLEAR MEMORY, or RELEASE. If a procedure or UDF is invoked recursively (calls itself), each recursive activation creates a new set of PRIVATE variables.

PRIVATE variables can be passed by value or by reference to other UDFs or UDPs called at the same level. In code blocks, only PRIVATE variables of the module where the block is **executed** are visible; see LNG.2.3.3.

For more information about variables, refer to the section LNG.2.6.

Description:

PRIVATE is an executable statement which creates and initializes a new variable or array in the dynamic scoping class. If the same named variable does not exist, it is equivalent to the creation of an autoPRIVATE variable using an assignment. The PRIVATE statement is equivalent to DECLARE and similar to the PARAMETERS command.

Short notation: if the PRIVATE declarator is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the declaration (and initialization) is placed at the start of every module in the .prg file. The scope, visibility and lifetime is equivalent to explicitly placed PRIVATE declarations in each of these entities.

Example:

```
PRIVATE var1, arr1[5], var2, arr2[2,3]
DECLARE arr3[4], arr5[2,2]
PRIVATE var6 := 24, arr7 := {0, 0, {0, 0}}
PRIVATE menu := {"Show", "Add", "Print", "Exit"}
```

Classification:

programming

Compatibility:

The initialization during the declaration is new in FS4. PRIVATE variables are available in all xBASE languages. Only FlagShip supports an unlimited number of variables, 64k * 64k array element size and short notation.

Related:

DECLARE, PARAMETERS, PUBLIC, LOCAL, STATIC, GLOBAL, FIELD, MEMVAR

PROCEDURE

Syntax 1:

```
PROCEDURE <udpName> [AS USUAL]
[PARAMETERS <parList>]
     <statements>...
RETURN
```

Syntax 2:

```
PROCEDURE <udfName> ([<parList>]) [AS USUAL]
     <statements>...
RETURN
```

Syntax 3:

```
[STATIC|INIT|EXIT] PROCEDURE
        <udfName> ([<parList>]) [AS USUAL]
     <statements>...
RETURN
```

Purpose:

Identifies the beginning of a user-defined procedure (UDP) or a startup/exit procedure.

Arguments:

<**udpName**> is the name of the procedure UDP. The name may be of any length, only the first 10 characters are significant and is not case sensitive (for more details refer to section LNG.2.3). Names starting with an underscore are reserved for FlagShip.

<**udfName**> is the name of a function (UDF), see below.

RETURN terminates the execution of the UDP and passes control back to the calling program. Any number of RETURNs are accepted within the UDP.

Options:

STATIC PROCEDURE declares an UDP, which is visible in the current .prg file only. Several STATIC UDPs and UDFs (and only one public UDP/UDF) may be defined with the same name in different .prg files.

Because the references to a STATIC function are resolved at compile-time, they will hide public UDP or UDF with the same name. STATIC procedures are not visible and therefore cannot be used during a macro evaluation or as UDF for ACHOICE(), MEMOEDIT() etc.

When the keyword STATIC is omitted, the UDP becomes public and the name is visible to the whole application.

PARAMETERS <**parList**> specifies one or more comma-separated PRIVATE variables which receive the calling arguments. See more in the PARAMETERS command.

(<**parList**>**)** is the alternative syntax to the PARAMETERS command, but the variables in <paramList> have LOCAL type and may optionally be typed, see below.

AS USUAL (proto)types the procedure and specifies, that the compiler should include its PROTOTYPE into the repository file.

Init/Exit:

INIT PROCEDURE declares an initialization procedure, which will be executed at program startup. An arbitrary number of INIT PROCEDUREs may be declared. They will be successively invoked prior to the first executable statement in the main module, one after the other. The visibility of the INIT procedures is restricted to the FlagShip system. Each procedure receives a copy of the Unix or Windows command line arguments given when invoking the executable, passed to the <parList>. See the note on linking and calling below.

EXIT PROCEDURE declares an exit procedure, which will be executed at program termination. Any number of EXIT PROCEDUREs may be declared for the whole application. The EXIT procedures are successively invoked after the last executable statement in the main module (or from the QUIT or CANCEL command) prior to returning to the Unix/Windows shell. The visibility of the EXIT procedure is restricted to the FlagShip system. Each EXIT procedure receives a numeric parameter, representing the sequence order of the EXIT procedure, starting with one. The execution of an EXIT procedure cannot be guaranteed when the system encounters an unrecoverable error. See also linking and calling note.

Linking and calling the INIT/EXIT procedures: If the .prg source file consists only of INIT and/or EXIT procedures, the automatic compilation rule (see below) cannot apply. Instead, the source .prg or the object .o file must be specified when invoking FlagShip during the compile/link phase. Also, the ANNOUNCE/REQUEST declarators (or EXTERNAL <prgname> if compiled without the -na switch) must be used to specify the external. The INIT and/or EXIT procedures will be executed in the same order as their corresponding files were specified in the FlagShip command line during the linking phase.

Prototyping of parameters:

The local parameters specified in brackets (according to syntax 2) may optionally be typed (with all usual <type>s according to LOCAL..AS), and/or prototyped as optional. The syntax is equivalent to **(**<paramList>**)** of the PROTOTYPE declarator, e.g.

```
PROCEDURE myUdp (p1 AS CHAR, [p2 as NUMER], p3, [p4])
```

If the <type> is not given (e.g. parameters p3 and p4 in this example), AS USUAL is assumed. The parameter name enclosed in square brackets [ ] (visually) signals an optional parameter, used also in (and passed to) UDP prototypes. It does not change the behavior of parameter passing, nor the parameter order in any way.

Also the return procedure <type> may be prototyped by AS USUAL according to syntax 2.

Purpose: Giving the parameters a <type> allows a compile-time check of the parameters (arguments) passed to the procedure at places where it is invoked. This compile-time check will help you to avoid unexpected RTEs (run-time errors) and simplify parameter validation in the procedure body. See also "parameter passing" below. Use the PROTOTYPE declarator (e.g. in an #include file), when the UDP is invoked in other than the current file (prototyping); or when the UDP is specified in the same file, but is invoked before its declaration (forward prototyping) to take advantage of the compile-time checking.

Note: the PROTOTYPE statement is automatically created in the repository file (for AS USUAL typed UDPs only) by using the -ru compiler switch, see FSC.1.3.

All standard FlagShip functions and procedures are prototyped in the stdfunct.fh file.

Description:

Functions and procedures increase both readability and modularity, isolate changes and standardize a block of frequently-used statements.

A user-defined procedure UDP is called with the command :

```
DO udpname
DO udpname WITH param1 [, param2 ...]
```

Procedures can also be called by using **macros**, e.g.:

```
udpname = "my_proced"
myparam = "xyz"; xyz := 5
DO &udpname  WITH "test", &myparam
DO (udpname) WITH "test", &(myparam)
DO my_proced WITH "test", 5
```

Procedures may also be called in FlagShip using the UDF syntax. See FUNCTION command.

The UDP may call itself **recursively**. The number of recursions is in FlagShip limited only by the available RAM + swap disk space to store the local data of each recursion.

Automatic procedures:

If the compiler switch -na is **not** given, FlagShip generates an automatic PROCEDURE carrying the name of the file without extension, for compatibility purposes. When starting the source file with a PROCEDURE of the same name as the file, an (additional) automatic procedure is not generated.

Parameter passing:

The calling arguments when using the DO...WITH command are passed to a user-defined procedure by reference, except constants, expressions and database fields, which are always passed by value. To pass a variable by value, the argument has to be enclosed in parentheses, e.g. DO myproc WITH (var1), (var2), var3.

The UDP copies the passed argument values or references into predefined PUBLIC or LOCAL variables in the <parList>. The number of arguments passed and parameters received need not be the same. Arguments can be skipped or left off the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. Refer to LNG.2.3.2 and (CMD) PARAMETERS for a more detailed discussion.

On typed parameters, only arguments of the specified parameter type are accepted. If the prototype of the UDP is known at compile time (see prototyping), an incorrect argument passing is reported by the FlagShip compiler. If the prototype or the argument type is unknown at compile time, and an incorrect argument type is passed, a run-time error occurs. On optional parameters (i.e. enclosed in square brackets), only the specified type or NIL is accepted.

UDF vs. UDP

In FlagShip, the only difference between the call to a function (UDF) or procedure (UDP) is the convention of default parameter passing. Both UDF and UDP can be used interchangeably, so if an UDP is called using the function syntax, the arguments/ parameters are passed by value, instead of by reference.

Automatic compilation:

If the compiler switch `-m` is **not** given, every time it finds a DO statement and the name of the procedure is unknown, the compiler searches the current directory for a source file with the same name in order to compile it. Refer to the (CMD) DO statement.

Example 1:

Notice the main procedure calling two subprocedures, two of them in the same program file, two in a separate file:

```
*** Main program file test.prg
DO Proc1
DO Proc2 with "Main"
DO Proc3 with 5
RETURN
PROCEDURE Proc1
? "The first procedure"
DO Proc2 with PROCNAME()                && current procedure name
RETURN
PROCEDURE Proc2
PARAMETERS par1
? "The second procedure, called from " + par1
DO Proc4
RETURN
*** eof test.prg
*** file proc3.prg
** PROCEDURE Proc3                      && omit this declaration
PARAMETERS p1
// any statements
RETURN

PROCEDURE proc4
// any statements
RETURN
*** eof proc3.prg
Compile: $ FlagShip test.prg -otest
```

Example 2:

Usage of INIT/EXIT procedures, e.g. to measure the execution time:

```
*** Main program file test.prg
```

```
STATIC timecpu, timeall                    // .prg wide scope

PROCEDURE main (cmd1, cmd2)                // main module
USE address
LIST name, address FOR inkey() != 27
RETURN                                     // return to OS

INIT PROCEDURE startup (cmd1, cmd2)     // init procedure
timeall := SECONDS()
timecpu := SECONDSCPU()
RETURN

EXIT PROCEDURE exitproc ()                 // exit procedure
?
? "Time elapsed  : ", SECONDS()   - timeall, "seconds"
? "real CPU time : ", SECONDSCPU()- timecpu, "seconds"
RETURN
Compile: $ FlagShip test.prg -Mmain -na -otest
```

Classification:

programming

Compatibility:

The STATIC clause, the usage of formal LOCAL parameters and the INIT/EXIT procedures are compatible to C5. The <parN> in the EXIT PROCEDURE is available in FS4 only. FlagShip accepts the interchangeable UDF/UDP calling convention. Typed parameters and typed functions are supported by FS4 and VO. The definition of optional parameters by using square brackets is available in FlagShip only.

Related:

DO, SET PROCEDURE, FUNCTION, PROTOTYPE, LOCAL, PCOUNT(), Param()

PROTECT INSTANCE

Syntax 1:

```
      [STATIC] CLASS <ClassName> [INHERIT <SuperClass>]
```

and optional:

```
      INSTANCE <Name> [:= <exp>] [AS <type>]
      EXPORT [INSTANCE] <Name> ...
      HIDDEN [INSTANCE] <Name> ...
      PROTECT [INSTANCE] <Name> ...
```

Syntax 2:

```
      PROTOTYPE [STATIC] CLASS <ClassName>
               [INHERIT <SuperClass>]
```

and optional:

```
      INSTANCE <Name> [AS <type>]
      EXPORT|HIDDEN|PROTECT [INSTANCE] <Name> [AS <type>]
```

See detailed description in the CLASS command.

PROTOTYPE

Syntax 1:

```
PROTOTYPE [STATIC] CLASS <ClassName>
        [INHERIT <SuperClass>]
```

and optional:

```
EXPORT|HIDDEN|PROTECT [INSTANCE] <Name> [AS <type>]
INSTANCE <Name> [AS <type>]
```

Syntax 2:

```
PROTOTYPE ACCESS [METHOD] <methName> [ () ]
        CLASS <className> [AS <type>]

PROTOTYPE ASSIGN [METHOD] <methName> (<par1>)
        CLASS <className> [AS <type>]

PROTOTYPE [PROTECT] METHOD <methName>
        [(<paramList>)]
        CLASS <className> [AS <type>]
```

Syntax 3:

```
PROTOTYPE FUNCTION <udfName> [(<paramList>)]
        [AS <type>]
```

Purpose:

Informs the compiler about the class entities or about the type of a user defined function in order to optimize the class access and/or perform type checking during the compilation phase.

Description:

PROTOTYPE (according to syntax 1 and 2) informs the compiler about the CLASS structure, it's instances and methods. If PROTOTYPEs of the class are unknown at compile-time, the slower run-time address resolving is generated, see LNG.2.11.6. Refer also to the CLASS and METHOD description. For the FlagShip standard classes, the prototypes are specified in the <xxx>class.fh, or the summarized "stdclass.fh" file, which may automatically be included from within "std.fh".

PROTOTYPE (according to syntax 3) informs the compiler about the UDF type (and parameters), to perform compile-time and/or run-time invocation and parameter checking. For the FlagShip standard functions, the prototypes are specified in the "stdfunct.fh" file, and can also be automatically included from "std.fh".

Since the PROTOTYPE is non-executable compiler information only, it may be placed anywhere in the source. The prototype becomes active for all subsequent lines in the .prg source file. You may preferably place prototypes in a separate, project specific .fh file, which will be #include'd in the required .prg sources. The most convenient method is to #include "myproto.fh" at the end of the **local copy** of the std.fh file.

Automatic prototype generation: the FlagShip compiler is able to automatically extract all prototypes from your source into a file named "reposit.fh", when the compiler switch -rc and/or -ru is specified (see section FSC.1.3).

Hint: In a large application, you may pre-compile all *.prg sources of the application (e.g by using -c -rc -ru -r=myprot.fh switches), then check the produced file myprot.fh and #include it into the local copy of the std.fh file... and your application may be finally compiled. The compiler will now know all declared (typed) functions, classes etc. and may therefore issue warnings when using the -w3 and/or -w4 option. Additionally, all occurrences of known classes are early bound, which will speed up the execution significantly, see also chapter LNG.2.11.6 .

Syntax 1:

CLASS prototyping is used if the class declaration is specified in another source file (or a library module).

Note, that the instance <**Name**> in the class PROTOTYPE has to match the <Name> of the instance declaration (in the CLASS statement, without case sensitivity, but at least in the first 10 significant characters). The order in which the instances are given does not matter. For additional info and arguments used, refer to the CLASS description.

Syntax 2: The CLASS METHOD prototyping is used

a. together with syntax 1, if the class declaration and theirs entities are specified in another source file (or a library module). In this case, all instances and access, assign, methods must also be declared with the same name as in the class declaration. Their order does not matter, but the class prototypes according to syntax 1 must be declared first. You may also #include the, by the FlagShip compiler automatically created, 'reposit.fh' file according to sect. FSC.1.4.2.

b. during method creation, when the method refers to a yet undeclared access, assign or usual method (forward prototyping). It is not necessary to prototype a method, which was formerly declared in the same source file, since the FlagShip compiler internally holds tables of the known classes (and its entities) encountered in the currently compiled source file. Otherwise, when an (yet) unknown method is invoked, the code for a run-time access is generated (late binding), which results in slower performance.

For additional info and arguments used, refer to the METHOD description.

Syntax 3: UDF prototyping is used for compile-time and run-time checking of the arguments being passed and of the returned values.

If the UDF return type is prototyped (e.g. PROTOTYPE MyUdf() AS LOGICAL), the compiler reports an error, if the result is assigned to a typed variable of an incompatible type, or if an invalid RETURN value was used within the UDF body. On the other hand, assigning an untyped UDF to a typed variable may result in a run-time error "attempt to assign <UDF-return-type> to fixed <vartype>". The assignment to an untyped, or to a typed AS USUAL variable is always accepted.

When also the UDF parameters are prototyped, the compatibility of the passed arguments are checked both at compile-time (on known types, e.g. LOCAL.. ..AS) and at run-time. It allows an early detection of passing wrong arguments or a wrong argument count, which mostly avoids a run-time error. The run-time check simplifies parameter validation, since only the specified <vartype> is accepted, otherwise a run-time error occurs.

Arguments:

AS <**type**> (proto)types the return value or the parameter to be fix and of the specified <type> only. If the AS <type> is omitted, the implicit USUAL type is assumed. The compatible types (see also LOCAL..AS) for return values and parameter prototyping are:

Prototype	Accepted variable or constant
C-like types	not allowed for UDF prototyping
ARRAY	ARRAY
CHARACTER	CHARACTER, PSZ, STRING
CODEBLOCK	CODEBLOCK
DATE	DATE
INTVAR	INTVAR, NUMERIC, all C-like types
LOGICAL	LOGICAL and INTVAR, NUMERIC, C-like types whereby 0 (zero) is converted to .F., all other values to .T.
NUMERIC	NUMERIC, INTVAR, all C-like types
OBJECT	OBJECT, <userClass>, <stdClass>
PSZ	CHARACTER, STRING, PSZ
SCREEN	SCREEN
STRING	CHARACTER, PSZ, STRING
USUAL	any type
<stdClass>	<stdClass>, OBJECT
<userClass>	<userClass>, OBJECT

Arguments:

<**paramList**> specifies one or more comma separated LOCALy scoped parameters, corresponding to the parameter list of the procedure, function, or method declaration.

Knowing the prototype of the UDF (or method), the FlagShip compiler will check the correspondence of the type and number of passed arguments in all subsequent occurrences of this UDF within the .prg file.

Note, that only the order, number (and type if given) of parameters have to match in the UDF declaration and the PROTOTYPE statement, the name of the parameter variables may differ.

Each parameter <parName> in the <paramList> of the FUNCTION, PROCEDURE or METHOD declarator and in the PROTOTYPE statement can be specified as

- <parName> : Untyped parameter, similar to a local variable. Only the parameter name is given, arguments of any type are accepted.

- <parName AS type> : Typed parameter, similar to a LOCAL ... AS variable. Only arguments of the specified <type> are accepted.
- **[**<parName>**]** or **[**<parName AS type>**]** : the square brackets [ ] specify optional parameters, which will accept arguments of the specified <type>, as well as NIL values. All arguments of optional parameters rightmost in the <paramList> may be omitted.
- **[...]** : Any number of optional, untyped parameters (accepted in the PROTOTYPE statement only). Useful e.g. for prototyping of functions with many arguments, e.g. written in Extend C API.
- <@parName> or <@parName AS type> or [<@parName>] : Automatic (implicit) parameter/ argument passing-by-reference, instead of the default passing-by-value. This may speed-up the execution significantly, especially on large strings. The compiler will not generate a temporary copy of the argument for parameters prefaced by the at-sign @. The UDF is called as if the argument is explicitly prefaced by the at-sign @ (see also LNG.2.3.2). **Warning**: all modifications of such implicit referenced parameters in the function body will also modify the incoming argument, including database fields and array elements (except constants).

For your convenience, the same parameter syntax may also be used in the FUNCTION, PROCEDURE or METHOD declarator. The FlagShip compiler will then produce the corresponding prototypes fully automatically when the compiler switch -ru is set.

Example 1:

Valid prototypes are:

```
PROTO udf1 () AS usual                 // no args accepted
PROTO udf2 ([...]) AS usual            // any no of args
PROTO udf3 (p1, p2 AS numer) AS logic  // 2 args required
PROTO udf4 (p1 AS char, ;              // 1 to 3 args,
           [p2 as logic], [p3]) AS nume //   at least one
PROTO udf5 (p1 AS char, ;              // at least 1st
           [p2 as logic], p3) AS numer  //    and 3rd req.
PROTO udf6 ([@p1 AS char], ;           // any number, but
           [...]) AS usual             // the 1st is char
PROTO udf7 (@p1, @p2 AS char) AS char  // 2 args, passed
                                       //    by reference
PROTO CLASS myClass
   EXPO var1  AS intvar
   EXPO name2 AS char
   HIDD invi3 AS usual
PROTO METH meth1 () CLASS myClass AS array
PROTO METH meth2 (p1, [@p2 AS char]) CLASS myClass AS object
PROTO ACCE name1 () CLASS myClass AS char
PROTO ASSI name2 (p2 AS usual) CLASS myClass AS char

FUNCT udf8(p1, @p2 AS char) AS usual // passed to reposit.
FUNCT udf9(p1, @p2 AS char)          // not passed to repos
```

Example 2:

Refer to the CLASS and METHOD description, section LNG.2.11.5 and the <FlagShip_dir>/system/smallrdd/smallrdd.prg file for examples of the CLASS prototyping.

Example 3:

Refer also to the "stdfunct.fh" file for prototypes of the standard FlagShip functions.

Classification:

programming, compiler/linker

Compatibility:

Prototyping is available in FlagShip only. VO manages it through the 'repository'. For compatibility to Clipper, you may specify

#ifndef FlagShip #command PROTOTYPE <x> => #endif

Related:

INSTANCE, METHOD, LOCAL..AS, FUNCTION, PROCEDURE, LNG.2.11

PUBLIC

Syntax:

```
PUBLIC <memvar> [:= <exp>] [, ... ]
```

or:

```
PUBLIC <array> [<dim>]
PUBLIC <array> [<dim1>,<dim2>,<dimN>]
PUBLIC <array> [<dim1>][<dim2>][<dimN>]
PUBLIC <array> := {<exp>,... }
```

Purpose:

Creates and initializes the specified memory variables or arrays in the PUBLIC class, i.e. to be visible for the whole application.

Arguments:

<**memvar**> is the list of variables or arrays to be created as PUBLICs. In this list, arrays and other variables can be interchanged. The name may be of any length, but only the first 10 characters are significant (see more LNG.2.6). Variable names in the FlagShip language are not case sensitive.

<**array**> is the name of the array to be created. The naming convention is the same as with <memvar>. The square brackets [ ] behind the <array> name do not in this case specify an optional argument, but are a required part of the syntax. The number of elements for each array dimension can be specified as [dim1, dim2,..,dimN] or [dim1][dim2][dimN]. The maximum number of dimensions and of the elements per dimension in FlagShip is 65535. Array elements can be handled like ordinary memory variables. Different elements of the same array can be of different types. Each element may contain another sub-array (non-symmetric structure), cf. LNG.2.6.4.

Options, Initializing:

<**exp**> is any valid FlagShip expression including a literal (constant) array to initialize the variable. If the initializer (:= <exp>) is not given, the variable is set to FALSE (.F.) (or all array elements) will be set to NIL. Initializing of a variable created by composed macro (e.g. PUBLIC var&macro := value) is not supported, but the sequence PUBLIC var&macro ; var&macro := value is ok.

The array elements can be declared and initialized with a starting value using an array (literal) constant (see LNG.2.7) including any valid expression and the assign := operator. The initialization will be done at variable creation time, i.e. when executing the PUBLIC statement.

Scope, Visibility:

PUBLIC variables have dynamic scope. These variables are visible for both hierarchically higher and lower modules starting at the time of the PUBLIC declaration. The PUBLIC variable can be later temporarily hidden using a PRIVATE, PARAMETERS, LOCAL, STATIC or GLOBAL declaration. The PUBLIC variable can be explicitly destroyed using CLEAR ALL, CLEAR MEMORY, or RELEASE.

For more information about variables, refer to the section LNG.2.6.

Description:

PUBLIC is an executable statement which creates and initializes a new variable or array in the dynamic scoping class. The PUBLIC statement is equivalent to the PRIVATE declaration on the highest program level (main).

An attempt to create a PUBLIC **variable** with the same name as an existing and visible PRIVATE variable is simply ignored. If the creation of a public **array** is requested, the previous PUBLIC or PRIVATE array with the same name is destroyed and replaced by the new one; if a dynamically scoped variable having the same name already exists, the new array declaration is ignored. Attempting to specify a PUBLIC variable that conflicts with a previous FIELD, LOCAL, STATIC or TYPED declaration of the same name results in a compiler error.

PUBLIC variables can be passed by value or by reference to other UDFs or UDPs called from within. In code blocks, only active PUBLIC (and PRIVATE) variables of the module where the block is **executed** are visible; see LNG.2.3.3.

Reserved Variables:

The following, reserved variables will be set up by the compiler and cannot be deleted via CLEAR MEMORY, but their contents can be redefined using an assignment.

PUBLIC FLAGSHIP : when the FlagShip compiler encounters such a declaration, it initializes it with the logical value TRUE (.T.) instead of FALSE. On the other hand, when compiling with Clipper, the variable remains FALSE. This allows a program to check which platform it is running and take different actions accordingly. Another possibility is to compile different blocks of code using the preprocessor directives #ifdef FlagShip ... #else ... #endif.

PUBLIC GETLIST [0] : an automatically created array to carry the GET objects for the command @...GET. As with all other PUBLIC arrays, this default array can be hidden via a PRIVATE, LOCAL or STATIC declaration (e.g. LOCAL GetList := {}) to create nested GET/READs to any level.

Example:

See also section LNG.9 for the compatibility notes.

```
PUBLIC FlagShip, Clipper
PUBLIC var1, subdir, arr1[5], arr2[10,10]
subdir = "D:\data\public\"
#ifdef FlagShip
# ifdef FS_WIN32
   ? "invoking CMD/DIR in FS/Windows"
   RUN ("DIR " + subdir + "*.* >tmp.out 2>&1")
# else
   subdir := "/home/data/public/"
   ? "invoking ls -l in Linux/Unix"
   RUN ("ls -l " + subdir + "* >tmp.out 2>&1")
# endif
#else
   ? "running under DOS with Clipper"
   RUN ("DIR " + subdir + "*.* >tmp.out")
#endif
```

```
? MEMOREAD("tmp.out")
IF !FILE(subdir + "address.dbf")
   alert("File " + subdir + "address.dbf n/a")
ENDIF
USE &subdir.address
```

Classification:

programming

Compatibility:

Initialization during the declaration is new in FS4. PUBLIC variables are available in all xBASE languages. Only FlagShip supports an unlimited number of variables, and up to 64k * 64k array element size.

Related:

DECLARE, PARAMETERS, PRIVATE, MEMVAR, FIELD, LOCAL, STATIC, GLOBAL

PROTECT PUBLIC

Syntax:

```
PROTECT PUBLIC <memvar> [:= <exp>] [, ... ]
```

Purpose:

Creates and optionally initializes protected public variable(s), which cannot be deleted but may be overwritten by any other value, as opposite to CONSTANT which cannot be overwritten later.

The scope and visibility is equivalent to PUBLIC variables.

Classification:

programming

Compatibility:

New in FS5

Related:

PUBLIC, CONSTANT, PRIVATE, DECLARE, STATIC

PUSH KEY
POP KEY

Syntax 1:

```
PUSH KEY [CLEAR]
```

Syntax 2:

```
POP KEY [ALL]
```

Purpose:

PUSH KEY and POP KEY is provided mainly for FoxPro compatibility

Description:

Syntax 1: PUSH KEY saves all ON KEY, ON KEY LABEL and SET KEY definitions on internal stack for later restoring by POP KEY. The optional clause "CLEAR" deletes all key assignments of ON KEY, ON KEY LABEL and SET KEY.

Syntax 2: POP KEY restores the last ON KEY, ON KEY LABEL and SET KEY structure previously saved by PUSH KEY.

Classification:

programming

Translation:

```
_PushKey(lClear)  or  SetKeySave(lClear)
_PopKey(lAll)     or  SetKeyRest() = _PopKey(.T.)
```

Compatibility:

New in FS5, compatible to FoxPro

Related:

ON KEY, OnKey(), SetKey(), SET KEY, SetKeySave(), SetKeyRest()

QUIT

Syntax:

`QUIT [<exitCode>]`

or:

`CANCEL`

Purpose:

Terminates program execution, closes all opened files, and returns control to Unix or Windows.

Options:

<**exitCode**> is optional numeric value returned by the application on exit. The default setting is 0. You alternatively may set the exit code by ErrorLevel(num) at any time before QUIT. The RETURN(num) in main module is equivalent to QUIT <num>.

Description:

CANCEL or QUIT are available from anywhere in a program to terminate execution and to return to the operating system. The same result is achieved if the RETURN command is used on the top level or the user aborts via the break key (^K or another defined with FS_SET("break")).

When the program terminates, all open files are closed and flushed to the disk. Active record/file locks are released.

When a new console window was created in X11 or Windows environment (e.g. for application running in terminal or basic mode without own console), a delay of 10 seconds occurs before the console window is closed. You may redefine this delay by setting

```
    _aGlobSetting[GSET_N_WAITCLOSEWIND] := 10  // this is default
```

See more details in ConsoleOpen(). This delay is disabled, when the application was compiled by using -io=b switch.

The return code is normally set to 0 or to 1..9 if the process ends with a fatal or run-time error. A user return code can be set with ERRORLEVEL() or the <exitCode> parameter of QUIT. The InitIoQuit() function (or your re-defined UDF) also sets exit code on user abort, see details in <FlagShip_dir>/system/initiomenu.prg and ErrorLevel()

Classification:

programming (and database)

Translation:

__QUIT([exitCode])

Compatibility:

QUIT <num> is available in VFS7 only.

Related:

RETURN, ^K abort, FS_SET(), SETCANCEL(), ERRORLEVEL()

READ

Syntax:

```
READ [SAVE]
        [ALIGN|NOALIGN]
        [SELECT <varC|posN>]
        [SKIPOVER|NOSKIPOVER]
        [EXITCHECK]
        [CLEAR|DESTROY]
        [CYCLE]
```

Purpose:

Activates the full-screen editing mode using a list of pending GETs (objects).

Options:

The **SAVE** clause retains the list of current GETs to enable editing the same GETs by issuing another READ. Without it, the current GETs are cleared when READ ends except when ReadSave(.T.) is called during the READ.

The **ALIGN** or **NOALIGN** clause temporarily overrides the current SET GUIALIGN setting. It specifies if the columns of @..SAY..GET should be aligned to the same virtual column position via the GuiAlign() function. The align apply mainly for GUI mode with proportional fonts. See also SET GUIALIGN and GuiAlign(), available in source in <FlagShip_dir>/system/getsys.prg.

SELECT <varC> or **SELECT <posN>** causes READ to start with GET item specified by variable name <varC>, or with item number <posN> 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.

SKIPOVER clause allows to skip over validated fields, **NOSKIPOVER** forces to stay in the field which does not meet the VALID criteria. Skipping over a field can mostly apply in GUI by mouse click on different field or widget in the GetList.

If **EXITCHECK** is specified, READ will check all VALID conditions at exit and stay on the unsatisfied field even if SKIPOVER was set.

The **CLEAR** or DESTROY clause clears GUI Get widgets on exit. If not specified, the widget remain visible same as in Terminal i/o.

With the **CYCLE** clause, the READ will not be terminated when moving forward over the last GET item, or backward over the first GET. It instead cycles from the last to first, and from first to last item. The READ CYCLE will be terminated by ESC, Ctrl-W or CLEAR GETS.

Description:

The READ command enables full-screen editing using the pending list of GET fields stored in the GETLIST[] array since the most recent CLEAR, CLEAR GETS, CLEAR ALL or READ was executed.

Each GET field definition consists of an object, where the screen coordinates, formatting, color, and pre- and post validation conditions are stored. These values are specified by the @...GET command or assigned to the object. The user can edit, re-enter or confirm the field data. Using the navigation keys, the user can move between fields. The content of the field-editing buffer is stored in an associated memory or FIELD variable.

The execution starts with the first pending GET field in the current GETLIST array and is finished when all available fields are processed or a termination key is pressed. If there is a format procedure active (see SET FORMAT), READ executes that procedure prior to entering the full-screen editing mode.

When the current GET field is finished (by filling the rightmost column and SET CONFIRM is OFF) or by pressing the GET or READ termination key, control is passed to the optional plausibility checking specified by the RANGE and/or VALID clause. If FALSE is returned from the VALID condition or the value is out of the RANGE boundary, the cursor remains within the current field, allowing the user to correct his entry. The ESC key leaves READ without storing the current field and without plausibility checking, if SET ESCAPE is ON.

To modify active GET field during the READ execution, use WHEN or VALID clause of @..GET, or SET KEY function. See "Validity" chapter below and examples in <FlagShip_dir>/examples/getvalid*.prg

READ is finished when the appropriate termination key is encountered, or the last pending GET field is terminated, or CLEAR GETS executed during the READ process. Thereafter, the GET fields remains yet visible, except READ CLEAR was used. To overwrite yet visible but inactive GET fields by @..SAY, ?, ?? or other display commands, clear the screen area first by @ row,col CLEAR TO row,col or the whole screen by CLS, CLEAR SCREEN or CLEAR.

Full-screen **Navigation Keys** for GETs and READ:

Key		Action	
Cursor <-	ctrl-S	moves cursor one position left	
Cursor ->	ctrl-D	moves cursor one posit. right	
Cursor Up	ctrl-E	previous GET field	
shift-TAB		previous GET field	
Cursor Down	ctrl-X	next GET field	
Enter	ctrl-M	next GET field	
TAB	ctrl-I	next GET field	
Home	ctrl-A	first character in field	*
End	ctrl-F	last character in field	*
ctrl-Cursor <-	ctrl-Z	previous word in field	
ctrl-Cursor ->	ctrl-B	next word in field	
ctrl-Home	ctrl-]	first GET field of READ	
ctrl-End	ctrl-W	last GET field of READ or exit *	*
Left-Mouse-Button		on other field: select field **	*
Mouse-Wheel		previous/next GET field **	*

* Quick keys: the first instance of the [Home] or [End] key pressed moves the cursor to the first or last valid character; a repeated key moves the cursor to the start or end of the field.

** ctrl-W and ctrl-End behavior depends on settings: it exits READ with CYCLE clause, or _aGlobSetting[GSET_L_READ_CTRLW_EXIT] := .T. otherwise it skips to last GET field of READ. See Tuning below.

*** GUI mode only.

Edit Keys for the GET field:

Key		Action	
Insert	ctrl-V	Insert mode on/off	*
Delete	ctrl-G	Delete character at cursor	
Backspace <=	ctrl-H	Delete previous character	
	ctrl-T	Delete word right	
	ctrl-Y	Delete rest of the field	
	ctrl-U	Undo, restore original field	
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	**

* the insert mode continues to remain active for the next GET or READ.

** GUI mode only. See description in "cut-and-paste" below.

GET and READ **Termination Keys** initiate a post validation and store the GET field contents into an associated memory or FIELD variable:

Key		Action	(S = save,
			V = post-valid,
			T = terminate READ)	

Cursor up	ctrl-E	being in first field	SV T	*
		otherwise: prev. field	SV	
Cursor down	ctrl-X	being in last field	SV T	*
		otherwise: next field	SV	
Enter, Return	ctrl-M	being in last field	SV T	
		otherwise: next field	SV	
ctrl-Home	ctrl-]	go to first field	SV	
ctrl-End	ctrl-W	terminates READ or last GET	SV T	**
PgUp	ctrl-R	terminates READ	SV T	
PgDn	ctrl-C	terminates READ	SV T	
Escape (Esc)		terminates READ	T	*

* termination keys: READ termination depends on the current setting of READEXIT() and SET ESCAPE.

** ctrl-W and ctrl-End behavior depends on settings: it exits READ with CYCLE clause or _aGlobSetting[GSET_L_READ_CTRLW_EXIT] = .T. otherwise it skips to last GET field of READ. See Tuning below.

Terminating READ is also possible by executing the BREAK, CLEAR, CLEAR GETS, or CLEAR ALL command from a SET KEY procedure or from a user defined function initiated by the VALID clause.

Validity, Plausibility:

Each GET field can include a pre-valid and/or plausibility (post- valid) condition checking by using the @...GET clauses WHEN, VALID or RANGE.

Before the user can enter a GET field (object), control passes to the associated WHEN <condition> if one is given. If the condition returns TRUE, editing is enabled; otherwise, the field is skipped. When the user presses a GET exit key, control passes to the associated RANGE or VALID post-condition if one has been specified. If either of the conditions return FALSE, or the numeric value is out of the RANGE boundary, control remains within the current GET field until a valid value is entered or the user presses the Esc key. If both clauses are specified, RANGE is performed first.

See WHEN and VALID examples below, in @..GET and in <FlagShip_dir>/examples/getvalid*.prg

Update-Status:

When any GET field is changed by the user (but not in a VALID or SET KEY function), the UPDATED() function will return TRUE.

Nested Reads:

By executing a VALID function or SET KEY procedure (background routine) when in READ, another set of GET..READ may be temporary initiated, if a LOCAL, STATIC or PRIVATE array GETLIST[0] is created there. All subsequent @...GETs and READ will refer to this local set of GET fields, until the procedure or function returns control back to the active READ.

Redirection:

When one of the navigation key is redirected via SET KEY or ON KEY or SET FUNCTION, the redirection is executed instead of the default behavior.

User-modifiable READs and Objects:

During READ execution, the current GET object may be determined by using GETACTIVE(); the associated export variables (including the editing buffer) can be revoked or changed within a background routine. The type of the current GET variable may not be changed without executing `GETACTIVE():SETFOCUS()`.

For more programming control over the READ command, you may modify <FlagShip_dir>/system/**getsys.prg**. Other user-defined READs may also be performed if the procedure name is assigned to the `get:GETREADER` export variable.

Multiuser:

If one or more GETs refer to database fields, RLOCK() or FLOCK() in the associated working area must be executed before the READ statement. The database should be UNLOCKed after READ. See also LNG.4.8 and Timer paragraph below.

Unicode:

To display multi-byte characters (known as Unicode, used for Asian languages) during the READ input, enable it by SET MULTIBYTE ON either global or latest before the READ statement, or use the MULTIBYTE clause of @..GET. Also the font needs to be set correspondingly to display Unicode. See further description in @..[SAY]..GET and section LNG.5.4 for Unicode details.

Copy and Paste:

Depending on the currently used i/o mode (GUI, Terminal), you may insert/overwrite characters in the GET field by cut/copy 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(s).

To **copy** part of the GET field 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 (like Ctrl-C, right or middle-mouse-button menu etc).

To **paste** clipboard at current GET field 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 GET content is partially overwritten by the text from clipboard (same as you would type it).

With enabled Unicode support, you may use popy-and-paste as well, also from text document encoded in UTF-8. See LNG.5.4 for details.

Tuning:

The READ is fully tunable, since available in source code in the *<FlagShip_dir>/ system/getsys.prg* file. You may copy it to your working directory, and compile according to the header in source file, then link with your application.

You additionally may tune the standard READ behavior by following switches:

The copy and paste buttons or keys are user modifiable by assigning corresponding INKEY() value (see inkey.fh for K_* manifests) to:

```
  _aGlobSetting[GSET_G_N_GET_COPY1 ] := K_MBUTTONDOWN   // copy
  _aGlobSetting[GSET_G_N_GET_COPY2 ] := K_ALT_C         // copy
  _aGlobSetting[GSET_G_N_GET_PASTE1] := K_SH_MBUTTONDN  // paste
  _aGlobSetting[GSET_G_N_GET_PASTE2] := K_ALT_V         // paste
```

where the default settings (set in initio.prg) are shown here. 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. Of course, you also may modify the behavior directly in *<FlagShip_dir>/system/getsys.prg* source, see above.

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.

In both GUI and Terminal i/o, you may specify the behavior of Home and End key, whether this keypress should skip to first/last valid character in field, or to the field begin/end

```
   _aGlobSetting[GSET_L_GET_HOME2CHAR ] := .T. // default = 1st
   _aGlobSetting[GSET_L_GET_END2CHAR  ] := .T. // default = last
```

however a double press on the Home/End key will skip cursor to begin or end of the READ field, and vice-versa.

In GUI, you may modify the behavior of mouse click on READ field: Should mouse click in current field execute oGet:Home() ?

```
   _aGlobSetting[GSET_G_L_GET_MOUSEHOME]:= .F.  // default = no
```

Should mouse click in another field activate this field and perform there oGet:Home() ?

```
   _aGlobSetting[GSET_G_L_GET_MOUSENEW] := .T.  // default = yes
```

Should mouse click allow position behind the last valid char ?

```
   _aGlobSetting[GSET_G_L_GET_MOUSEOUT] := .F.  // default = no
```

You also may disable mouse wheel by assigning
```
   _aGlobSetting[GSET_L_READ_ACCEPT_WHEEL] := .F. // default = .T.
```
The Ctrl-W and Ctrl-End keys terminates READ by default (same as in VFS6 and Clipper 5.x without setting #define CTRL_END_SPECIAL). To re-define them to skip to last GET item of READ (i.e. to behave same as Clipper'87 or FS4 or VFS5), assign
```
   _aGlobSetting[GSET_L_READ_CTRLW_EXIT] := .F.  // default = .T.
```
which however may be overloaded by the CYCLE clause, which always causes exit from READ.

The SET KEY redirection forces to re-display the visible GET field. You may avoid it by assigning
```
   _aGlobSetting[GSET_L_READ_INKEY_PLAIN] := .F. // default = .T.
```
which then uses InkeyTrap() instead of Inkey()

If you wish to skip to next/previous field by using KEYBOARD value pass KEYBOARD chr(K_TAB) or KEYBOARD chr(K_SH_TAB) which is considered also for Listbox, Checkbox etc. where other keys are eaten. You may change this default by assigning
```
   _aGlobSetting[GSET_A_READ_SKIP] := {K_TAB, K_SH_TAB}  // default
```
Another alternative to skip GETs is the function ReadSelect().

In GUI mode, the default cursor mode in GET/READ is vertical bar in front of the current character, independent of the insert or overwrite mode. You can change the mode and/or color by oGet:SetCursor() or globaly (before next READ) by assigning mode and colors for the overwrite and insert cursor mode (here defaults):
```
   _aGlobSetting[GSET_A_READ_GUICURSOR] := {{0,{0,0,0}},{0,{0,0,0}}}
```
according to oGet:SetCursor() described in section OBJ.Get

In GUI mode, the GET is a widget (or control in MS terminology, see LNG.5.3) which remain displayed after finishing READ (w/o the CLEAR clause). This prevents over-writing of GETs by @..SAY after READ and in some circumstances it may also result in collecting memory. You may re-display all the GETs by SAY when finishing READ by assigning
```
   _aGlobSetting[GSET_L_READ_REDISPL] := .T.   // default is .F.
```
latest before executing the READ statement. This setting apply for @..GET fields in READ but not for other widgets like ListBox, CheckBox, RadioButton, PushButton etc - you may clear them by @..CLEAR..

Additional tuning is described in the @..GET command.

Timer:

You may abort the READ after specified time period by issuing e.g. KeySec(K_ESC, 900) before READ and KeySec(.F.) thereafter. This simulates press of ESC key after 15 minutes (of inactivity), e.g. to unlock the record for editing by others. You also may use KeyTime(...) for similar purposes. These Key*() functions are available in FS2 Toolbox, see section FS2:Date/Time/Triggers for details.

Example 1:

Access to database fields in multiuser mode:

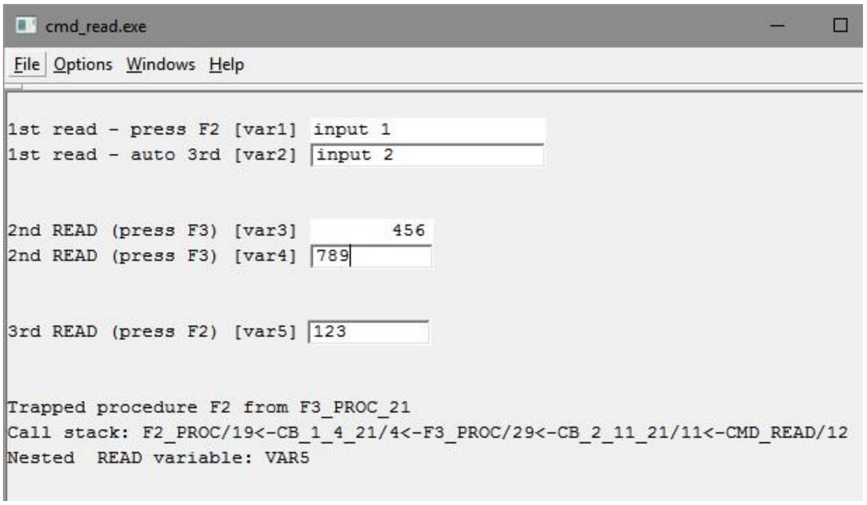
```
FIELD name, city
USE address SHARED
@ 1,0 say "Name   " GET name
@ 2,0 say "City   " GET city
DO WHILE !RLOCK() ; ENDDO          // lock .dbf fields
READ                               // modify & replace dbf fields
UNLOCK                             // unlock in multiuser
```

Example 2:

Nested GET/READs to several levels:

```
SET FONT "Courier", 10
oApplic:Resize(25,80,,.T.)
SET KEY K_F2 to f2_proc
SET KEY K_F3 to f3_proc
LOCAL   var1 := var2 := space(20)
PRIVATE var3 := var4 := var5 := 0
@ 1, 0 SAY "1st read - press F2 [var1]" GET var1
@ 2, 0 SAY "1st read - auto 3rd [var2]" GET var2 ;
           VALID !empty(var2) .and. ;
                 f3_proc(PROCNAME(), PROCLINE(), READVAR() )
READ
setpos(15,0)
wait

PROCEDURE f2_proc (procName, procLine, actVarName)
LOCAL myvarname := READVAR()
LOCAL getlist := {}                // required for nesting
output (procName, actVarName)      // or myvarname
@ 5, 0 SAY "2nd READ (press F3) [var3]" GET var3
@ 6, 0 SAY "2nd READ (press F3) [var4]" GET var4
READ
RETURN

FUNCTION f3_proc (procName, procLine, actVarName)
LOCAL getlist[0]    // required for nesting, same as getlist := {}
output (procName, actVarName)
@ 9, 0 SAY "3rd READ (press F2) [var5]" GET var5
READ
RETURN var5 > 0

STATIC FUNCTION output (procName, actVarName)
@ 12,0 SAY "Trapped procedure F2 from " + procName
@ 13,0 SAY "Call stack: " + procstack(1)
@ 14,0 SAY "Nested  READ variable: " + actVarName
RETURN NIL
```

Output:



```
1st read - press F2 [var1] input 1
1st read - auto 3rd [var2] input 2


2nd READ (press F3) [var3]       456
2nd READ (press F3) [var4] 789


3rd READ (press F2) [var5] 123


Trapped procedure F2 from F3_PROC_21
Call stack: F2_PROC/19<-CB_1_4_21/4<-F3_PROC/29<-CB_2_11_21/11<-CMD_READ/12
Nested  READ variable: VAR5
```

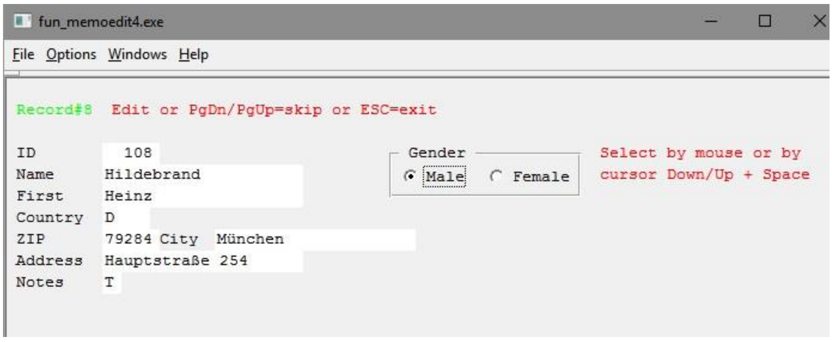
Example 3:

Edit/skip simplified address database (with ASCII/PC8/OEM charset), using also @..GET..RADIOGROUP and MEMOEDIT(). For browsing and editing, see FUN.DBEDIT() Example 1.

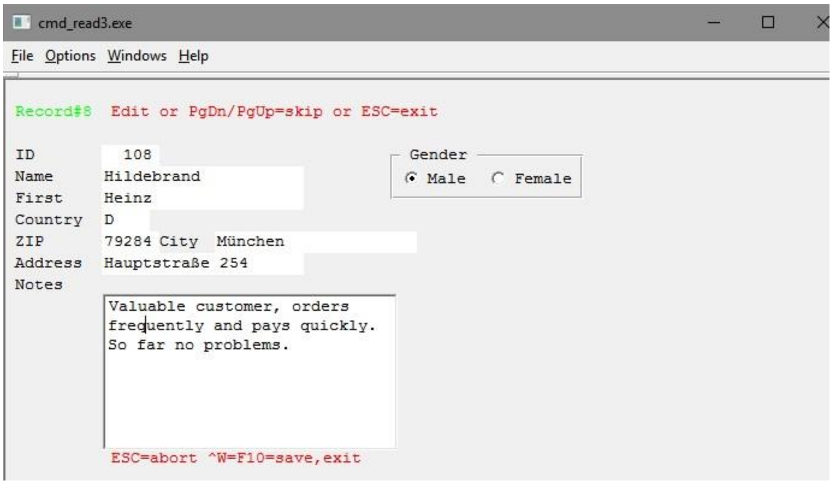
```
LOCAL nID, nGender, lNote
set font "Courier",10
oApplic:Resize(25,85,,.T.)

if !file("address.dbf")
  dbcreate("address",{{"ID","N",5,0}, {"Male","L",1,0}, ;
                      {"Name","C",20,0}, {"First","C",20,0}, ;
                      {"Country","C",4,0}, {"ZIP","N",5,0}, ;
                      {"City","C",20,0}, ;
                      {"Address","C",20,0}, {"Memo","M",10,0} } )
endif
USE address NEW SHARED
if !used()
   alert("sorry, database address is not available")
   quit
endif
SET SOURCE ASCII         // edit ASCII/PC8/OEM data

while .T.
   if !eof()
     nGender := if(MALE, 1, 2)
     nID := ID
     lNote := !empty(MEMO)
   else
     append blank
     nGender := 1
     nID := lastrec()
     lNote := .T.
   endif
   @ 1, 1 say "Record#" + ltrim(recno())     GUICOLOR "G+"
   ?? "  Edit or PgDn/PgUp=skip or ESC=exit" GUICOLOR "R+"

   @ 3, 1 say "ID       " GET nID pict "99999"
   @ 3,40, 5,60 GET nGender RADIOGROUP ;
        {Radiobutton{-1,-1,"Male"}, Radiobutton{-1,-10,"Female"}} ;
         Caption "Gender" ;
         ToolTip "Select by mouse or cursor Down/Up + Space" ;
         WHEN RadioInfo(.T.,3,62) VALID RadioInfo(.F.,3,62)
   @ 4, 1 say "Name    " GET Name
   @ 5, 1 say "First   " GET FIRST
   @ 6, 1 say "Country " GET COUNTRY
   @ 7, 1 say "ZIP     " GET ZIP Pict "99999"
   @ 7,17 say "City "    GET CITY
   @ 8, 1 say "Address " GET ADDRESS
   @ 9, 1 say "Notes   " GET lNote VALID myMemo(9,10, 17,40 )
   // _aGlobSetting[GSET_L_READ_INKEY_PLAIN]:= .F. //accept SET KEY
   READ
   if lastkey() == K_PGUP
     SKIP -1
     loop
   elseif lastkey() == K_PGDN
     SKIP 1
     loop
   endif
   // _aGlobSetting[GSET_L_READ_INKEY_PLAIN] := .T. // set default
   if lastkey() != K_ESC
      replace MALE with (nGender == 1)
   else
      if alert("Abort editing?",{"Yes","No"}) == 1
        exit
      endif
   endif
enddo
wait

FUNCTION myMemo(ytop, xtop, ybot, xbot)
LOCAL text, actscreen := SAVESCREEN (ytop, xtop, ybot, xbot)
set key K_F10 to exitMemo
@ ytop, xtop CLEAR TO ybot, xbot
@ ytop, xtop TO ybot, xbot DOUBLE
@ ybot, xtop +1 SAY "ESC=abort ^W=F10=save,exit" GUICOLOR "R+"
text := MEMOEDIT (MEMO, ytop+1, xtop+1, ybot-1, xbot-1, .T.)
if lastkey() != K_ESC .and. !(text == MEMO)
  REPLACE Memo with text
endif
lastkey(.T.)        // clear LastKey() buffer
set key K_F10 to    // disable redirection
RESTSCREEN (ytop, xtop, ybot, xbot, actscreen)
RETURN .T.

FUNCTION exitMemo()
KEYBOARD K_CTRL_W   // simulate Ctrl-W by F10
return
```

```
function RadioInfo(what,row,col)
@ row,col clear to row+1,maxcol()-1
if what
   @ row,col   say "Select by mouse or by"  GUICOLOR "R+"
   @ row+1,col say "cursor Down/Up + Space" GUICOLOR "R+"
endif
return .T.
```

Output:



```
Record#8  Edit or PgDn/PgUp=skip or ESC=exit

ID         108
Name      Hildebrand
First     Heinz
Country   D
ZIP       79284 City  München
Address   Hauptstraße 254
Notes     T

Select by mouse or by
cursor Down/Up + Space
```



```
Record#8  Edit or PgDn/PgUp=skip or ESC=exit

ID         108
Name      Hildebrand
First     Heinz
Country   D
ZIP       79284 City  München
Address   Hauptstraße 254
Notes
          Valuable customer, orders
          frequently and pays quickly.
          So far no problems.

          ESC=abort ^W=F10=save,exit
```

Example 4:

For more examples see the section (CMD) @..SAY..GET.

Classification:

programming

Class:

uses GET class, prototyped in <FlagShip_dir>/include/getclass.fh

Compatibility:

The use of objects is compatible to C5. In Clipper, the GET class cannot be inherited to user defined class. Clipper supports only plain GET fields.

The ALIGN, NOALIGN, SKIPOVER, NOSKIPOVER, EXITCHECK, CLEAR, DESTROY clauses are new in VFS5, MULTIBYTE and cut-and-paste support is new in VFS6, tuning is available since VFS7.

Source:

<FlagShip_dir>/system/getsys.prg

Translation:

```
READ          =>    READMODAL(GetList) ; GetList := {}
READ SAVE     =>    READMODAL(GetList)
```

Related:

@...GET, CLEAR GETS, ReadGetPos(), ReadSelect(), ReadExit(), ReadInsert(), ReadKey(), ReadKill(), ReadModal(), ReadSave(), ReadUpdated(), ReadVar(), SET FORMAT, SET KEY, SET MULTIBYTE, UNLOCK, LastKey(), NextKey(), Flock(), Rlock(), OBJ.Get

RECALL

Syntax:

```
RECALL [<scope>]
       [FOR <condition>]
       [WHILE <condition>]
```

Purpose:

Reinstates DELETEd records in the current working area. If the record was not deleted, no action is performed.

Options:

<**scope**> is the part of the current database file to be undeleted. The default scope is the current record if a condition is not specified, or ALL if a condition is specified.

<**condition**>**:** The FOR clause specifies that the set of records meeting the condition within the given scope are to be recalled. The WHILE clause stops recalling when the first record not fulfilling the condition is reached.

Description:

The deleted records are invisible when SET DELETED is ON and the database pointer was moved. To reach deleted records, use GOTO or SET DELETED OFF.

Multiuser:

RLOCK() is required when recalling one record, while FLOCK() when <scope> or <condition> is used. Otherwise, AUTOxLOCK() is used, when SET AUTOLOCK is enabled (the default).

Example:

```
USE employee
DELETE ; ? DELETED()                    &&   .T.
RECALL ; ? DELETED()                    &&   .F.
SKIP
? DELETED()                             &&   .F.
RECALL ; ? DELETED()                    &&   .F.
```

Classification:

database

Compatibility:

The automatic lock is not available in Clipper. FlagShip's autolock is similar to FoxPro and VO.

Translation:

```
RECALL       => DBRECALL()
RECALL [..]  => DBEVAL ({|| DBRECALL()}, [{for}],[{while}],;
                          [next], [rec], [.rest.] )
```

Related:

DELETE, PACK, SET DELETED, SET AULTOLOCK, ZAP, DELETED(), oRdd:RECALL()

REFRESH

Syntax:

REFRESH

Purpose:

Refreshes the screen contents from the last valid output buffer in terminal i/o mode.

Description:

In the Unix multiuser/multitasking environment, a screen output from different programs can be re-routed to one physical screen, which may garbage the output and deviate from the logical screen image buffers of the FlagShip application.

Note: To avoid long transfer time (e.g. on serial connected terminals), the curses library optimizes the output, displaying the changed characters only. The required parts of the curses library are linked into the FlagShip compiled executable; see section SYS.

By-passing the curses in any way (e.g. using #Cinline printf() output, activating other virtual shell windows or sessions, rerouting the output to /dev/tty.., printing from a child program etc.) may cause unpredictable results of subsequent screen output.

In such a case, use the REFRESH command or REFRESH() function to re-display the current FlagShip output and reset the correct cursor coordinates. Executing the sequence SAVE SCREEN ... "odd output" ... RESTORE SCREEN will not cause the same effect as REFRESH in all cases, but in RUN only.

Example:

```
SETPOS (10,5)
?? "Now, the directory is listed:"
y1 := ROW(); x1 := COL()                    // 10, 34
RUN MESSAGE "press any key" ls -l *
INKEY (0)
y2 := ROW(); x2 := COL()
REFRESH
? "old:", y1, x1                            // 10  34
? "new:", y2, x2                            // 10, 34
```

Classification:

screen oriented output

Compatibility:

The command is available in FlagShip only.

Translation:

REFRESH( )

Related:

REFRESH(), SAVE/RESTORE SCREEN, RUN

REINDEX

Syntax:

`REINDEX [EVAL <expL1> [EVERY <expN2>]]`

Purpose:

Rebuilds all open indices in the current working area.

Options:

EVAL <**expL1**> specifies a condition (similar to the WHILE <condition>, see the general command description), that may be executed at a specific record interval given by the **EVERY** <**expN2**> clause. The <expL1> must return TRUE to continue reindexing. The EVAL clause may be used, for example, to monitor the progress of indexing, with a UDF. If <expN2> is not specified, the default value is one (each record).

Description:

REINDEX performs the same action as INDEX ON..., but uses the index criteria already stored in the index header. Therefore, if the index file is corrupted, or the database structure was changed, the INDEX ON command should by used.

The REINDEX command is generally used to update indices which were not assigned to the database during its modification or appending. See also FlagShip's integrity checking in INDEX ON and INDEXCHECK().

REINDEX obeys the UNIQUE and ASCEND/DESCEND status as well as to the FOR <condition> as first created with INDEX ON. The current SET UNIQUE program status in 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.

Multiuser:

Exclusive access to the required database must be acquired by USE...EXCLUSIVE. In a multiuser environment, the time-consuming REINDEX can be omitted entirely, if all relevant index files are **always** assigned to the open databases. To select the required index file, use the SET ORDER command.

Example 1:

```
USE employee INDEX name
REPLACE ALL salary WITH salary + 100
SET INDEX TO id, birthdate, salary, name
REINDEX
```

Example 2:

Report the percentage of the reindex process:

```
LOCAL count, perc := 0
USE address NEW EXCLUSIVE
count := LASTREC()
SET INDEX TO adrname
REINDEX EVAL mydisplay(perc++) EVERY INT(count/100)

FUNCTION mydisplay (out)
@ 20,10 say "Reindexing, " + STR(out,3) + "% ready"
RETURN .T.
```

Classification:

database

Compatibility:

The index structure depends on the used RDD. The default driver DBFIDX uses special index files named .idx, which are not compatible to Clipper's .NTX or dBASE .NDX files. The internal structures of the index files and the locking mechanism are not compatible in these different dialects.

The EVAL and EVERY clause is new in FS4. Integrity checking is available with the FlagShip default driver only.

Translation:

```
REINDEX             => DBREINDEX()
REINDEX [EVAL...]   => ORDCONDSET (,,,, {||eval}, every)
                       ORDLISTREB ()
```

Related:

INDEX, PACK, SET INDEX, SET EXCLUSIVE, SET UNIQUE, SET ORDER, USE, INDEXCHECK(), INDEXNAMES(), INDEXDBF(), ISDBEXCL(), oRdd:REINDEX()

RELEASE

Syntax 1:

`RELEASE <memvarList>`

Syntax 2:

`RELEASE ALL [LIKE | EXCEPT <skeleton>]`

Purpose:

Deletes specified PRIVATE and PUBLIC memory variables.

Arguments:

<**memvarList**> is the list of variables to be deleted.

ALL deletes all visible variables in the dynamic scope.

<**skeleton**> is a wildcard mask (* and ? are supported) which specifies a group of variables to delete (ALL LIKE) or not to delete (ALL EXCEPT).

Description:

The RELEASE command performs different actions, depending on how it is specified:

- On syntax 1, the most recently declared variables and arrays are deleted whether PUBLIC or PRIVATE.
- On syntax 2, the scope of deleting becomes the current procedure level, and it can be narrowed if a wildcard is specified. Only PRIVATE and autoPRIVATE variables created in the current procedure are affected; a NIL value is assigned to the specified variables.

It is not necessary to RELEASE private (declared or automatic) variables before leaving a PROCEDURE or a FUNCTION. They will be released automatically.

LOCAL, STATIC and TYPED variables are not affected by the RELEASE command. Local variables are released automatically when the procedure or UDF where the variables were declared terminates. Static variables cannot be released since they exist for the duration of the program.

Example:

```
PUBLIC v1,v2,v3,name
STORE "John" TO v1,v2,v3,name
RELEASE ALL LIKE v*
? TYPE("v1")                          // U
? TYPE("name")                        // C
```

Classification: programming

Compatibility:

The behavior of the syntax 2 in FS4 and C5 differs slightly from and C87. In FlagShip, any number of variables is supported, so the RELEASE is not needed at all.

Related:

CLEAR ALL, CLEAR MEMORY, PRIVATE, PUBLIC, RESTORE, SAVE, LOCAL, STATIC, GLOBAL

RENAME ... TO

Syntax:

`RENAME <file1> TO <file2> [WITHMSG]`

Purpose:

Gives a file a new name.

Arguments:

<**file1**> is the name of the file to be renamed. Standard Unix and Windows wildcards are supported (see "man mv").

<**file2**> is the new name for the file. If only the path (except the dot . alone) is specified, the file(s) from <file1> are **moved** to the path given in <file2>.

Both <file1> and <file2> can be given as parenthesed (<expC>). Both file names have to include the extension and can optionally be preceded by a path designator.

Option:

<**WITHMSG**> if specified, run-time warning message is displayed on failure. Default is no warning message.

Description:

RENAME is a file command that changes the name of a specified file to a new name, very similar to the Unix command "mv", or "REN" of MS-Windows. This command does not use SET DEFAULT nor SET PATH.

If the <file2> exists, it is overwritten without any warning. The success or error may be checked using DOSERROR() or FILE().

Both <file1> and <file2> (if they exist) must be closed before renaming or moving. Attempting to rename an open file will produce unpredictable results.

When a database file is RENAMEd it is also necessary to RENAME the associated memo .dbt file.

Example:

```
? FILE("prices.dbf")                   && .T.
? FILE("old.dbf")                      && .F.
RENAME prices.dbf TO old.dbf WITHMSG
? FILE("old.dbf")                      && .T.
RENAME "[a-c]*.db*" TO /usr/myname     && move them
```

The same action may be also done with:

RUN mv prices.dbf old.dbf RUN ("mv [a-c]*.db* /usr/myname 2>/dev/nul")

Classification:

system, file access

Compatibility:

The RENAME command is equivalent to the Unix command "mv" or the similar DOS command "REN". The usage of wildcards, WITHMSG clause and the DOSERROR() checking is available in FlagShip only.

RENAME will be affected by the settings for the automatic path, pathname 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.

Translation:

```
FRENAME ("fileFrom", "fileTo", [.msg.])
```

Related:

FRENAME(), COPY FILE, ERASE, RUN, DOSERROR(), FILE(), FS_SET(), Unix: mv, Windows/DOS:RENAME

REPLACE ... WITH

Syntax:

```
REPLACE
      [<scope>]
      <field1> WITH <exp1>
         [, <alias> -><field2> WITH <exp2>]
         [, <field3> WITH <exp3>,...]
      [FOR <condition>]
      [WHILE <condition>]
```

Purpose:

Puts the results of evaluating the given expressions into the specified database fields.

Arguments:

<**field**> is the name of the field to change. The field can be of any type.

<**exp**> is the expression to REPLACE with.

Options:

<**alias**> has to be specified if a field belongs to a working area other than the current one.

<**scope**> is the portion of the current database file to REPLACE. The default is the current record. Specifying a <condition> changes the default to ALL.

FOR <**condition**> specifies the conditional set of records to REPLACE within the given scope.

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

Description:

REPLACE is a database command that assigns new values to the contents of one or more field variables in the current record in the specified working areas. The target field can be character, date, logical, memo, or numeric. REPLACE automatically updates all indices assigned to the specified working area.

REPLACE performs the same function as the assignment operator (:= or =) on aliased or as FIELD declared variables.

Note: replacing a field which is a part of the current index key expression may change the relative position of the record within the index file. Therefore, replacing a key field within a <scope> (like REPLACE ALL or NEXT <n> or REST etc.) or with FOR/WHILE clause may be hazardous on such active index, because this will often not replace all expected records in the <scope>. The reason is the SKIP to next record in <scope> according to the (permanently changed) index sequence order. The solution is the sequence n = INDEXORD() ; SET ORDER TO 0 ; REPLACE <scope> ... ; SET ORDER TO (n), or the similar but much less effective CLOSE INDEX, REPLACE <scope>... and then SET INDEX TO... plus REINDEX.

Sizes and Special characters, Tuning:

● In **"C" (character) fields**, any string containing ASCII character values 0..255 is accepted, also embedded zero (0x00) bytes. The size of character field is fix, the max size is 64 Kbytes. If the trimmed <exp1> is longer than field size, FlagShip raises run-time-error message (RTE 205, loosing data). To avoid this warning, assign

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

FlagShip will then silently truncate (by loosing) the rest of data. Use this switch for backward compatibility to unmodified Clipper or older FlagShip sources which silently truncates the rest.

● In **"N" (numeric) fields**, the stored value is rounded if necessary. FlagShip raises run-time error message (RTE 205, loosing data) when the <exp1> value = <intPart>.<deciPart> overflows the field size:

* for N<FldSize>.<FldDec> field when the <intPart> (plus '-' sign if applicable) is greater than <FldSize> - <FldDec> -1, and
* for N<FldSize>.<0> field when the <intPart> (plus '-' sign if applicable) is greater than <FldSize>.

Selecting "ignore" within the dialog, 0.0 is stored in the field. **Tuning:** On special needs, you may assign

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

to avoid this run time message, FlagShip then tries to use the <FldDec> part also for <intPart> if possible, e.g.

```
<exp1> = -1234.67812 and N8.2 field: stored = -1234.68 (ok)
<exp1> = 12345.78912 and N8.2 field: stored = 12345.79 (ok)
<exp1> = 123456.8912 and N8.2 field: stored = 123456.8
<exp1> = 1234567.912 and N8.2 field: stored = 1234567.
<exp1> = 12345678.12 and N8.2 field: stored = 12345678
<exp1> = 123456789.1 and N8.2 field: stored = 12345678 (!!)
```

so on overflow you will loose precision, or in worst case store incorrect data, as the example shows. Therefore use this tuning setting with care.

● **Date fields "D"** are always stored within 8 bytes in YYYYMMDD format, independent from the current state of SET DATE or SET CENTURY.

Logical fields "L" are stored in 1 byte as "T" or "F".

● In **"M" (memo) fields**, the 10 byte value points to location in .DBT or .FPT file where the data are stored. FlagShip supports both DBT and FPT files (data structure), the kind of memo file is determined from the database header. For new database, you may decide which kind is used by SET MEMOFILE TO DBT (default) or SET MEMOFILE FPT, see further details in the DbCreate() function.

- in **.DBT** file, you can store any string containing ASCII character values 1..255, except the CHR(0) = 0x00 and CHR(26) = 0x1A characters, which terminates the memo field. If these chars are used in the saved data, use MemoEncode() to store such strings in the memo field and MemoDecode() to read it from. The memo field is of variable size (in 512 bytes segments) and supports up to 2 GBytes for each data (some xBase drivers supports only 64 Kbytes).

- in **.FPT** file, you can store any string containing ASCII character values 0..255. The memo field is of variable size (usually in 64 bytes segments, modifiable) and supports up to 65535 characters (64 KB) each. If <exp1> is longer, RTE 301 occurs, except you set

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

  the date is then truncated to 64KBytes. Hint: if you need to store strings larger than 64kb, use .DBT or .DBV memo field (type "VC*")

Both .DBT and .FPT re-uses same blocks for new data, when the new string occupy less or equal blocks as the previous one, otherwise a new sequence of blocks is used.

● **"VC*" are variable length fields**, storing any text or binary data in a file of the same name as the database, with **.DBV** extension. The data may be stored compressed (by LZH algorithm) if the 3rd digit type is Z ("VCZ") or with setting SET(_SET_COMPRESS,.T.). In some cases, the compressed data size may be longer than the original. FlagShip optimizes the amount of data by storing the shorter string (compressed or uncompressed), except you set

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

which always stores compressed (hence crypted) data in VCZ fields. Per default, the full byte range CHR(0..255) is accepted. If you wish to replace Hard-CR = chr(13,10) or chr(10) by space, and also replace soft-CR = chr(141,10) by Hard-CR, set

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

In compressed storage, binary 0 and 0x1A are accepted and handled automaticaly. If you wish to avoid this, set

```
_aGlobSetting[GSET_L_BINARY0_VFIELD] := .F.  // default is .T.
```

If some automatic ASCII or ISO translation is enabled, and you wish to store binary data in VC* field (like images etc.) as well, disable the automatic translation (temporary) with

```
lSetting := Set(_SET_VMEMOBIN, .T.)  // default is .F.
REPLACE ...  // store binary data in VC* memo field
Set(_SET_VMEMOBIN, lSetting)         // restore current setting
```

The total size of V* record is up to 2 Gbytes each. Previous data is re-used when the new data size is less or equal to previous size, otherwise a new record is created in the .DBV file.

● **"VB*" are variable length fields**, storing binary and BLOB data in a file of the same name as the database, with .DBV extension. The data may be stored compressed (by LZH algorithm) if the 3rd digit type is Z ("VBZ") or with setting SET (_SET_COMPRESS, .T.). The size and re-using of records is same as in "VC*" fields.

Multiuser:

RLOCK() is required for replacing a single record, while FLOCK() or an EXCLUSIVE database when <scope> or <condition> is used. If a field of another working area is replaced by specifying its alias, the corresponding record must also be locked with an alias->RLOCK(). If the database or record is not locked by the programmer, FlagShip invokes AUTORLOCK(), when SET AUTOLOCK is enabled (the default).

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

Example 1:

```
USE employee NEW ALIAS empl
USE expenses NEW ALIAS exp INDEX exp_id
SEEK empl->Id
SELECT empl
REPLACE name WITH exp->name
REPLACE Spent WITH Exp->Bus_fare + Exp->Dinner, ;
        Text  WITH Exp->Text
// or:  FIELD->spent := Exp->Bus_fare + Exp->Dinner
//      empl->Text   := Exp->Text
```

Example 2:

Using long memo fields: String variables in FlagShip can contain up to 2 GBytes, whilst the database .FPT Memo field is limited by xBase specification to 64 Kbytes. The alternative is to use “VC” fields. To store larger data to .FPT, use e.g.

```
SET MEMOFILE TO FPT    // create .FPT memo instead of default .DBT
DBCREATE("test", {{"IdNum","N",5,0}, {"Memo1","M",10,0}, ;
                  {"Memo2","M",10,0},{"Memo3","M",10,0} })
USE test
cLongStr := replicate("x", 163840)                // 160 kb
APPEND BLANK
REPLACE FIELD->Memo1 with LEFT  (cLongStr, 65500)        , ;
        FIELD->Memo2 with SUBSTR(cLongStr, 65501, 65500) , ;
        FIELD->Memo3 with SUBSTR(cLongStr, 131001,65500)
...
cLongStr := FIELD->Memo1 + FIELD->Memo2 + FIELD->Memo3
? LEN(cLongStr)                                   // 163840
```

Example 3:

Typical example for multiuser/multitasking:

```
SET EXCLUSIVE OFF                      && set multiuser on
SELECT 5
USE address                            && see more: USE ...
SET INDEX TO name

SEEK "Brown"                           && or user entry
IF FOUND()                             && change data
   xname = name
   xmid  = midname
   xcity = city
ELSE                                   && new entry
   xname = SPACE(25)
   xmid  = SPACE(30)
   xcity = SPACE(LEN(city))
```

```
ENDIF

@ 5, 0 SAY "Name   " GET xname
@ 5,40 SAY "Middle " GET xmid
@ 6, 0 SAY "City   " GET xcity
READ                                   && lock not required

if lastkey() <> 27                     && Esc key pressed ?
   if EOF()
      APPEND BLANK
      WHILE NETERR(); APPEND BLANK; END
   else
      WHILE !RLOCK () ; END
   endif
   REPLACE name WITH xname, midname WITH xmid, ;
           city WITH xcity
   UNLOCK
endif
```

Classification:

database

Compatibility:

The automatic locking is not available in Clipper. FlagShip's autolock is similar to FoxPro and VO. Clipper, FS4, VFS5 and VFS6 silently truncates rest of character field, VFS7 allows tuning, see above for compatibility. The automatic use of .FPT is available in FlagShip only, other drivers (Foxbase, FoxPro, Clipper with DBFCDX) uses it as default. Clipper's .DBT records are limited to 64KB ea.

Translation:

```
REPLACE exp1 WITH exp2  => _FIELD->exp1 := exp2
REPLACE exp1 WITH exp2 [FOR, WHILE...] =>
    DBEVAL({|| _FIELD->exp1 := exp2 [_FIELD->exp...]}, ;
           [{for}], [{while}], [next], [rec], [.rest.] )
```

Related:

APPEND, APPEND BLANK, JOIN, UPDATE, SET EXCLUSIVE, FIELD, MEMVAR, FLOCK(), RLOCK(), UNLOCK, COMMIT, SET MEMOFILE, DbCreate(), oRdd:REplace(), oRdd:FIeldPut()

REPORT EDIT

Syntax:

`REPORT EDIT <file>|(<expC>)`

Purpose:

Interactively creates or modifies reports for use with the REPORT FORM command.

Arguments:

<**file**> is the file which holds the definition of the report. If the file does not exist, a new report is created, otherwise the stored one is modified. The default extension is .frm.

Description:

If the <file> does not exist, a new .frm file is created, otherwise the available one is modified. When executing the REPORT EDIT command, a full design screen for a label appears:

```
myreport.frm F2:pg F3:column F4:group F5:fields F7:displ F10:save ESC:quit
┌F2-----------------------------------------------┐┌F3: column 1/25--------┐
│  Page header  line 1  :     Extract from the  ││  Expr: ADDRESS->IDENTN│
│               line 2  :        for active cus ││  Head: Address        │
│               line 3  :                       ││    #2: Number         │
│               line 4  :                       ││    #3:                │
│  Page   wide/high     : 365 chars    65 lines ││    #4:                │
│  Margin left/right    :   0 chars     0 chars ││  Wide: 8              │
│  Double spacing/plain :   F           F       ││  Deci: 0 Delete/Insert│
│  Eject  begin/end     :   F           T       ││  Total F Add/Replace R│
└-----------------------------------------------┘└-----------------------┘
╔F4═══════════════════════════════════════════════╗┌F5:ADDRESS.dbf---------┐
║  Group key (express.) : article                 ║│ IDENTNUN   N   6   0  │
║        header text #1 : article: &article       ║│ COMPANY    C  25   0  │
║                    #2 :                         ║│ CUST_NO    C  10   0  │
║                    #3 :                         ║│ CUST_TYPE  C   1   0  │
║                    #4 :                         ║│ ADDRESS    C  25   0  │
║        Summary/Eject  : F  /  T                 ║│ STREET     C  25   0  │
║  Subgroup key (expr.) :                         ║│ CITY       C  25   0  │
║        header text #1 :                         ║│ ZIPCODE    C   6   0  │
║                    #2 :                         ║│ TURNOV     N  12   2  │
╚═════════════════════════════════════════════════╝└-----------F6:next dbf┘
┌F7-----------------------------------------------------------------------┐
│     ....,....1....,....2....,....3....,....4....,....5....,....6....,..│
│PgHd:      Extract from the customer's database                          │
│PgHd:         for active customers only                                 │
│GrHd:article: &article                                                   │
│                                                                         │
│Colm:[--1--] [-----------2----------] [---3----] [4-] [-----------5----│
│TxtD:Address Company                  Custommer  type Address            │
│TxtD:Number  name                     Number                             │
│Data: nnnnnn xxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxx x    xxxxxxxxxxxxxxxxx│
│                                                                         │
└-------------------------------------------------------------------------┘
(F4) enter group and sub-group data.  CursUp/CursDn: move   ENTER: confirm
```

The name of the report file is displayed at the top. If not available, you are prompted to "Retry", "Create", "AutoCreate" or "Quit", which lets you re-enter the correct name, create a new report or do that automatically, using the data out of the currently selected database.

You may choose to edit the desired REPORT part by pressing the corresponding function key. With F5/F6 you may select a database from the current or another working area into the database window, or switch to the next database. From this window you may overtake the field name and database alias over to the EXPR. field of the column window, when pressing RETURN.

With F7, you may view the current form of the REPORT with the whole data summarized inside.

You may also choose to prepare the initial report automatically during creation. This will overtake the first 25 fields of the currently selected database and automatically assign adequate column headers. The time to create a report may be reduced by using this option

Example:

Creates and prints a report

```
USE test2
USE sales INDEX article NEW
IF !FILE(salerep.frm")
   REPORT EDIT salerep                  // create report
ENDIF
SEEK 1000
IF FOUND()
   REPORT FORM salerep TO PRINT ;
          FOR   datesold >= DATE() -30 ;
          WHILE article <=  2000 NOCONS
ENDIF
```

Classification:

programming

Compatibility:

The command is available in FlagShip only. To create/modify reports in dBASE III+, use CREATE REPORT; in Clipper the program RL.EXE can be used.

Source:

available in <FlagShip_dir>/system/repoedit.prg

Translation:

__REPOEDIT ("file")

Related:

REPORT FORM, LABEL EDIT

REPORT FORM

Syntax:

```
REPORT FORM <file1>|(<expC1>) [<scope>]
        [FOR <condition>] [WHILE <condition>]
        [TO PRINTER] [NOCONSOLE]
        [TO FILE <file2>|(<expC2>) [ADDITIVE]]
        [NOEJECT] [SUMMARY] [PLAIN]
        [HEADING <expC3>] [MESSAGES <expA4>]
```

Purpose:

Displays a formatted report defined in a .frm file.

Arguments:

<**file1**> is the file which holds the definition of the report. The default extension is .frm.

Options:

<**scope**> is the part of the current database file to report. The default scope is ALL. Either keywords or an expression can be specified.

<**condition**> specifies additional FOR or/and WHILE filtering; see the general command description.

TO PRINTER: echoes output to a printer file. To disable the screen output, use SET CONSOLE OFF.

TO FILE <**file2**>**:** echoes output (ADDITIVE) to the specified file; see also the general command description. Note that formfeed characters are not echoed to the file. To include form feed characters to the file, execute SET PRINTER TO <file**2**> and use the clause TO PRINTER instead of TO FILE.

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

NOEJECT: Suppresses initial page eject when the TO PRINTER clause is used.

SUMMARY: In this case, REPORT FORM displays only total lines of groups, sub-groups, and the grand total line. Detail lines are suppressed.

PLAIN: Suppresses the display of page headers. Moreover, the report title and column headings are displayed only at the beginning of the report. If both PLAIN and HEADING are specified, PLAIN takes precedence.

HEADING <**expC3**>**:** Heading is displayed at the top of each page. Note that <expC3> is evaluated only once at the beginning of the report before the record pointer is moved. Up to three lines can be given, the semicolon ";" acts as the line separator.

MESSAGES <**expA4**>**:** User-defined messages printed during the REPORT FORM execution. The messages are stored in an array[5], with the defaults:

```
    array[1] := "Page Nr."
    array[2] := "* Subsubtotal *"
    array[3] := "** Subtotal **"
    array[4] := "*** Total ***"
    array[5] := "defined page has too few lines"
```

All messages have to be given to be accepted.

Description:

The REPORT FORM executes a report the definition of which is stored in a .frm file. If a path is not specified, the file is searched in the current directory and then in the SET PATH directories. If not found, a run-time error occurs.

The current .frm is created using REPORT EDIT or RL.EXE of Clipper or CREATE REPORT of dBASEIII+.

REPORT FORM sequentially accesses records in the current working area displaying a tabular and optionally grouped report with page and column headings.

Multiuser:

In a multiuser environment, no locking action is required. To ensure that the printed data is not changed while REPORTing it, FLOCK() can be used.

Example:

User break using the ESC key is possible.

```
LOCAL text := {"Page", "*", "", "TOTAL", ;
               "article.frm definition error"}
USE article INDEX name
REPORT FORM articles TO PRINT ;
   WHILE INKEY() # 27 ;
   FOR YEAR(Publ_date) >= YEAR(DATE()) ;
   HEADING "Interesting Articles ;;" + ;
           "I Read This Year ("+ STR(YEAR(DATE())) +".)" ;
   MESSAGES text NOCONSOLE
```

Classification:

programming

Compatibility:

Report files .frm from Clipper or dBASEIII are supported. The clause NOCONSOLE is new in FS4, clauses MESSAGES and ADDITIVE are available in FlagShip only.

Translation:

```
__REPORTFORM ("file1", .print., "file2", .noconsole., ;
             {for}, {while}, next, rec, .rest., .plain., ;
             "expC3", .noeject., .summary., expA4)
```

Related:

REPORT EDIT, DISPLAY, LABEL FORM

REQUEST

Syntax:

`REQUEST <moduleList>`

Purpose:

Declare a module request list for the linker.

Arguments:

<**moduleList**> is a list of external modules to be linked into the executable.

Description:

In contrast to the similar EXTERNAL statement, which specifies a link request to an external procedure name, the REQUEST declarator specifies a request for a module name, such as a .prg or a module declared by the ANNOUNCE statement.

As with EXTERNAL, the request command is used if some modules are not called directly by their names (like DO...NAME or NAME()), but from within macros or INDEX key only. Also, if a .prg file contains only INIT / EXIT PROCEDUREs, a REQUEST command elsewhere may be required to avoid a run-time error "unresolved external".

Example:

```
*** file test.prg *** compiled by: FlagShip test*.prg -na
REQUEST test5
// or: EXTERNAL test2
var := "test2"
DO &var
QUIT
*** file test1.prg ***
ANNOUNCE test5                         // new module name
PROCEDURE test2
? "being now in test2"
RETURN
EXIT PROCEDURE endproc                 // called by FlagShip only
? "bye, bye"
RETURN
```

Classification:

compiler/linker

Compatibility:

Available in FS4 and C5 only.

Related:

ANNOUNCE, EXTERNAL

RESTORE FROM

Syntax:

```
RESTORE FROM <file>|(<expC>) [ADDITIVE]
```

Purpose:

Retrieves PRIVATE and PUBLIC memory variables from a memory (.mem) file.

Arguments:

<**file**> is a memory file to be read. If an extension is not specified, .mem is assumed.

Options:

ADDITIVE: When specified, adds memory variables loaded from the memory file to the existing pool of memory variables. Unless hidden via PRIVATE, memory variables with the same name are overwritten.

Description:

The RESTORE command recreates the names and values of PUBLIC and PRIVATE variables previously saved to the <file> using the SAVE..TO command. Because the class and scope of the variables is not saved, the class of the restored variables depends on the ADDITIVE clause:

- If ADDITIVE is not given, all PUBLIC and PRIVATE variables are released (equivalent to the CLEAR MEMORY command) before the restored variables are created in the PRIVATE class.
- When the clause ADDITIVE is specified and the PUBLIC or PRIVATE variable of the same name is visible, the former value will be overwritten, but the class remains unchanged. Other, or the currently invisible variables will be created as PRIVATEs.

FlagShip also stores and restores PRIVATE and PUBLIC arrays and screen variables, except when FS_SET ("memcompat", .T.) is specified.

LOCAL, STATIC and TYPED variables are unaffected by SAVE and RESTORE. Since the visibility of these variables has precedence, the restored variables of the same name are invisible within the UDF or the STATIC scope, unless they are preceded by the `MEMVAR->` or `M->` alias.

Example:

Usage of variables, created in previous session:

```
PUBLIC test, colors
IF FILE("restvar.mem")
   RESTORE FROM restvar ADDITIVE
ELSE
   test := 25
   colors := {"W+/B", "R/N"}
ENDIF
IF FILE("screens.mem")
   RESTORE FROM screens ADDITIVE
   RESTORE SCREEN FROM scr1
ENDIF
SETCOLOR (colors[1])

SAVE SCREEN TO scr1
SAVE ALL LIKE   scr* TO screens
SAVE ALL EXCEPT scr* TO restvar
QUIT
```

Classification:

programming

Compatibility:

Clipper's screen contents, stored in variables of type "C" cannot be directly used in FlagShip, which stores it in "S" variable types. Use the transfer functions `ScrDos2Unix()` for such. Note that a screen from one terminal can be correctly re-displayed on the same terminal type only.

The content of the screen variables in Terminal i/o and GUI is not compatible to each other. Screen variables in GUI mode cannot be converted via `ScrDos2Unix()` or `ScrUnix2Dos()`.

Unlike Clipper, FlagShip also STOREs and RESTOREs the contents of arrays and screen variables. Normally, the saved arrays and screens can also be read by the DOS dialects, since they simply ignore them. To avoid saving and restoring arrays and screens variables, set the compatibility switch `FS_SET("memcompat", .T.)`.

To transfer .mem files from/to DOS, use `FS_SET("memcompat", .T.) and` binary transfer protocol, see section SYS.

Translation:

__MRESTORE ("file", .add.)

Related:

SAVE, PRIVATE, PUBLIC, LOCAL, STATIC, GLOBAL, CLEAR MEMORY, SAVE SCREEN, FS_SET()

RESTORE SCREEN

Syntax:

```
RESTORE SCREEN [FROM <memvar>]
```

Purpose:

Displays a screen that has been saved to a memory variable.

Options:

<**memvar**> is a screen variable to which the screen display was saved using the SAVE SCREEN command. This variable is of the type "S" and cannot be used for string or arithmetic operations. It may however, be translated to a character type and back, using SCREEN2CHR(), CHR2SCREEN() respectively. For manipulating the screen variable, see (EXT) _retscw().

If <memvar> is not specified, the screen display is saved and restored to/from an internal variable which is overwritten each time a new screen is saved.

Description:

RESTORE SCREEN is used for redrawing the whole screen that was saved with SAVE SCREEN. Normally, it is used to change a screen temporarily and return to it later. Otherwise, the screen would have to be repainted. To STORE and RESTORE a part of the screen, the functions SAVESCREEN() and RESTSCREEN() should be used instead. Saving/restoring only the required part of screen will speed up the application visibly.

***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 RESTORE SCREEN by setting

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

which avoids clearing the restored area beforehand and is therefore significantly faster.

Example:

```
USE authors
SAVE SCREEN TO myscreen
REPORT FORM authors
RESTORE SCREEN FROM myscreen
```

Classification:

programming

Compatibility:

To save the contents of a screen, FlagShip uses a variable of type "S" in contrast to the Clipper's type "C", which is not binary compatible. Use the transfer functions SCRDOS2Unix() or SCRUnix2DOS() when you need to save/restore the DOS stored screen contents. See _retscw() in chapter EXT if you need to manipulate the contents of the screen variable. See also compatibility notes in RESTORE FROM command.

Translation:

__XRESTSCREEN( )

Related:

SAVE SCREEN, SAVESCREEN(), RESTSCREEN(), SCREEN2CHR() CHR2SCREEN(), SCRUnix2DOS(), SCRDOS2Unix()

RETURN

Syntax:

`RETURN [<exp>]`

Purpose:

Terminates a procedure (UDP), function (UDF) or the entire program returning control to either the calling procedure or the Unix or MS-Windows operating system.

Arguments:

<**exp**> is an expression of any type that evaluates to the return value for a user-defined function. If not specified, the UDF returns NIL.

Description:

In a procedure (UDP) or function (UDF), FlagShip releases all PRIVATE, autoPRIVATE and LOCAL variables created there and returns to the calling procedure. When RETURN is executed at the highest level, control is passed to Unix or Windows resp.

There can be more than one RETURN in a UDP or UDF. RETURN (NIL) is also assumed when reaching another PROCEDURE or FUNCTION command, or the end-of-file.

The RETURN statement passes control only one level up, to the calling procedure. However, using the BREAK command it is possible to jump more than one level at a time, into the next BEGIN SEQUENCE...END structure. This is similar to RETURN TO MASTER of dBASE, but more flexible.

Example:

```
FUNCTION myudf (par1, name, par3)
? name
RETURN par1 + par2

PROCEDURE myudp
PARAMETERS par1, name, par3, retpar
? name
retpar = par1 + par2                    // passes back by param
RETURN
```

Classification:

programming

Compatibility:

In an UDF, FlagShip accepts a RETURN without an <exp>, returning NIL. Clipper requires a given return value.

Related:

CANCEL, QUIT, BEGIN SEQUENCE, FUNCTION, PROCEDURE

RUN

Syntax:

```
RUN [WAIT|NOWAIT] [MESSAGE <expC1>]
        <Unix command|Windows command>|(<expC2>)
```

or:

```
! [WAIT|NOWAIT] [MESSAGE <expC1>]
        <Unix command|Windows command>|(<expC2>)
```

Purpose:

Executes Unix or Windows command, program or script within the current application. This allows the use of the power of Unix and Windows commands and shell. In MS-Windows, RUN executes internal CMD commands and .exe, .com or .bat files.

Arguments:

<**Unix|Windows command**> may be any executable program or script within the path. Any character expression has to be enclosed in parentheses. Macro expressions can also be used and will be expanded before submitting the command to the shell. In Windows, you may execute internal CMD commands (like VOL, DIR, COPY etc.) via `RUN "CMD /C command.."` but FlagShip will precede the command string by "CMD /C " automatically (for internal shell commands only), if not disabled, see section "Tuning" below. You should use parenthesed expression (expC2) instead of command constant, when the command/expression contain backslashes or spaces.

Options:

WAIT or **NOWAIT**: optional modifier. With WAIT (default), the application will wait until the command will finish. NOWAIT will trigger the command to background and continue execution of the application. NOWAIT is similar to Unix command "shell_call &". Do not use WAIT/NOWAIT clause together with the "&" postfix.

MESSAGE <**expC1**> is an optional, user defined message to be printed on the screen, when the executed Unix command is finished. Note, no FlagShip output mapping is active when the MESSAGE is printed; it works as does the "echo <expC1>" from the Unix shell would. Before <expC1> is printed, a NEW LINE is executed (similar to the WAIT command).

Note that both options, if any given, needs to precede the command.

Return code:

The return code may be checked via DosError() function. Note: this return code is system dependant and correspond to the return value of system function system() or of errno if system() returns -1. On some oper. systems, you will get the true exit code by calculating nRet := int(DosError() / 256)

Description:

At RUN command, FlagShip invokes a new shell and passes it the Unix or Windows command to be executed. The required command must be available in the current path or else given with an absolute path.

When the <Unix/Windows command> ends (or when the background process is started by "&" postfix or by NOWAIT clause), the control returns back to the application, executing the next FlagShip statement.

In Linux and Unix, FlagShip uses the system() function (see "man 3 system"), which calls "sh -c <Unix_command>". With NOWAIT clause, FlagShip adds " &" to the <Unix_command> if not available there.

In MS-Windows, FlagShip checks <Windows_command> for CMD's internal command (like DIR, CD, CALL, COPY etc.) and if so, invokes the CMD shell command; otherwise it executes the command via _spawnvp(), by using the PATH environment to locate it when <Windows_command> does not include drive and/or path. You need to specify fully qualified name including drive and path, when the path of the given executable is not included in the current MS-Windows PATH environment variable. When the path or file name contain spaces, you need to enclose the corresponding parameters in quotas, see example below.

To allow the output from the program called to be inspected, print a prompt (using e.g. the MESSAGE clause or the equivalent "; echo..." statement) and stop further execution with INKEY(0) after the RUN command; see example.

Shell access: You may run a shell by specifying the argument "sh" (or "csh", "ksh" respectively) to the RUN command. To exit the shell, type "exit". In MS-Windows, invoke CMD or COMMAND for that reason.

Background processing: the executable or script called may run in background, if the RUN command specification ends with an ampersand (&) character or by using the NOWAIT clause. The current application will not wait for the called executable to finish, but will carry on with its own execution immediately. The program called becomes a child of the calling executable and will terminate latest when the current application terminates. Applicable in Unix/Linux only. Note that any input to, or output from the background program may cause the called application to hang.

User break: when the program called is a FlagShip application, both programs will receive the break and debug signals (^K and ^O).

Environment variables: cannot be set from a RUN command, since they are local to the executing subprocess and do not affect the calling application. Use FS_SET ("setenv") instead. The current PATH environment variable is used to search for the executable, when the command does not include path. Available environment variables can be retrieved by GETENV() function.

Screen output: in Terminal i/o mode, the output goes to the screen and is handled by the curses library, as described in section SYS. This library optimizes the current output stream. If the by RUN called programs produce any screen output, it will be thereafter "invisible" for the curses buffer from the calling FlagShip application. Also, the new cursor position is not conveyed to the calling application; the screen output becomes undefined (for the curses library). To synchronize the physical screen with the current application, execute any of the REFRESH, [@..]CLEAR.., RESTSCREEN() or SCROLL() functions after RUN. To avoid this de-synchronizing, best to redirect the output to file and fetch the result, see example 2 and 3 below.

In Basic i/o mode, the output from the by RUN called programs goes to stdout or stderr.

In GUI mode, the output from the called program goes to stdout or stderr, which is usually assigned to the console (or console window) and hence does not affect the current screen - or is not displayed at all. Best to redirect the output to file and fetch the result, see example 2 and 3 below.

Compatibility note: since the Unix and MS-Windows commands usually differs from each other, you may use

```
#ifdef FS_WIN32
   RUN Windows-Command...
#else
   RUN Unix-Command...
#endif
```

Tuning:

If an error occurs, it is displayed as run-time-error. You may disable this by assigning

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

You may display the full RUN command & time on console by assigning

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

To disable the detection of internal CMD commands (in FlagShip for MS-Windows) and avoid it automatic prefacing by "CMD /C ", set

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

Example 1:

Execute a simple Unix or DOS/Windows program:

```
SAVE SCREEN
#ifdef FS_WIN32
   RUN ("CMD /C dir *.prg | more")    // MS-Windows
   WAIT
#else
   RUN MESSAGE "press any key..." ls -l *.prg | pg   // Unix
   // or: RUN ("ls -l *.prg | pg ; echo press any key...")
   INKEY (0)
#endif
RESTORE SCREEN
```

Example 2:

To pass the output (stdout and stderr from shell or CMD) into file (and omit restoring the screen), you may use:

```
#ifdef FS_WIN32
   RUN ("dir *.prg >temp.txt 2>&1")
#else
   RUN ("ls -la *.prg >temp.txt 2>&1")
#endif
? MemoRead("temp.txt")      // or: TYPE temp.txt
```

Example 3:

Execute another FlagShip program "prg2[.exe]" in the background, which creates a file prg2.txt containing return values on exit:

```
 IF FILE("prg2.txt")
    ERASE FILE prg2.txt
 ENDIF
 #ifdef FS_WIN32
    RUN ("prg2.exe par1 par2")           // execute prg2 with param
 #else
    RUN ("./prg2 par1 par2 &")           // in Linux/Unix:
background
 #endif
 WHILE .not. FILE("prg2.txt")
    INKEY(3)
    ? "waiting for prg2 to be finished"
 ENDDO
 values = MEMOREAD ("prg2.txt")          // get results
```

Example 4:

Invoke MS-Word with available document in the Windows version of FlagShip. Note that the command and/or parameters must be enclosed in quotas when the path or file name include spaces (which would be otherwise interpreted by CMD/COMMAND as parameter delimiter). Hint: handle RUN in the same way as when you invoke an executable at command line level.

```
#ifdef FS_WIN32
  myPath := getenv("HOMEPATH")+"\Documents\"  // may contain spaces
#else
  myPath := getenv("HOME") + "/Documents/"
endif
myText := TextProcessor(myPath + "letter.doc")
? myText
wait

/***************************************************
 * invokes text processor for given file name,
 * returns edited data
 */
FUNCTION TextProcessor(cFile)
local cEditor := ""
#ifdef FS_WIN32
   // OpenOffice or LibreOffice
   cEditor := getenv("ProgramFiles(x86)")+"\program\swriter.exe"
```

```
    if !file(cEditor)
       cEditor := getenv("ProgramFiles")+"\program\swriter.exe"
    endif
    if !file(cEditor)
       cEditor := getenv("ProgramFiles(x86)") + ;
                  "\LibreOffice 3.6\swriter.exe"
    endif
    if !file(cEditor)
       // MS-Office
       cEditor := getenv("ProgramFiles(x86)") + ;
                  "\Microsoft Office\Office\WinWord.EXE"
    endif
    if !file(cEditor)
       cEditor := FindExeFile("Notepad.exe")
    endif
 #else
    // OpenOffice or SOffice
    cEditor := FindExeFile("ooffice")
    if empty(cEditor)
      cEditor := FindExeFile("soffice")
    endif
    if empty(cEditor)
      cEditor := FindExeFile("gedit")
    endif
 #endif
 if empty(cEditor)
    alert("cannot locate text editor, contact programmer")
    return ""
 endif
 RUN ('"' + cEditor + '" "' +cFile + '"') // accepts spaces in files
 // wait cEditor+" success/err:" + ltrim(doserror())+ ", any key..."
 return MemoRead(cFile)
```

Example 5:

Start MS-Word (Winword) in Windows as sub-process, continue processing of the application. Note the notification of path and/or file name including spaces: the executable (with path) and/or the file name needs to be passed to Windows enclosed in double quotas. When the command uses variables, enclose it in parentheses.

```
 ? "Invoking MS-Word as separate process..."
 RUN NOWAIT ;
     '"C:\Program Files\Microsoft Office\Office\Winword.exe" /w'
 if doserror() != 0
    ? "could not invoke Word, CMD return code =", ltrim(doserror())
    ? " =", doserror2str()
 else
    ? "Word is active, exit it separately (before exit application)"
 endif

 // or alternatively:

 cCommand := '"C:\Program Files\Microsoft Office\' + ;
             'Office\Winword.exe"'
 cDocFile := '"D:\Documens and Settings\Default User\' + ;
             'My Documents\letter.doc"'
 RUN NOWAIT (cCommand + " " + cDocFile)
```

Example 6:

Execute a time-consuming Unix command in background, omit error output, obtain its data later:

```
LOCAL ii := 0, text
FERASE("find.ready")

// trigger Unix find command in background
RUN "(find / -name '[k-m]*.prg' -print > find.data " + ;
    " 2>/dev/null ; touch find.ready)&"

// now, continue the program execution,
// e.g. allow an user input
// or display the system is working:
@ 5, 0 SAY "Searching"
DO WHILE .NOT. FILE("find.ready") .and. INKEY() != 27
   @ 5, 10 SAY substr("\|/-", (ii++ % 4) +1, 1)
ENDDO
// display data from the background job:

IF LASTKEY() != 27
*  @ 5, 10 SAY "READY"
*  TYPE find.data
// --- or use the more comfortable: ---
   @ 5, 0 SAY "Scroll by PgDn,PgUp. Continue with ESC"
   text := MEMOREAD("find.data")
   IF LEN(text) < 10
      text := "***** No data for [k-m]*.prg found *****"
   ENDIF
   MEMOEDIT (text, 7,0, MAXROW() -1, MAXCOL(), .F.)
ENDIF
FERASE("find.ready")
```

Example 7:

For true inter-process communication, see also the example in the section EXT.

Classification: system call

Compatibility:

As opposed to the equivalent DOS execution, there are practically no limits to using RUN on Unix and in MS-Windows. If the available RAM space is insufficient, the additional swap disk area will be used automatically.

Keep in mind the differences in system command names on DOS and Unix (ls instead of DIR etc.) and the different DOS vs. Unix screen handling. For portability, #ifdef FlagShip.. ..#else...#endif or the PUBLIC FLAGSHIP variable can be used.

The MESSAGE clause is new in FS4, WAIT/NOWAIT in FS6 and both are not available in Clipper.

Translation: `__RUN ("expC2" [, "expC1"] )`

Related: REFRESH, REFRESH(), CLEAR SCREEN, SAVE/RESTORE SCREEN

SAVE TO

Syntax:

```
SAVE TO <file>|(<expC>)
           [ALL [ LIKE | EXCEPT <skeleton>]]
```

Purpose:

Saves PRIVATE and PUBLIC memory variables to a memory (.mem) file.

Arguments:

<**file**> is the name of the file where the specified memory variables are saved. If no extension is specified, the file is created with a .mem extension.

Options:

ALL saves all visible dynamic (PRIVATE and PUBLIC) variables. This is the default setting, if no other clause is specified.

ALL LIKE <**skeleton**> defines a set of visible PRIVATE and/or PUBLIC variables to be saved.

ALL EXCEPT <**skeleton**> defines a set of visible PRIVATE and/or PUBLIC variables not matching the <skeleton> to be saved.

<**skeleton**> is a wildcard mask (* and ? are supported) which specifies a group of variables for the ALL LIKE or EXCEPT clause. The wildcard character "*" matches any group of adjacent characters. The wildcard character "?" matches any single character and can be specified anywhere within the <skeleton>.

Description:

The specified memory variables are copied to the memory file without regard to their scope (PUBLIC or PRIVATE).

FlagShip also stores and restores PRIVATE and PUBLIC arrays and screen variables, unless the `FS_SET("memcompat", .T.)` was specified for compatibility to Clipper.

LOCAL, STATIC and TYPED variables are unaffected by SAVE and RESTORE.

Example:

```
PRIVATE var1 := 1,   var2 := "two"
PUBLIC  var3 := .T., var4 := date()
LOCAL   var5 := "test"
abc := "autoprivate"

DO myPROC WITH var5

PROCEDURE myPROC
PARAMETERS var_par
LOCAL var1, var4                // does not affect SAVE TO
SAVE TO testsav1 ALL LIKE var*  // var1..5, var_par
SAVE TO testsav2                // abc, var1..5, var_par
RETURN
```

Classification:

programming

Compatibility:

FlagShip stores the screen contents in "S" variable types, which is not compatible to Clipper's variables of type "C". Use the transfer functions `SCRDOS2Unix()` or `SCRUnix2DOS()` for such purposes in terminal i/o mode (conversion is not available for GUI mode). Note that the screen from one terminal can only be correctly re-displayed on a terminal of the same type having the same value in the `TERM` environment variable.

The content of the screen variables in Terminal i/o and GUI is not compatible to each other. Screen variables in GUI mode cannot be converted via ScrDos2Unix() or ScrUnix2Dos().

Unlike Clipper, FlagShip also `STORE`s and `RESTORE`s the contents of arrays and screen variables. Normally, the saved arrays and screens can also be read by the DOS dialects, since they simply ignore them. To omit saving and restoring array and screen variables, set the compatibility switch `FS_SET ("memcompat", .T.)`.

To transfer .mem files from/to DOS, a binary protocol must be used, see section SYS.

Translation:

```
__MSAVE ( "file", "skeleton", .like. )
```

Related:

RESTORE FROM, PRIVATE, PUBLIC, LOCAL, STATIC, GLOBAL, FS_SET(), CHR2SCREEN(), SCREEN2CHR(), SCRDOS2Unix(), SCRUnix2DOS()

SAVE SCREEN

Syntax:

`SAVE SCREEN [TO <memvar>]`

Purpose:

Saves the current screen contents to a **screen** variable.

Options:

TO <**memvar**> specifies a variable to which the display screen was saved, and from which it will be RESTOREd. This variable is of the type "S" and cannot be used for string or arithmetic operations. The variable can be of any storage class including LOCAL, STATIC, or an array element. To store it to a character or memo FIELD and back, use the translation SCREEN2CHR() or CHR2SCREEN() respectively. On how to manipulate the screen variable, see (EXT) _retscw(). When using “V*” memo field, you may store and restore the screen variable data “as is” (required in GUI mode, where SCREEN2CHR() and CHR2SCREEN() is not applicable).

If this clause is not specified, the screen display is saved to an internal variable which is overwritten each time a new screen is saved.

The content of the screen variables in Terminal i/o and GUI is not compatible to each other. Screen variables in GUI mode cannot be converted via Screen2chr() nor Chr2screen() but will be stored to memory variable by the same way as in Terminal i/o mode. See addit. description about screen variables in the SaveScreen() function.

Description:

SAVE SCREEN is used in conjunction with RESTORE SCREEN to avoid repainting an original screen that has been temporarily replaced. The command is a synonym for the SAVESCREEN(0, 0, MAXROW(), MAXCOL()) function.

To STORE and RESTORE a part of the screen, the functions SAVESCREEN() and RESTSCREEN() should be used instead.

All the current visible characters, colors, attributes and the cursor position is saved and will be faithfully redisplayed. For "odd output" from RUN programs or other sessions etc., see the (CMD) REFRESH command.

In GUI mode, you may compress the resulting image size by setting SET SCRCOMPRESS ON. However, the compressed image may loose some precision at RESTORE, similarly to compressing of jpeg files.

Example:

```
USE authors
SAVE SCREEN TO Scr1
CLEAR
REPORT FORM authors
WAIT
RESTORE SCREEN FROM Scr1
USE
```

Classification:

programming

Compatibility:

For saving a screen contents, FlagShip uses the variable of type "S" as opposed to the Clipper's type "C", which is not binary compatible. Use the transfer functions SCRDOS2Unix() or SCRUnix2DOS() for such purposes (applicable in terminal i/o only). See also compatibility notes in the RESTORE FROM command.

Translation:

__XSAVESCREEN( )

Related:

RESTORE SCREEN, CHR2SCREEN(), RESTSCREEN(), SAVESCREEN(), SCREEN2CHR(), SCRDOS2Unix(), SCRUnix2DOS()

SEEK

Syntax:

```
SEEK <exp> [SOFTSEEK]
```

Purpose:

Seeks through an index file until the first key matching the given expression is found.

Options:

SOFTSEEK overrides the current SET SOFTSEEK state, and executes the SEEK as if SOFTSEEK were ON.

Arguments:

<**exp**> 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).

If SET ANSI is set ON, or SET DBREAD is set to ANSI, the <exp> is translated automatically by Ansi2oem(). See also examples/setansi.prg with additional hints.

Description:

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.

When the searched value is not found, the current state of SET SOFTSEEK affects the returned from FOUND(), EOF() and the position of the record pointer:

- If SOFTSEEK is OFF (the default), FOUND() returns FALSE, EOF() returns TRUE, and the database is positioned at eof = LASTREC() +1.
- If SOFTSEEK is ON or the SOFTSEEK clause is used, and there are keys with a value greater than the searched argument, the database pointer is positioned to the first record with a greater key value, FOUND() returns FALSE and EOF() returns FALSE.
- If SOFTSEEK is ON or the SOFTSEEK clause is used, and there is no key with a value greater than the searched argument, the database is positioned at eof = LASTREC() +1, FOUND() returns FALSE and EOF() returns TRUE.

The SET DELETED and SET FILTER switch/condition is considered. The current state of SET EXACT does not affect the search; the comparison is the same as with SET EXACT OFF.

SEEK is identical to FIND, but has a slightly different syntax: `FIND &<var>` is identical to `SEEK <var>` and `FIND (<var>)` is identical to `SEEK <var>`.

For a more complex SEEK search, the SEEK EVAL command may be used.

Tuning:

You may force COMMIT on SEEK by setting

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

it behaves then same as VFS6 and Clipper.

Example:

To list all employees whose last name is "Clifton"

```
USE employee INDEX name, zip
SEEK "Clifton"
IF FOUND()
   LIST REST Firstname, Lastname, Birthdate;
      WHILE Lastname = "Clifton" .and. ;
            INKEY() != 27
ENDIF

SET SOFTSEEK ON
SET ORDER TO 2                         // index: zip
SEEK 12345                             // 12345 and above
SET SOFTSEEK OFF
IF FOUND()
   ? "zip code 12345:", city
ELSEIF .NOT. EOF()
   ? "next zip code to 12345 =", zip, ":", city
ELSE
   ? "address for zip code 12345 and up not available"
ENDIF
```

Classification:

database

Compatibility:

SOFTSEEK is available in FlagShip (with default RDD) only.

Translation:

DBSEEK (exp)

Related:

FIND, SEEK EVAL, INDEX, LOCATE, REINDEX, SET DELETED, SET EXACT, SET INDEX, SET SOFTSEEK, USE, EOF(), FOUND(), RECNO(), oRdd:Seek()

SEEK EVAL

Syntax:

```
SEEK EVAL <expB>
```

Purpose:

Seeks through an index file until the evaluated code block returns TRUE.

Arguments:

EVAL <**expB**> is a code block, which performs some comparisons with the index key of the controlling index. The scope is REST (the search starts with the current record).

Description:

By using SEEK EVAL, a complex search (like substring etc.) in the index file can be performed. It is similar to the LOCATE command, but significantly faster, since the database record is read during the search process on request only (e.g. when a field name is specified in the code block body). Of course, the database pointer is positioned correctly latest at the end of the search.

The search of the controlling index starts with the current record and continues until the <expB> code block returns TRUE or until end-of-file.

The code block receives the current index key, and the corresponding record number as parameters in that order, and must return logical FALSE to continue the search or TRUE to stop.

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.

When the searched value is not found, FOUND() returns FALSE, EOF() returns TRUE, and the database is positioned at eof = LASTREC() +1.

The current SET SOFTSEEK state does not affect the SEEK EVAL command. The SET DELETED and SET FILTER switch/condition is considered. The current SET EXACT state affects string comparisons within the code block.

Note, the scope is REST (i.e. starting from the current record). For the first complete search, the record pointer has to be positioned to the first logical record using GOTO TOP. To continue the search, use SKIP prior to issuing the SEEK EVAL command.

The sequence GOTO TOP; SEEK EVAL {|key| key=exp} is equivalent to SEEK exp , but the latter is executed faster.

The database record pointer (LASTREC()), if used in the code block, delivers the record number before SEEK EVAL was started. Use the second code block parameter to determine the correct record number.

Example:

To find a name entered in any order

```
LOCAL seekname  := "Smith", maxrec := lastrec()
LOCAL seekblock := {|key, recno|                 ;
                    UPPER(seekname) $ key .AND.  ;
                    !deleted() .AND. recno < maxrec}
USE address NEW
INDEX ON UPPER(name) TO adrname
LIST name FOR UPPER(seekname) $ UPPER(name)         // slow
*   5  John Smith
*   9  Peter Smith
*  12  Peter & Paul Smith Corp.
*  36  Smith and Partner Ltd.

GOTO TOP
SEEK EVAL seekblock                  // 5 John Smith
WHILE ! EOF()
   ? RECNO(), name
   SKIP
   SEEK EVAL seekblock               // 9 Peter Smith (etc.)
ENDDO
```

Classification:

database

Compatibility:

Available in FlagShip (with the default DBFIDX driver) only

Translation:

_SEEKEVAL (expB)

Related:

SEEK, FIND, INDEX, LOCATE, CONTINUE, SKIP, oRdd:SEEKEVAL()

SELECT

Syntax:

`SELECT <workArea>|(<expN>)`

or:

`SELECT <alias>|(<expC>)`

Purpose:

Changes the current working area.

Arguments:

<**workArea**> is a number between zero and 65534. If zero is specified, the lowest available working area without an open database is selected. The argument can be specified as a numeric expression (<expN>) enclosed in parentheses (not to be confused with the equivalent SELECT() function syntax).

<**alias**> is the ALIAS name of the working area containing the opened database file with the same name or alias. The alias can be specified as a character expression (<expC>) enclosed in parentheses. The case of <alias> string is not significant. If the alias is not found (i.e. the database is yet not open), the first unused working area is selected, similar to SELECT 0 or the USE... NEW clause.

Description:

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.

A zero argument selects the first unused working area, similar to the USE...NEW clause.

FlagShip supports the direct usage of the <alias>-> selector in assignments, expressions and function calls, which is equivalent to SELECTing the required working area, see also section LNG.2.9 and LNG.2.3.2:

```
USE address ALIAS addr  NEW ; act := SELECT()
SELECT 15
USE other

* ---- aliased -------           * ---- is equivalent to ---------
addr->name := xyz                SELECT addr
                                 FIELD->name := xyz
                                 SELECT other
? adress->(EOF())                SELECT addr; ? EOF() ; SELE other
("ad"+"dr")->(MyUdf(1))          SELECT addr; MyUdf(1); SELE other
15->(MyUdf(2))                   SELECT 15  ; MyUdf(2); SELE other
```

Each working area has the following attributes:

Attribute/Action	Retrieving Command/Function
Open/close work area	USE, CLOSE DATA
Indices	USE..INDEX, SET INDEX
Relations	SET/CLOSE RELATION
Filtering	SET FILTER, SET DELETED
Searching	SEEK, LOCATE, FIND
Moving	GOTO, SKIP
Alias name	ALIAS()
Database file	DBF(), INDEXDBF()
Working area no.	SELECT()
Index file ext, names	INDEXEXT(), INDEXNAMES()
Index key, contrl.no.	INDEXKEY(), INDEXORD()
Index integrity	INDEXCHECK()
Record number	RECNO()
Record count	LASTREC(), RECCOUNT()
Field count	FCOUNT()
Field name	FIELD()
Field description	AFIELDS()
Beginning-of-file flag	BOF()
End-of-file flag	EOF()
Filter condition	DBFILTER(), DELETED()
Locate/Seek result	FOUND()
Relation	DBRELATION(), DBRSELECT()
Header size	HEADER()
Network cmd result	NETERR()
Locking	RLOCK(), FLOCK(), UNLOCK, AUTOxLOCK(), SET AUTOLOCK

Multiuser:

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

Tuning:

See tuning details for _aGlobSetting[GSET_L_DBCOMMIT_SELECT] in SET COMMIT

Example:

```
old_area = SELECT()
SELECT 0
USE magazine ALIAS mag                 && or: USE...NEW
// other statements
mag_select = SELECT()
SELECT (old_area)
// other statements
SELECT (mag_select)                    && or: SELECT mag
```

Classification:

database

Compatibility:

FlagShip supports 65534 working areas simultaneously, Clipper and VO up to 250, dBASE up to 10 or 40.

Translation:

```
DBSELECTAREA (expN | expC)
```

Related:

USE, SET INDEX, SET COMMIT, ALIAS(), SELECT()

SET ALTERNATE

Syntax 1:

`SET ALTERNATE TO [<file>|(<expC>) [ADDITIVE] ]`

Syntax 2:

`SET ALTERNATE on|OFF|(<expL>)`

Purpose:

Echoes console output (e.g. of the ?/?? commands) to an ASCII text file.

Arguments:

TO <**file**> is the name of an ASCII text file to which the output will be redirected and can include a path and an extension. If the file extension is not specified, .txt is assumed. When the TO... clause is not given, the currently open alternate file (if any) will be closed.

Option:

ADDITIVE causes the specified alternate file to be appended to instead of overwritten. If not specified, the specified <file> is truncated.

Arguments:

ON/OFF activates or deactivates the output to the current open alternate file. The toggle will not be switched to ON if the alternate file is not opened. Alternatively, the parenthesized <expL> may be used, whereby logical TRUE is the same as ON.

Description:

FlagShip allows to redirect console command output (such as ?, LIST, REPORT FORM, LABEL FORM) to four different devices/files at a time: the SCREEN device, and the ALTERNATE, PRINTER and EXTRA files or devices.

In commands, which support the TO FILE <file> clause (like LIST, REPORT FORM etc.), these clauses perform a similar function as SET ALTERNATE. In other commands (like ?, ??, QOUT() etc.), an additional redirection to a text file (or device) using the SET EXTRA command is possible.

Full-screen commands such as @...SAY cannot be echoed to by the SET ALTERNATE or EXTRA command; use SET DEVICE instead.

By setting the output OFF, the alternate file remains open. Closing the alternate file using SET ALTERNATE TO or CLOSE ALTERNATE will reset the toggle to OFF. Only one alternate file may be opened at the same time.

Tuning:

You may set the new-line character by 8th element in FS_SET("prset") e.g.

```
   #ifdef FS_WIN32       /* here: should apply for Windows only */
     FS_SET("prset", {NIL,NIL,NIL,NIL,NIL,NIL,NIL,chr(13,10) } )
   #endif
```

before printing to ALTERNATE file via ? or QOUT(). The default setting is line-feed = chr(10).

Example:

```
SET ALTERNATE TO protocol.doc           && or: TO /dev/lp1
SET ALTERNATE ON
USE publish
DO WHILE .NOT. EOF() .AND. INKEY() # 27
   ? Name, Address, Zip, Town
   SKIP
ENDDO
SET ALTERNATE TO
```

Classification:

programming

Compatibility:

The ADDITIVE clause is new in FS4.

Translation:

SET (_SET_ALTERNATE, .on.)
SET (_SET_ALTFILE, "file", .additive.)

Related:

?, ??, DISPLAY, LIST, LABEL FORM, REPORT FORM, TEXT, TYPE, QOUT(), QQOUT(), SET EXTRA, SET PRINTER, SET()

SET ANSI

Syntax:

```
SET ANSI on|OFF|(<expL>)
```

Purpose:

Change the behavior how to read from and store data into database.

Arguments:

ON/OFF activates or deactivates the automatic translation of ANSI <-> PC8 character set. The logical value .T. correspond to ON, .F. is the same as OFF. Default value is OFF.

Description:

With SET ANSI ON or SetAnsi(.T.), a database access of character or memo translates the PC8/ASCII/OEM charset 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.

Note: both the FS4 and Clipper always use PC8/ASCII charset in the database, i.e. chr(132) for a-umlaut.

Example:

See examples/setansi.prg for a complete example with description

Classification:

programming, database

Compatibility:

New in FS5

Related:

SET SOURCE, SetAnsi(), SET DBREAD, SET DBWRITE, Ansi2oem(), Oem2Ansi(), SET KEYTRANSL|CHARSET,

SET AUTOCOMMIT

Syntax:

```
SET AUTOCOMMIT on|OFF|(<expL>)
```

Purpose:

Sets or disables an automatic COMMIT on the UNLOCK command or corresponding DbUnlock() function.

Arguments:

When specified **ON**, an automatic COMMIT will be enabled, which then flushes the database (and indices) changes physically to hard disk at every UNLOCK command or at DbUnlock() function. Apply for databases open in SHARED mode, ignored for EXCLUSIVE open databases.

OFF (the default) disables the automatic commit.

<**expL**> is optional parenthesized logical value or expression, where .T. is equivalent to ON, and .F. to OFF

Description:

In multi-user mode, the changes in the database and indices are usually hold in operating system buffer and written physically to the hard disk by the COMMIT command or DbCommit() function.

FlagShip can perform this action automatically when SET AUTOCOMMIT is ON. But since this may slow-down the performance (especially when using UNLOCK in a large loop), the default setting is OFF. See also SET COMMIT for additional tuning.

Tuning:

SET AUTOCOMMIT simply set _aGlobSetting[GSET_N_DBCOMMIT_UNLOCK] to 0 (OFF) or to 1 (ON). You also may assign this manually or via the SET COMMIT command.

Classification:

database

Compatibility:

Available in FlagShip only. In FlagShip VFS7, the SET AUTOCOMMIT had also set _aGlobSetting[GSET_L_DBCOMMIT_SELECT]:= .T. (or .F.) which is disabled now in VFS8 (and newer) for performance.

Translation:

```
SET ( _SET_AUTOCOMMIT, <expL> )
```

Related:

SET COMMIT, SET(), COMMIT, UNLOCK, DbCommit()

SET AUTOLOCK

Syntax 1:

```
SET AUTOLOCK [TO] <expN>
```

Syntax 2:

```
SET AUTOLOCK ON|off|(<expL>)
```

Purpose:

Sets or disables the placement of an automatic record or file lock during a database write access in shared mode.

Arguments:

<**expN**>sets the period (in seconds), for which to attempt to successfully execute the automatic Rlock or Flock. Attempts are done in one second intervals. The default <expN> period is 10 (seconds). Possible values for <expN> are:

0	auto locking is enabled, the AUTOxLOCK() function tries to lock the database forever, until it succeeds.
>= 1	auto locking is enabled, the AUTOxLOCK() function tries to lock the database successfully for <expN> seconds. If not successful within this period, the user may choose the action to follow in a communication window, i.e. to try again, break, ignore (mostly resulting in a subsequent run-time error), or to exit.
- 1	is equivalent to 0, for FoxPro compatibility.
- 2	is equivalent to 0, for FoxPro compatibility.
- 3	and all values < -3: the auto locking is disabled.

ON/OFF is a shortcut of the syntax 1, while ON is equivalent to ...TO 10 (or the last positive value previously set), while OFF is equivalent to ...TO -3. The default is ON. When using the alternative parenthesized <expL>, TRUE is the same as ON.

Description:

When a database is open in SHARED (multiuser) mode, any writing access requires a record or file lock. Usually, the programmer controls these locks himself, by using the RLOCK() and FLOCK() functions and the UNLOCK command.

For your convenience, FlagShip RDD drivers can manage these locks themselves, if the lock was **not** already issued by the programmer and SET AUTOLOCK switch is active (the default).

In this case, the database driver calls the function AUTORLOCK() or AUTOFLOCK() respectively before the database write access, and AUTOUNLOCK() thereafter to release this lock and COMMIT the database. You may modify these functions, available in source code in the <FlagShip_dir>/system/autolock.prg file, e.g. to display waiting messages, manage the default BREAK, protocol the "unexpected" locks etc. If a modification is required, copy this file to your local directory; then compile and link it according to the instructions given in the source - or simply set a global switch in your application, see Tuning below.

Note, that SET AUTOLOCK is a global switch, valid for all databases, as opposed to the local oRdd:CONCURRENCYCONTROL instance of the DataServer class.

Performance hints, transactions:

Of course, this autolock mechanism is not so effective, as when the programmer controls the flow by RLOCK()...UNLOCK, FLOCK() ... UNLOCK or by explicitly invoking the AUTOxLOCK() ... AUTOUNLOCK() functions. This is because the programmer usually invokes only one lock attempt for multiple field replacements on the same record. On the other hand, the AutoLock functions have to perform (the same) lock and unlock (including Commit) for every field replacement. Therefore, your application may be faster when your program controls the locks itself, the Auto*Lock feature can be viewed as the "emergency break" to avoid run-time errors. Also, transactions can only be controlled by the programmer him/herself, through locking all required databases before the updates start.

Of course, if the AUTOLOCK is active, you may manually invoke the AUTORLOCK() instead of RLOCK() and AUTOFLOCK() instead of FLOCK(), to control the program flow in the same way, while taking advantage of the wait-until-success feature.

Tuning:

You may log AutoRlock(), AutoFlock(), AutoAppend() and AutoUnlock() and their failure by assigning any valid file name (optionally with path) to

```
_aGlobSetting[GSET_C_AUTOLOCK_PROT] := fileName   // def = ""
```

When this protocol file already exist, messages will be appended.

If the lock fails, the process sleeps for a small time period and then retry the lock anew. The sleep period is defined by

```
_aGlobSetting[GSET_N_AUTOLOCK_SLEEP] := milliSec  // def = 150
```

When these retries exceeds info-time-out period, a pop-up window informs user about waiting for lock. This info period is set by

```
_aGlobSetting[GSET_N_AUTOLOCK_INFO] := seconds    // def = 5
```

The pop-up info message (e.g. "Waiting for Lock") is displayed for

```
_aGlobSetting[GSET_N_AUTOLOCK_INFOWT] := seconds  // def = 2
```

and then disappears, continuing with retry. When the total time-out of SET AUTOLOCK TO (e.g. 10 seconds) expires too, user can decide to continue with retry, ignore this lock, jump per BREAK to recover of next BEGIN SEQUENCE or exit the application.

For background or Web/CGI applications, where user info and actions are not desired, set long (or forever) AUTOLOCK period, e.g. SET AUTOLOCK TO 3600 (or SET AUTOLOCK TO 0) and disable pop-up info by

```
_aGlobSetting[GSET_N_AUTOLOCK_INFO] := 0
```

Example 1:

```
BEGIN SEQUENCE
   USE mydata SHARED                 // open mutiuser
   SET AUTOLOCK -99                  // disable AUTOLOCK
   replace name with "Miller"        // run-time error !

   SET AUTOLOCK 0                    // enable AUTOLOCK
   replace name with "Miller"        // o.k. or --> RECOVER
   return
RECOVER using cText
   ? "sorry, could not ", cText, " the database " + DBF()
END SEQUENCE
```

Example 2:

```
SET EXCLUSIVE OFF                    // enable multiuser
* SET AUTOLOCK TO 10                 // the default setting
USE article INDEX article
SEEK 12345
if !found() ; return ; endif

DELETE                               // one AUTORLOCK
REPLACE amount with 0, price with 0  // two AUTORLOCKs

while !RLOCK() ; enddo               // one LOCK only
* or: AUTORLOCK()                    // "smart" RLOCK()
DELETE
REPLACE amount with 0, price with 0
COMMIT ; UNLOCK
* or: AUTOUNLOCK()                   // "smart" UNLOCK
```

Classification:

database

Compatibility:

SET AUTOLOCK is a superset and combination of FoxPro's SET LOCK and SET REPROCESS. This feature is not available in C5 and only partially available in VO. See also SET COMMIT for additional tuning.

Source code:

The functions are available in <FlagShip_dir>/system/autolock.prg

Translation:

SET (_SET_AUTOLOCK, <expN>| <expL>)

Related:

AUTOxLOCK(), SET(), SET MULTILOCKS, SET COMMIT, RLOCK(), FLOCK(), oRdd:ConcurrencyControl

SET BELL

Syntax:

`SET BELL on|OFF|(<expL>)`

Purpose:

Toggles the sounding of the bell in READ.

Description:

When ON, the bell sounds if a character is being entered into a GET field which does not conform to the PICTURE clause or if the entry is out of the RANGE limit. It also sounds when a character is entered at the last position of a GET. When using the alternative parenthesized <expL>, TRUE is the same as ON.

To sound the bell explicitly, you can use either CHR(7) or the TONE() function. This bell does not depend on the state of SET BELL.

Example:

```
SET BELL ON                             // enable bell
SET FORMAT TO articles
READ
SET FORMAT TO
SET BELL OFF                            // disable bell

IF LASTKEY() = K_ESC
   ?? CHR(7)                            // sound a bell
   @ MAXROW(),0 SAY "Abort - are you sure (y/n) ?"
   IF UPPER(CHR(INKEY(0))) == "Y"
      QUIT
   ENDIF
ENDIF
```

Classification:

programming

Compatibility:

The ability to sound a bell depends on the terminfo definition and/or the terminal emulation software, if used. Some terminals use an "optical bell" which flashes the screen output instead of sounding an acoustic bell.

Translation:

SET (_SET_BELL, .on.)

Related:

SET CONFIRM, CHR(), TONE()

SET CENTURY

Syntax:

`SET CENTURY on|OFF|(<expL>)`

Purpose:

Toggles the display and input of century digits for date values.

Arguments:

ON/OFF activates or suppresses the output of century digits. Alternatively, the parenthesized <expL> may be used, whereby logical TRUE is the same as ON.

Description:

The stored date values always contain the complete year information, including the century. The information is stored as a LONG value (or 8 bytes ASCII in .dbf fields) representing the number of days since January 1, 0001. The supported date range in FlagShip is therefore from 01/01/0001 up to 12/31/9999.

When CENTURY is OFF, only the two last digits of the year are displayed or can be entered. Setting CENTURY to ON changes the date format displayed to contain four digits for the year.

Example:

```
? DATE()                                && 07/22/93
SET CENTURY ON
? DATE()                                && 07/22/1993
SET DATE GERMAN
SET CENTURY (.F.)
? DATE()                                && 22.07.93
```

Classification:

programming

Compatibility:

Clipper supports date values from 01/01/0100 to 12/31/2999.

Translation:

__SETCENTURY (.on.)

Related:

SET DATE, SET EPOCH, CTOD(), DATE(), DTOC(), DTOS(), DAY(), MONTH(), YEAR(), SET()

SET CHARSET

Syntax:

```
SET CHARSET|KEYTRANSL [TO] ISO|ANSI
SET CHARSET|KEYTRANSL [TO] PC8|ASCII|OEM
```

see SET KEYTRANSL and SET GUICHARSET below

SET COLOR TO

Syntax:

```
SET COLOR|COLOUR TO
      [<standard>
       [,<enhanced>
        [,<border>
         [,<background>
          [,<unselected>
           [,<extra>
            [,<disabled>
             [,<unselWindow>]]]]]]]] | (<expC>)
```

Purpose:

Changes the screen color setting.

Arguments:

Each argument of the list specifies a list of color settings for the five types of screen painting activity. Each argument contains a color pair containing the foreground and background color, separated by a slash (/).

Output	Pos	Color pair	Usage i.e.
standard	1	foreground/background	SAY, ?
, enhanced	2	foreground/background	GET, MENU, ACHOICE
, border	3	foreground/background	boxes etc.
, background	4	foreground/background	hot-key, accelerator
, unselected	5	foreground/background	READ
, extra	6	foreground/background	reserved
, disabled	7	foreground/background	disabled GET, PROMPT
, unselWidow	8	foreground/background	GETs in unsel.window

If no argument is given, the default color setting is reset to to defaults specified in `_aGlobSetting[GSET_G_AC_DEFCOLOR]` for GUI and `_aGlobSetting[GSET_T_C_-DEFCOLOR]` for Terminal i/o, see below.

Skipping a foreground or background color within a setting does not change the default or previously set color.

Options:

<**standard**> is the color pair (foreground/background) used to paint with all console and full-screen commands' and functions' output, such as ?, ??, @..SAY, @..BOX, @..PROMPT, @..CLEAR, CLEAR SCREEN, ACHOICE(), DBEDIT(), MEMOEDIT() etc. It can be set explicitly using the SETSTANDARD command.

<**enhanced**> specifies a color pair, which is used for painting highlighted displays, like the active GET field in READ, the light bar in MENU TO, DBEDIT(), and ACHOICE(). It can be set explicitly using the SETENHANCED command.

<**border**> is not supported in FlagShip (nor in Clipper). It specify the color to paint the area around screen or the background color for some other xBASE dialects. This color pair is used in FlagShip for other purposes, like the border in Popup's or in ACHOICE().

<**background**> is originally used by some other xBASE dialects for CGA cards and not supported as such in FlagShip nor Clipper. In FlagShip, it is used for other purposes, like the hot-key color in Terminal i/o mode.

<**unselected**> is a color pair used to display currently inactive GET fields and un-selectable array members in ACHOICE(). Can be set explicitly using the SETUNSELECTED command.

<**extra**> is a color pair reserved for future use.

<**disabled**> is a color pair used to display disabled GET fields. Apply in GUI mode only, ignored otherwise. Used also for un-selectable PROMPT items, even in Terminal i/o mode.

<**unselWindow**> is a color pair used to display GET fields when the focus is taken away from the application window. If not specified, the GET color remain unchanged. Apply in GUI mode only, ignored otherwise.

(<**expC**>) is a character string enclosed in parentheses containing the color settings. This allows the color settings to be specified as an expression in place of a literal string or a macro variable. Instead of character string, you may alternatively use an array of RGB triplets, or a Color or ColorPair object variable.

Description:

SET COLOR is a synonym for the SETCOLOR() function that defines colors for subsequent screen painting activities. Each argument can specify foreground (the displayed text) and background (the color underlying the text). Spaces are displayed as background only.

Note: In the case of color settings, a list containing commas in a macro variable can be used.

Attributes: The foreground color setting also supports blinking (*) and high intensity (+) attributes. In Terminal i/o mode, these attributes affect only the foreground color, even if mentioned with the background color of the pair. In GUI mode, the high intensity attribute can also be used for background color, the blinking attrib is ignored. High intensity enhances brightness of painted text on a monochrome display or changes the hue of the specified color on color monitors. The blinking attribute causes the foreground text to flash on and off at a set hardware interval.

Colors in FlagShip may be specified by a string containing letters, numbers or RGB string, or optionally by an RGB array.

- The letters and numbers are fix and specify 16 different colors, available on any VGA screen. Compatible to other xBase dialects.
- A RGB string triplet is similar to the common HTML notation. The color is defined as a string "#RRGGBB" starting with "#" followed by 6 hexadecimal characters

(each 0..9,A..F). The first two hex chars ("00" to "FF") specify the amount of red portion in the resulting color, the second two characters the portion of green, the last two hex characters the portion of blue color.

- Instead of a string, optionally an array of RGB triplets can be used, e.g. when calculating colors. This array is either:

* one-dimensional array with three numeric elements (ea 0..255), specifying the red, green and blue color portion of foreground color, e.g. aColor := {128,128,0} ; @..SAY.. COLOR (aColor)

* or a 2-dimensional array containing two color pair triplets for foreground and background of the standard color, e.g. aColor := {{0,0,0},{0,0,128}} ; SET COLOR TO (aColor)

* or 3-dimensional array specifying each corresponding color pair, e.g. a2:= {{0, 0, 0}, {0, 0, 128}}; a5:= {{RGBCOLOR_WHITE},{0,0,128}} aColor := {NIL, a2, NIL, NIL, a5} ; @..GET... COLOR (aColor) which set the enhanced and unselected color pairs only.

* For some standard color triplets (the sub-array of 3 elements), there are predefined constants RGBCOLOR* in #include "color.fh"

You may use letters, numbers and RGB strings interchangeably in a single specification, e.g. "B/N, 1/R, #00FF00/W, 12/#C0C0C0" etc.

The RGB notation allows a combination of 16 million colors and is fully supported in GUI mode, provided your GUI environment is set to 16 mio (or more) colors. If your environment support 256 colors only, you should preferably use hex values 00,33,66,99,CC,FF in a triplets combination, or the standard RGB triplets as given in the table below. In Terminal i/o mode, the Symbol/Letter notation is commonly used. If given in RGB notation, the closest color letter is calculated from the RGB triplet. Conversion failure is displayed when the developer's mode is set by FS_SET("devel",.T.)

The standard colors are:

Color *	Symbol/Letter	Num.Code	RGB String	RGB Array
Black	N	0	#000000	{ 0, 0, 0}
Blue	B	1	#000080	{ 0, 0,128}
Green	G	2	#008000	{ 0,128, 0}
Cyan	BG or GB	3	#008080	{ 0,128,128}
Red	R	4	#800000	{128, 0, 0}
Magenta	RB or BR	5	#800080	{128, 0,128}
Brown (dark yellow)	GR or RG	6	#808000	{128,128, 0}
White (light gray)	W or RGB	7	#DCDCDC	{220,220,220}
Gray (dark)	N+	8	#808080	{128,128,128}
Bright blue	B+	9	#0000FF	{ 0, 0,255}
Bright green	G+	10	#00FF00	{255,255, 0}
Bright cyan	BG+	11	#00FFFF	{ 0,255,255}

Bright red	R+	12	#FF0000	{255, 0, 0}
Bright magenta	RB+ or BR+	13	#FF00FF	{255, 0,255}
Bright yellow	GR+ or RG+	14	#FFFF00	{255,255, 0}
Bright white	W+ or RGB+	15	#FFFFFF	{255,255,255}
Mid gray	W-		#C0C0C0	{192,192,192}
Blank	X			
Underline (mono)	U			
Reverse Video	I			
Standard background	?		RGBSTRING_BG	{RGBCOLOR_BG}

* Note: in Terminal i/o mode for Unix/Linux, the proper output depends on the correct setting of the terminfo variables setf (set foreground), setb (set background color), bold (set high intensity), blink (set blinking), invis (invisible), rev (reverse), smul (underline) and sgr0 (disable setting) for the current terminal TERM. See also section SYS.

The "?" symbol is replaced by standard background color, taken from m->oApplic:ColorBackground property. It corresponds to main window color in GUI mode, or "N" (= #000000) in other i/o modes, but may be re-defined by any valid value upon request.

The current color setting is always active for Terminal i/o mode. In GUI mode, it is considered only if SET GUICOLOR is ON or when the GUI color is explicitly specified, e.g. in @..GET..GUICOLOR... This is because GUI design rules recommend not to use colors at all, except when explicitly required.

Tuning:

You may set your own default colors 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 w/o argument or SetColor(-1) to change color defauls.

Example 1:

```
STATIC colors [3]
IF ISCOLOR()
   colors[1] := "W+/B,R+/GR,,,B/W"          // standard
   colors[2] := "W/B,N/W,,,N/BG"            // other color
   colors[3] := "GR+/B,R+/B"                // messages
ELSE
   colors[1] := "W+,/W,,N"                  // standard
   colors[2] := "W/N,N/W,,,N/W"             // other color
   colors[3] := "U,W*"                      // messages
ENDIF

SET COLOR TO (colors[1])
CLEAR SCREEN
@ 1,1 SAY "name     " GET FIELD->name
@ 2,1 SAY "address  " GET ADR->address VALID check()
READ
```

```
IF LASTKEY() = K_ESC
   SET COLOR TO (colors[3])
   @ MAXROW(),0 SAY "Are you sure to quit (y/n) ? "
   IF (UPPER(CHR(INKEY(0))) == "Y"
      QUIT
   ENDIF
   SET COLOR TO (colors[1])
ENDIF

FUNCTION check
LOCAL actcolor := SETCOLOR()
IF EMPTY(ADR->address)
   SET COLOR TO (colors[3])
   SETENHANCED                             // enhanced color
   ?? CHR(7)                               // bell
   @ MAXROW(),0 SAY "Address must be given!"
   SETSTANDARD
   SET COLOR TO &actcolor                  // or TO (actcolor)
   INKEY(5)                                // wait 5 seconds
   @ MAXROW(),0                            // clear msg
   RETURN .F.
ENDIF
RETURN .T.
```

Example 2:

```
#include "color.fh"
@ 5,2 SAY "hello light blue on std. GUI background" ;
  COLOR "B+/N" ;                              // Terminal mode
  GUICOLOR {{0,0,255},{RGBCOLOR_BG}}          // GUI mode
? "hello dark red on std. GUI background" ;
  GUICOLOR ("R/" + RGBSTRING_BG) COLOR ("R/N")
```

Example 3:

See also example in SETCOLOR() which allows user to choose the preferred color setting.

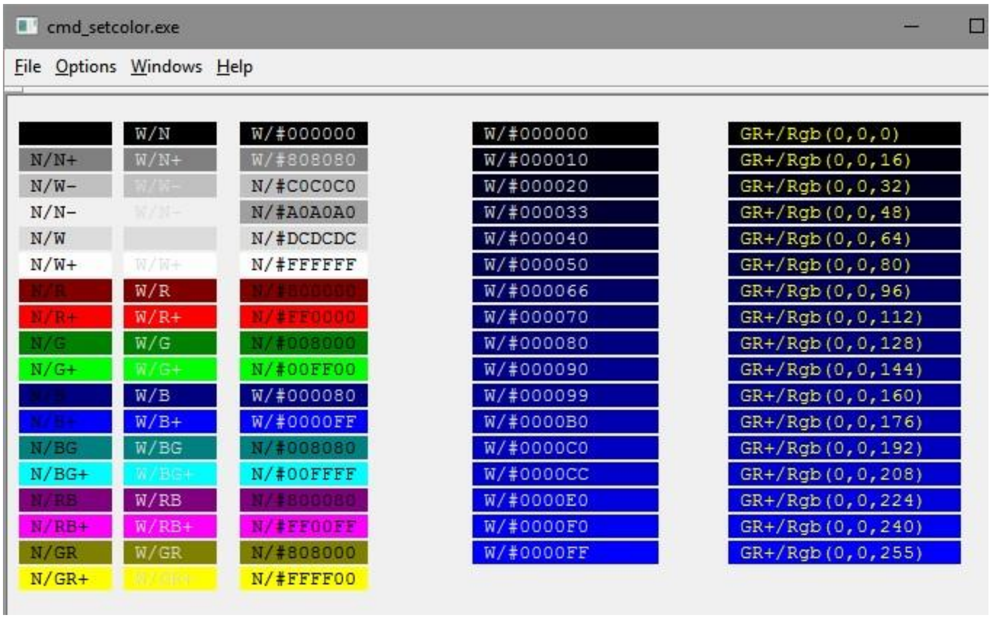
Example 4:

Display all standard and some RGB colors as background

```
#include "color.fh"
aStd1 := {"N","N+","W-","N-","W","W+","R","R+","G","G+", ;
          "B","B+","BG","BG+","RB","RB+","GR","GR+"}
aStd2 := {"#000000","#808080","#C0C0C0","#A0A0A0","#DCDCDC", ;
          "#FFFFFF","#800000","#FF0000","#008000","#00FF00", ;
          "#000080","#0000FF","#008080","#00FFFF","#800080", ;
          "#FF00FF","#808000","#FFFF00" }
aCol3 := {"00", "10", "20", "33", "40", "50", "66", "70", "80", ;
          "90", "99", "B0", "C0", "CC", "E0", "F0", "FF"}

SET FONT "courier",10
for ii := 1 to len(aStd1)    // using std.symbols
   fg := "N/"
   @ ii,1 SAY padr(" " + fg + aStd1[ii], 8);
          COLOR (fg+aStd1[ii]) GUICOLOR (fg+aStd1[ii])
   fg := "W/"
   @ ii,10 SAY padr(" " + fg + aStd1[ii], 8) ;
           COLOR (fg+aStd1[ii]) GUICOLOR (fg+aStd1[ii])
```

```
next
for ii := 1 to len(aStd2)    // using std.symbols + RGB string
   fg := if(ii < 3 .or. ii == 11 .or. ii == 12, "W", "N")
   bg := "/" + aStd2[ii]
   @ ii,20 SAY padr(" " + fg+bg, 11) COLOR (fg+bg) GUICOLOR (fg+bg)
next

SET GUICOLOR ON
for ii := 1 to len(aCol3)    // using user defined RGB string
   fg := "W/"
   bg := "#0000" + aCol3[ii]
   @ ii,40 SAY padr(" " + fg + bg,16)  COLOR (fg+bg)
next

iRow := 1
for ii := 0 to 256 step 16   // using array of calcul. RGB colors
   aColor := {{RGBCOLOR_YELLOW},{ 0,0,min(ii,255) }}
   @ iRow++,62 SAY padr(" GR+/Rgb(0,0," + ltrim(aColor[2,3]) + ;
               ")",20) COLOR (aColor)
next
SET GUICOLOR OFF
setpos(20,0)
wait
```

Output:



```
          W/N      W/#000000     W/#000000     GR+/Rgb(0,0,0)
N/N+      W/N+     W/#808080     W/#000010     GR+/Rgb(0,0,16)
N/W-      W/W-     N/#C0C0C0     W/#000020     GR+/Rgb(0,0,32)
N/N-      W/N-     N/#A0A0A0     W/#000033     GR+/Rgb(0,0,48)
N/W                N/#DCDCDC     W/#000040     GR+/Rgb(0,0,64)
N/W+      W/W+     N/#FFFFFF     W/#000050     GR+/Rgb(0,0,80)
N/R       W/R      N/#800000     W/#000066     GR+/Rgb(0,0,96)
N/R+      W/R+     N/#FF0000     W/#000070     GR+/Rgb(0,0,112)
N/G       W/G      N/#008000     W/#000080     GR+/Rgb(0,0,128)
N/G+      W/G+     N/#00FF00     W/#000090     GR+/Rgb(0,0,144)
N/B       W/B      W/#000080     W/#000099     GR+/Rgb(0,0,160)
N/B+      W/B+     W/#0000FF     W/#0000B0     GR+/Rgb(0,0,176)
N/BG      W/BG     N/#008080     W/#0000C0     GR+/Rgb(0,0,192)
N/BG+     W/BG+    N/#00FFFF     W/#0000CC     GR+/Rgb(0,0,208)
N/RB      W/RB     N/#800080     W/#0000E0     GR+/Rgb(0,0,224)
N/RB+     W/RB+    N/#FF00FF     W/#0000F0     GR+/Rgb(0,0,240)
N/GR      W/GR     N/#808000     W/#0000FF     GR+/Rgb(0,0,255)
N/GR+     W/GR+    N/#FFFF00
```

Classification:

programming

Compatibility:

In Terminal i/o mode, colors are available only if both parameters "colors" and "pairs" are set in the terminfo (or FStinfo.src); refer to section SYS. You may determine the color capability using the function ISCOLOR(). The Unix curses does not support the black on black setting ("N/N" or "N+/N") since this is used as the default terminal color. To hide the output, use e.g. "N/X", "W/N" etc. or the "X" color setting.

In GUI mode, colors are considered only if SET GUICOLOR is ON, or when the special GUICOLOR clause (available in many commands) was specified. See also text.

The ability to display all colors specified or the additional attributes depends also on the hardware capabilities of the current terminal, the OS dependent curses library and/or the software setting of the used terminal emulation.

Color pairs 6..8 (extra, disabled and unselWindow) as well as the use of RGB triplets are available in FS5 only.

Translation:

SETCOLOR (expC)

Related:

SETCOLOR(), COLORSELECT(), SETSTANDARD, SETENHANCED, SETUNSELECTED, ISCOLOR()

SET COMMIT

Syntax:

`SET COMMIT [TO] DEFAULT | SECURE | FAST`

Purpose:

Sets the performance tuning for flushing of changed database records by automatically executed DbCommit(). Of course, any of these tuning switches _aGlobSetting[...] can also be set explicitly.

Arguments:

<**SET COMMIT [TO] DEFAULT**> resets defaults according to <FlagShip_dir>/ system/stdio.prg, which corresponds to standard Clipper behavior.

<**SET COMMIT [TO] SECURE**> is even more secure than <DEFAULT> setting but may be slower. It corresponds to former VFS7 behavior.

<**SET COMMIT [TO] FAST**> enables faster performance for most database operations, but may be less secure in some occurrences.

	DEFAULT	SECURE	FAST	See
_aGlobSetting[GSET_N_DBCOMMIT]	:= 4	4	1/4	(1)
_aGlobSetting[GSET_N_DBCOMMITALL]	:= 4	4	1/4	(2)
_aGlobSetting[GSET_L_DBCOMMIT_EXCL]	:= .F.	.F.	.F.	(3)
_aGlobSetting[GSET_L_DBCOMMIT_FLOCK]	:= .F.	.T.	.F.	(4)
_aGlobSetting[GSET_L_DBCOMMIT_SELECT]	:= .F.	.T.	.F.	(5)
_aGlobSetting[GSET_L_DBCOMMIT_SEEK]	:= .F.	.F.	.F.	(6)
_aGlobSetting[GSET_L_DBCOMMIT_SKIP]	:= .F.	.T.	.F.	(7)
_aGlobSetting[GSET_L_DBCOMMIT_SKIP0]	:= .T.	.T.	.T.	(8)
_aGlobSetting[GSET_L_DBCOMMIT_GOTO]	:= .T.	.T.	.F.	(9)
_aGlobSetting[GSET_L_DBCOMMIT_ORDER]	:= .F.	.T.	.F.	(10)
_aGlobSetting[GSET_L_DBCOMMIT_APPE]	:= .F.	.T.	.F.	(11)
_aGlobSetting[GSET_N_DBCOMMIT_UNLOCK]	:= 0	1	0	(12)

(1) The kind of flushing files by DbCommit() 1:asynchronous, 2:synchronous, 4:flush (default), 5:synchr+flush, see below. In MS-Windows, only 4 apply.

(2) The kind of flushing files by COMMIT or DbCommitAll(): 1:asynchronous, 2:synchronous, 4:flush (default), 6:synchr+flush, see below. In MS-Windows, only 4 apply.

(3) Perform COMMIT and/or DbCommit() or flushing by auto-DbCommit() also on exclusive open database

(4) Perform flushing by auto-DbCommit() also on FLOCKed database, otherwise with RLOCKed or APPENDed records only.

(5) Perform flushing by auto-DbCommit() on/before SELECT or DbSelectArea() or on alias-> for other workarea.

(6) Perform auto-DbCommit() on/before SEEK or FIND or DbSeek().

(7) Perform auto-DbCommit() on/before SKIP n or DbSkip(n).

(8) Perform auto-DbCommit() on/before SKIP 0 or DbSkip(0).

(9) Perform auto-DbCommit() on/before GOTO, GO BOTTOM, GO TOP, DbGoto(), DbGoTop(), DbGoBottom().

(10) Perform auto-DbCommit() on/before SET ORDER, DbSetOrder().

(11) Perform auto-DbCommit() after APPEND BLANK for the new record

(12) Perform auto-DbCommit() on/before UNLOCK for changed records: 0:don't flush, 1:auto-flush by DbCommit(), 2:auto-flush by DbCommitAll(). The SET AUTOCOMMIT ON/OFF changes this switch.

The value for _aGlobSetting[GSET_N_DBCOMMIT or GSET_N_DBCOMMITALL] is:

=1 use asynchronous, delayed flushing by kernel sync() which flushes all open files, but not if the last sync was already done within past <n> seconds, set by

```
aGlobSetting[GSET_N_DBCOMMIT_FLUSH] := 3.0 // def 3 sec
```

to increase performance. 0 disables this optimization and calls sync() always (comparable to mode 2). You may use this mode 1 to increase performance in some cases, by decreasing security. Not applicable in MS-Windows where mode 4 is always used.

=2 use synchronous flushing by immediate sync(). This ensures flush of all open files but may be time consuming especially with many open files. Not applicable in MS-Windows.

=4 use flushing by fsync() or _commit(fd) in Windows for all files used in this work area, i.e. the current .dbf, .dbt, .fpt, .dbv, .idx. This ensures security by flushing the current work area, but may be slightly slower than mode 1 when only few files are open. Mandatory in MS-Windows.

=5 combination of method 1 and 4. Not applicable in MS-Windows.

=6 combination of method 2 and 4. Not applicable in MS-Windows.

In **Unix/Linux**, there are different standard system mechanism available for flushing: either sync(), which asynchronously flushes buffers to disk by scheduled kernel request, or fsync() which flushes specific file buffers programmatically. They differ in performance and security: sync() may be faster with only few open files (system wide), whilst fsync() ensures flushing of files in current work area, the performance does not depend on system-wide open files and the time of flushing is exactly anticipated.

MS-Windows supports only file specific flushing by system function _commit(fd), so the _aGlobSetting[GSET_N_DBCOMMIT[ALL]] is always 4; changing this value is therefore ignored in Windows application.

Description:

With the default DBFIDX driver, the **current database** is only automatically flushed (committed) to hard disk by DbCommit() when:

- the current record was **changed** and not flushed yet,
- **and** the database was open in SHARE mode (modifiable for EXCL),
- **and** the current record is RLOCK()ed (modifiable also for FLOCK) or auto-locked by SET AUTOLOCK, or the record was APPENDed, or the record is not locked but the database was open EXCLusive and the switch (3) above is .T.,
- **and** any of database movement by SELECT, FIND, SEEK, SKIP, GOTO, DbSeek(), DbSkip(), DbGoto(), SET ORDER, DbSetOrder() (modifiable by 6..9 above) **or** alternatively SELECT or DbSelectArea() or aliasing foreign database (modifiable by 5 above) follows.

FlagShip allows you tune the flushing mechanism by above settings. The SET COMMIT command is for your convenience only, you may set any of these switches freely as you need.

Classification:

programming

Compatibility:

Available in FlagShip VFS8 and newer.

Translation:

```
SET COMMIT DEFAULT => Setflush(0)
SET COMMIT SECURE  => Setflush(1)
SET COMMIT FAST    => Setflush(2)
```

Source:

The behavior is user modifiable, the source is available in *<FlagShip_dir>/ system/setflush.prg*

Related:

COMMIT, GO*, SEEK, SKIP, UNLOCK, DbCommit(), DbCommitAll()

SET COORD

Syntax 1:

```
SET COORD [UNIT] TO ROWCOL | PIXEL | MM | CM | INCH
```

Syntax 2:

```
SET COORD [UNIT] TO
```

Purpose:

Set required coordinate units for screen and printer output row and column coordinates (for printer only with SET GUIPRINTER ON). Apply in GUI mode only, ignored otherwise. The default is ROWCOL, which is also set by syntax 2.

Arguments:

TO ROWCOL all subsequently given output coordinates are calculated as rows and columns according to the used font. Default setting.

TO PIXEL all subsequently given output coordinates are calculated in pixels or re-calculated for printer resolution.

TO MM all subsequently given output coordinates are re-calculated from mm (1 mm = 0.0397") to current screen or printer resolution.

TO CM all subsequently given output coordinates are re-calculated from cm (1 cm = 0.397 inch) to current screen or printer resolution.

TO INCH all subsequently given output coordinates are re-calculated from inch (1" = 2.54 cm) to current screen or printer resolution.

TO reset defaults to ROWCOL

Description:

SET COORD controls the behavior of given coordinate units. Apply in GUI mode only, ignored otherwise. These units are considered in all subsequent commands and functions with coordinate input, like @..SAY, @..GET, @..PROMPT, SETPOS(), DEVPOS(), MemoEdit() etc. and for returned values from COL() and ROW() functions. With the most commands you may override the current SET COORD setting by the PIXEL clause, or by the similar parameter of corresponding function.

Using TO ROWCOL (the default) is convenient in the most cases. With proportional fonts (see SET FONT and LNG.5.3.1-2) the character size may vary. To control the output exactly, you may use SET COORD TO PIXEL or SET PIXEL ON or corresponding PIXEL clause, whereby the coordinates are pixel oriented (pixel is a "dot on the screen", i.e. smallest single component of a digital image). Alternatively, you may force the output in mm, cm or inch by corresponding SET COORD TO...

The re-calculation from mm, cm or inch to pixel on screen (or dpi for printer) depends on the system API, which may be imprecise in some cases. For the screen output, FlagShip determines the desktop size in mm by oApplic:DesktopXmm and oApplic:DesktopYmm at program start (in initio.prg) and stores it in global array elements `_aGlobSetting[GSET_G_N_DESKTOP_X_MM]` and `_aGlobSetting [GSET_-G_N_DESKTOP_Y_MM]`. When you detect significant differences, you may set these array elements manually before using SET COORD TO MM|CM|INCH and displaying data. The physical size of printer sheet is determined at print-time from the printer API; these data are available after the user selects corresponding printer driver by oPrinter:Setup().

Example:

```
* _aGlobSetting[GSET_G_N_DESKTOP_X_MM] := 520   // optional
* _aGlobSetting[GSET_G_N_DESKTOP_Y_MM] := 324   // optional

@ 10,3 say "text1"              // output at line 10, column 3
SET COORD TO MM
@ row(), col() + 55 SAY "text2" // output at same line +5.5cm right
@ 25.4, 76.2 SAY "text3"        // output 1" from top, 3" from left
SET COORD TO
@ 10,15 say "text4"             // output at line 10, column 15
setpos(20,0)
wait
```

Classification:

programming, screen and printer output

Compatibility:

Available in VFS7 and newer only.

Translation:

```
SET ( _SET_COORD_UNIT, _SET_COORD_ROWCOL or _SET_COORD_DEF or 0 )
SET ( _SET_COORD_UNIT, _SET_COORD_PIXEL  or 1 )
SET ( _SET_COORD_UNIT, _SET_COORD_MM     or 2 )
SET ( _SET_COORD_UNIT, _SET_COORD_CM     or 3 )
SET ( _SET_COORD_UNIT, _SET_COORD_INCH   or 4 )
```

Related:

@...SAY, @..GET, SET PIXEL, SET GUIPRINTER, OBJ.Applic, OBJ.Printer

SET CONFIRM

Syntax:

`SET CONFIRM on|OFF|(<expL>)`

Purpose:

Determines if moving to the next field in GET/READ when the field is filled, or selecting an item in MENU TO by the first letter should be confirmed or done automatically.

Arguments:

ON/OFF activates or deactivates the requirement to press the ENTER <┘ key to leave a GET entry or the item selected in MENU TO. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

SET CONFIRM controls the behavior of leaving the current GET and MENU TO choice:

- When SET CONFIRM is OFF (the default), the user can type past the end of a GET and the cursor will move to the next GET if there is one; otherwise the current READ terminates. In MENU TO, pressing the first menu character selects the item found and terminates the choice.
- When ON is set, an exit key (e.g. ENTER, PgDn, PgUp etc.) must be pressed to leave the current GET. In MENU TO, pressing the first menu character selects the item found and positions the light bar on it. The user must press an ENTER <┘ key to confirm the choice.

Example:

```
LOCAL answer := "N"
SET CONFIRM ON
@ 10,10 SAY "Erase the temp*.txt files?" GET answer
READ
SET CONFIRM OFF
IF LASTKEY() = 13 .AND. upper(answer) $ "YJO"
   CLOSE ALL
   RUN rm temp*.txt
ENDIF
```

Classification:

programming

Compatibility:

The support of MENU TO is available in FlagShip only.

Translation:

SET (_SET_CONFIRM, .T. | .F.)

Related:

@...GET, READ, MENU TO, SET BELL

SET CONSOLE

Syntax:

`SET CONSOLE ON|off|(<expL>)`

Purpose:

Activates or deactivates console display to the screen.

Arguments:

ON/OFF activates or suppresses the output of console commands and functions to the screen. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

SET CONSOLE affects the screen display of all console commands (see LNG.5.1.1). Setting it to OFF and using the SET ALTERNATE, SET EXTRA, or SET PRINTER commands or TO.. clause suppresses the screen output and sends it to the printer or file only. Some console commands have a NOCONSOLE option, which has the same effect as temporarily setting SET CONSOLE OFF.

For console commands that accept input (like ACCEPT, INPUT, and WAIT), SET CONSOLE affects the display of the prompts as well as the input echo.

The full screen commands (like @..SAY, @..BOX, @..TO etc., see LNG.5.1.2) are not affected by SET CONSOLE but may be re-routed to printer or file using the SET DEVICE command.

Example:

```
SET CONSOLE OFF
USE stock
LIST item, volume FOR volume > 100 TO PRINT
USE
SET CONSOLE ON
```

Classification:

programming

Translation:

SET (_SET_CONSOLE, .T. | .F.)

Related:

SET DEVICE, SET ALTERNATE, SET EXTRA, SET PRINTER

SET COORDINATE UNIT

Syntax:

```
SET COORDINATE [UNIT] [TO] [ROWCOL]
SET COORDINATE [UNIT] [TO] PIXEL| MM | CM | INCH |
        ( <expN> )
```

Purpose:

Sets the unit for subsequently given screen (and printer with active PrintGui() output) coordinates. Applicable in GUI mode only.

Arguments:

ROWCOL	all subsequent coordinates are in common rows and columns
PIXEL	all subsequent coordinates are in pixels
MM	all subsequent coordinates are millimeter
CM	all subsequent coordinates are centimeter (ea 10 mm)
INCH	all subsequent coordinates are in inch (ea 25.4 mm)
<expN>	parenthesized numeric value, e.g. UNIT_ROWCOL, UNIT_MM, UNIT_CM, UNIT_INCH, UNIT_PIXEL, UNIT_DOTS (specified in the set.fh include file)

Description:

SET COORDINATE is equivalent to SET UNIT command, see description there. SET COORD TO PIXEL is equivalent to SET PIXEL ON, SET COORD TO ROWCOL is equivalent to SET PIXEL OFF

Compatibility:

Available in FlagShip VFS7 and later only.

Translation:

```
SET ( _SET_COORD_UNIT, expN)
```

Related:

SET PIXEL, SET()

SET CURSOR

Syntax:

`SET CURSOR ON|off|(<expL>)`

Purpose:

Sets cursor to be visible or invisible.

Arguments:

ON/OFF activates or deactivates cursor visibility. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

SET CURSOR OFF hides the cursor, although it still exists, which means that editing can be done with the cursor being invisible.

SET CURSOR can be used to suppress displaying the cursor, except when editing text. Some commands and functions (like MENU TO, ACHOICE(), DBEDIT() etc.) will disable the cursor automatically by default.

Because in practice controlling cursor visibility depends on the terminal hardware and the terminfo description, FlagShip will set the invisible cursor to MAXROW(), MAXCOL(). The current COL() and ROW() values are not affected by the cursor ON/OFF state.

SET CURSOR ON/OFF is considered in Terminal i/o mode. For GUI mode, use SET GUICURSOR instead, where you can also set the text cursor at specific position or shape by SetGuiCursor(), or set the mouse cursor shape by MsetCursor().

Example:

```
SET CURSOR OFF
CLS
@ 1,1 TO 20,60 DOUBLE
SET CURSOR ON
Name = SPACE(15)
@ 10,10 SAY "Enter name: " GET Name
READ
SET CURSOR OFF
```

Classification:

programming

Compatibility:

Most Unix terminals (curses libraries) cannot disable the cursor, so the cursor stays visible at the bottom rightmost position of the screen. For GUI mode, use SET GUICURSOR instead.

Translation:

SETCURSOR (1 | 0)

Related:

SET CONSOLE, SETCURSOR(), SETPOS(), SET GUICURSOR, SetGuiCursor()

SET DATE

Syntax 1:

SET DATE [TO] AMERICAN | ansi | british | french | german | italian | japan | usa

Syntax 2:

SET DATE FORMAT [TO] <expC>

Purpose:

Sets the format for displaying and inputting date values.

Arguments:

SET DATE TO AMERICAN, ANSI, GERMAN etc. specifies the format of input and output date values:

SET DATE	Output	SET CENTURY ON
AMERICAN	mm/dd/yy	mm/dd/yyyy
ANSI	yy.mm.dd	yyyy.mm.dd
BRITISH	dd/mm/yy	dd/mm/yyyy
FRENCH	dd/mm/yy	dd/mm/yyyy
GERMAN	dd.mm.yy	dd.mm.yyyy
ITALIAN	dd-mm-yy	dd-mm-yyyy
JAPAN	yy/mm/dd	yyyy/mm/dd
USA	mm-dd-yy	mm-dd-yyyy

Arguments:

SET DATE FORMAT TO <**expC**> defines a character expression that directly specifies the date format. <expC> must be a string of 12 or fewer characters. Upper/lower letters D, M and Y specify the position of day, month, and year digits displayed. Other characters in the string are copied into date values displayed and are used as delimiters.

The FlagShip run-time system analyzes for proper formats and reports errors in developer mode.

Description:

Using SET DATE allows the control of date formatting in programs ported in different countries.

Example 1:

```
? DATE()                                && 07/26/93
SET DATE ANSI
? DATE()                                && 93.07.26
SET DATE BRITISH
? DATE()                                && 26/07/93
SET DATE FRENCH
? DATE()                                && 26/07/93
SET DATE ITALIAN
? DATE()                                && 26-07-93
SET DATE GERMAN
```

```
? DATE()                               && 26.07.93
SET CENTURY ON
? DATE()                               && 26.07.1993
```

Example 2:

Get the date format from a shell environment variable:

```
LOCAL lang := UPPER(GETENV("LANG"))
DO CASE
CASE SUBSTR(lang,1,4) = "BRIT"
   SET DATE BRITISH
CASE SUBSTR(lang,1,4) = "GERM"
   SET DATE GERMAN
OTHERWISE
   SET DATE USA
ENDCASE
```

Classification:

programming

Compatibility:

The JAPAN and USA clauses and syntax 2 are new in FS4. FlagShip supports date values from 01/01/0001 to 12/31/ 9999.

Translation:

```
SET DATE TO AMERICAN   => _DFSET("mm/dd/yyyy", "mm/dd/yy")
SET DATE TO GERMAN     => _DFSET("dd.mm.yyyy", "dd.mm.yy")
SET DATE TO USA        => _DFSET("mm-dd-yyyy", "mm-dd-yy")
SET DATE FORMAT TO (expC)   => SET(_SET_DATEFORMAT, expC)
```

Related:

SET CENTURY, SET EPOCH, CTOD(), DATE(), DTOC(), DTOS(), @...SAY..PICTURE, @...GET, READ, TRANSFORM()

SET DB3COMPAT

Syntax:

`SET DB3COMPAT on|OFF|(<expL>)`

Purpose:

Sets/enables dBaseIII+ database compatibility of modification date.

Arguments:

ON/OFF enables/disables the dBase3 compatibility of modification date in the .dbf file. The default is OFF. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

On every change of the .dbf file, the database header is updated by the current date (binary as YY MM DD in byte 1,2,3). It can be determined by LUPDATE().

The default dBaseIII, Fox and Clipper year storage is 00..99 where LUPDATE() adds 1900 to (independent of SET EPOCH), hence the specs do not consider 21th century at all.

FlagShip is fully Y2K conformant. With the default SET DB3COMPAT OFF, FlagShip supports year storage 00..FFh (0..255), which then directly supports years 1900 to 2155 and is displayed correctly by LUPDATE(); also in Clipper.

dBaseIII+ however does not like year > 99 and deny to open this database. So if you wish to open the by FlagShip modified .dbf in dBASEIII+ too, use SET DB3COMPAT ON.

You may retrieve the status by SET(_SET_DB3COMPAT) or set it by SET(_SET_DB3-COMPAT, .T. | .F.).

Example:

```
? set( _SET_DB3COMPAT )      // .T.
? date()                     // 04/15/11
USE mydbf
? Lupdate()                  // 03/27/11
append blank
? Lupdate()                  // 04/15/11

SET DB3COMPAT OFF
? set( _SET_DB3COMPAT )      // .F.
? Lupdate()                  // 04/15/11
append blank
? Lupdate()                  // 04/15/11
```

Classification:

programming, database

Compatibility:

New in FS7, backward compatible to former FlagShip releases. Note that FlagShip saves the header on any .dbf update, Clipper at closing the database.

Translation:

```
Set ( _SET_DB3COMPAT, .T. | .F. )
```

Related:

USE, DbUseArea(), APPEND BLANK, REPLACE, Lupate()

SET DBREAD
SET DBWRITE

Syntax 1:

```
SET DBREAD ANSI|ISO
SET DBREAD PC8|ASCII|OEM
```

Syntax 2:

```
SET DBWRITE ANSI|ISO
SET DBWRITE PC8|ASCII|OEM
```

Purpose:

Change the behavior how to read from or store data into database. This is a special case of SET ANSI ON/OFF.

Arguments:

ANSI|ISO activates the automatic translation for reading or writing data from/to database.

PC8|ASCII|OEM deactivates the automatic translation for reading or writing data from/to database. This is the default setting.

Description:

SET DBREAD and SET DBWRITE is a splitted behavior of SET ANSI to perform either read or write translation if both are not desired. SET DBREAD ANSI + SET DBWRITE ANSI is equivalent to SET ANSI ON, SET DBREAD PC8 + SET DBWRITE OEM is equivalent to SET ANSI OFF which is the default.

With SET DBREAD ANSI, a database access of character or memo translates the PC8/ASCII/OEM charset via Oem2Ansi() into ANSI/ISO charset (used for display in GUI mode or in X11 terminal without a corresponding mapping).

With SET DBWRITE ANSI, the replaced a char or memo field in the database will be first translated by Ansi2oem() from ANSI to PC8.

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.

Note: both the FS4 and Clipper always use PC8/ASCII charset in the database, i.e. chr(132) for a-umlaut.

Example:

See *<FlagShip_dir>examples/setansi.prg* for a complete example with description

Classification:

programming, database

Compatibility:

New in FS5

Related:

SetAnsi(), SET ANSI, Ansi2oem(), Oem2Ansi(), SET SOURCE, SET KEYTRANSL|CHARSET,

SET DECIMALS TO

Syntax:

`SET DECIMALS TO [<expN>]`

Purpose:

Sets the number of decimal places for displaying the results of numeric expressions.

Options:

<**expN**> is the number of decimal places to display. The default value on start-up is two. If <expN> is not given, SET DECIMALS is set to 0.

Description:

SET DECIMALS and the number of displayed decimals depend on the state of SET FIXED:

When FIXED is OFF (the default),

- SET DECIMALS affects the minimum number of decimal digits displayed by the EXP(), LOG(), SQRT() functions, the division operations (`/`, `%`, `/=` or `%=`) and exponentiation (`**`, `^` or `**=`).
- On assignment (`:=` or `=`), the number of decimal digits of the variable or constant is stored in the receiving variable structure.
- On addition and subtraction (+, -, ++, --, += or -=), the number of decimal places of the operand with a greater number of decimal places is stored.
- On multiplication (* or `*=`), the sum of decimal places of both operands is stored.

By setting FIXED ON, the results of all numeric expressions are displayed according to SET DECIMALS.

SET DECIMALS and SET FIXED only affect the way numbers are displayed (or strings created by STR*(), PAD*(), TRANSFORM() etc.) and have no effect on the precision of numeric calculations.

Example:

```
LOCAL a := 2, b := 3.456
? SQRT(2), a, b                         && 1.41  2  3.456
SET DECIMALS TO 6
? 10/3, a, b                            && 3.333333  2  3.456
SET FIXED ON
? a, b                                  && 2.000000  3.456000
```

Classification:

programming

Translation:

SET (_SET_DECIMALS, expN)

Related:

SET FIXED, @..SAY..PICTURE, @..GET..PICTURE, TRANSFORM()

SET DEFAULT TO

Syntax:

`SET DEFAULT TO <path>|(<expC>)`

Purpose:

Sets the directory where files are saved and created.

Arguments:

<**path**> specifies the directory. As opposite to SET PATH, the SET DEFAULT may include only one path.

The "\" signs are automatically translated to "/" and vice versa. For lower/upper path translation on Linux, where names are case sensitive, use FS_SET ("pathlower"**|**"pathupper"); for the substitution of a DOS drive letter, the environment variable x_FSDRIVE can be used.

SET DEFAULT TO with no argument releases the default path and the current Unix or Windows directory becomes the default.

Description:

The default directory, at the beginning of a FlagShip program is the current Unix/Windows directory. This default directory can be changed with SET DEFAULT.

When accessing files, the DEFAULT directory is searched first. To specify additional directories to search, the SET PATH command may be used. The RUN command is not affected by SET DEFAULT nor by SET PATH

SET DEFAULT is meant primarily to specify the location where files are created, saved and searched. SET DEFAULT does not change the current Unix/Windows directory.

Example:

```
PUBLIC FlagShip
? FILE("article.dbf")                               // .F.
#ifdef FS_WIN32
  SET DEFAULT TO "D:\smith\am"                      // Windows
  ? FILE("article.dbf"), FILE("Article.Dbf")        // .T.  .T.
#else
  SET DEFAULT TO "/usr/users/smith/am"              // Linux
  ? FILE("article.dbf"), FILE("Article.Dbf")        // .T.  .F.
  FS_SET ("lower", .T.)
  FS_SET ("pathlower", .T.)
  ? FILE("Article.Dbf")                             // .T.
#endif
```

Classification:

programming

Compatibility

FlagShip supports the automatic conversion of the otherwise case sensitive Unix path names and the substitution of DOS/Windows drive letters, see FS_SET() and LNG.9.5.

Translation:

```
SET ( _SET_DEFAULT, expC )
```

Related:

SET PATH, CURDIR(), FS_SET(), PUBLIC FlagShip. #ifdef FlagShip

SET DELETED

Syntax:

`SET DELETED on|OFF|(<expL>)`

Purpose:

Toggles the filtering of deleted records.

Arguments:

ON/OFF ignores or processes deleted records. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

SET DELETED ON causes most commands to ignore the deleted records; the database seems to include only undeleted records. The SET DELETED ON command is equivalent to SET FILTER TO .NOT. DELETED().

SET DELETED ON has no effect on indexing operations using INDEX ON or REINDEX. The RECALL ALL command recalls all records which have been DELETED().

If a record is referenced by its record number (e.g. the GOTO command or the RECORD in <scope> clause), the record is available even when SET DELETED is set ON.

Example:

```
SET DELETED ON
USE article
DELETE RECORD 34
COUNT TO undel
? undel, LASTREC()                    &&  99  100
SET DELETED (.F.)                     && or: DELETED OFF
COUNT TO all
? all, RECCOUNT()                     && 100  100
```

Classification:

programming, database

Translation:

SET (_SET_DELETED, .T. | .F.)

Related:

DELETE, INDEX, RECALL, SET FILTER, SET INDEX, USE, DELETED(), SET(), oRdd:Deleted

SET DELIMITERS

Syntax 1:

`SET DELIMITERS on|OFF|(<expL>)`

Syntax 2:

`SET DELIMITERS TO [<expC>|DEFAULT]`

Purpose:

Sets/enables delimiter characters for the display of GET entries in terminal i/o mode.

Arguments:

ON/OFF displays delimiters or suppresses the display. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Options:

TO <**expC**> is a character expression containing one or two characters. If there is only one character, it is used as both the beginning and the ending delimiter. If there are two, the first one becomes the beginning, and the other the ending delimiter. If there are more than two characters in the string, the first two are considered and the rest is ignored.

TO DEFAULT: Resets the delimiters to the default colons (::) value. SET DELIMITERS TO without parameters has the same effect.

Description:

The @...GET command can display delimiters that surround the GET edit field display. If DELIMITERS is ON, the delimiters add two characters to the length of the GET object display.

To suppress the visibility of the left, right, or both delimiters, spaces can be used as part of the character expression.

Normally, DELIMITERS are not necessary because FlagShip programs use the optically more attractive reverse video or enhanced color setting if INTENSITY is ON.

In GUI mode, the delimiters are not displayed (but the GET column is corrected, i.e. shifted one column right), since the GET itself use own GUI widgets (controls).

Example:

```
SET DELIMITERS ON
SET DELIMITERS TO "||"                  && chr(128), pipe
Name = "John    "
@ 10,10 SAY "Name" GET Name             && Result: Name |John    |
READ

SET DELIMITERS TO "><"
Name = "John    "
@ 10,10 SAY "Name" GET Name             && Result: Name >John    <
READ
```

Classification:

programming

Translation:

```
Set ( _SET_DELIMCHARS, expC )
Set ( _SET_DELIMITERS, .T. | .F. )
```

Related:

@...GET, READ, SET()

SET DEVICE TO

Syntax:

```
SET DEVICE TO SCREEN | printer [NEW]
SET DEVICE TO
```

Purpose:

Redirects the output of full-screen commands, like @..SAY to the screen or printer.

Arguments:

TO SCREEN: The screen is the default device. If SCREEN is specified as the device, all output from @...SAY goes to the screen.

TO PRINTER: redirects all @...SAY output to the device or file specified with SET PRINTER TO, and is not echoed to the screen. The SET MARGIN is obeyed. @...GETs are ignored. When PrintGui(.T.) or SET PRINTER GUI ON is active, the ASCII output is additionally redirected to file or device when SET PRINTER is ON.

TO PRINTER NEW causes the current printer file contents to be deleted , instead of appended to.

SET DEVICE TO disables @..SAY output to screen and to printer, but not to GUI printer if PrintGui() is active. You may use it when you print to GUI driver and don't wish to see the output on the sreen. Don't forgot to enable SET DEVICE TO SCREEN thereafter.

Description:

When @...SAY is sent to the printer, a formfeed character (the EJECT command) is sent each time when the printing row becomes less than in the previous command. EJECT resets the printing row and column (PCOL() and PROW() values) to zero also causing a formfeed. SETPRC() can be used to set the printing row and column to the desired value.

You may tune the printer device driver by FS_SET("prset") which may be advantageous when using proportional character set etc.

To redirect the @...SAY output to a text file, SET PRINTER TO <file> and SET DEVICE TO PRINTER may be used.

Example:

```
IF ISPRINTER()                          // FlagShip: always .T.
   SET DEVICE TO PRINTER
   @ 1,5 SAY "Time to go home!"
   EJECT
ENDIF
```

Classification:

programming, file access

Compatibility:

Note the default spooled printer output of FlagShip; for more information refer to SET PRINTER and LNG.5.1.6. The NEW clause is available in FS4 only.

Translation:

```
SET ( _SET_DEVICE, "SCREEN"|"PRINTER"|"")
```

Related:

@...SAY, EJECT, SET PRINTER TO, ISPRINTER(), PROW(), PCOL(), SETPRC(), SET(), FS_SET("prset")

SET DIRECTORY TO

Syntax:

`SET DIRECTORY [TO] [<expC>]`

Purpose:

Changes the current working directory.

Option:

<**expC**> specifies the path (and optional DOS drive) of the new current working directory.

When <expC> is not specified or is an empty string, the previous directory is selected.

Description:

On Unix/Windows, you cannot use RUN cd <expC> since the directory change applies to the current shell only and has therefore no influence on the application when the RUN command terminates. Use the SET DIRECTORY instead, which has the same effect as #Cinline / chdir (<expC>); / #endCinline.

Automatic path conversion to lowercase or uppercase with FS_SET("pathlow" **|** "pathupp") and drive substitution from x_FSDRIVE environment variables is supported.

Issuing SET DIRECTORY without arguments changes to the last directory before the previous SET DIRECTORY was executed.

Example 1:

```
? CURDIR()                               && /usr/peter/temp
SET DIRECTORY TO ../data
CURDIR()                                 && /usr/peter/data
SET DIRECTORY TO
? CURDIR()                               && /usr/peter/temp

SET DIRECTORY TO /tmp
? CURDIR()                               && /tmp
SET DIRECTORY TO /usr/john
? CURDIR()                               && /usr/john
SET DIRECTORY TO
? CURDIR()                               && /tmp
```

Example 2: *checks if given directory is available*

```
cDir1 := "..\a\b"
ok := IsDirAvail(cDir1)
? "Directory", cDir1, "available:", ok
cDir2 := "c:\tmp"
ok := IsDirAvail(cDir2)
? "Directory", cDir2, "available:", ok
? "current dir =", curdir()
wait
```

```
// -------------------------------------------------------
// checks if directory <cDirName> is available,
// returns .T. or .F., does not change current dir
FUNCTION IsDirAvail(cDirName)
   local ok, cCurDir
   cCurDir := curdir()
   ok := _setdir(cDirName)    // SET DIRECTORY ...
   if ok
      _setdir(cCurDir)        // back to current
   endif
return ok
```

Classification:

programming

Compatibility:

Compatible to DB4, not available in C5.

Translation:

_SETDIR (expc)

Related:

CURDIR(), SET DEFAULT, SET PATH

SET EJECT

Syntax:

`SET EJECT on|OFF|(<expL>)`

Purpose:

Performs automatic EJECT on full printer page in GUI mode for text output.

Arguments:

ON/OFF enables/disables the automatic EJECT. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

With PrintGUI(.T.) or when SET PRINTER GUI is ON, this SET EJECT performs automatic FormFeed (new page) when the line number exceeds printer's page limit. You may alternatively perform FormFeed manually by the EJECT command or oPrinter:GuiNewPage() method. SET EJECT is considered for text printing by ?, Qout() and @..SAY.. but not for other GUI drawings, where manual page control is required.

Example:

```
SET FONT "courier", 8
_aGlobSetting[GSET_G_N_ROW_SPACING] := -1   // reduce line spacing
SET CONSOLE OFF              // disable screen output
PrintGui(.T.)                // select & start GUI printer output
// SET MARGIN TO 5           // margin left = 5 chars
SET EJECT ON                 // autom. EJECT on full page

USE address
LIST Name, Address, Age      // prints to GUI/GDI printer

PrintGui()                   // flush to printer
SET CONSOLE ON               // enable screen output
SET FONT "courier", 10
_aGlobSetting[GSET_G_N_ROW_SPACING] := 2   // reset default spacing
```

Classification:

GUI printer output

Translation:

SET (_SET_EJECT, expL)

Related:

EJECT, SET PRINTER, SET GUIPRINTER, PRINTGUI(), Printer class

SET EOFAPPEND

Syntax:

`SET EOFAPPEND on|OFF|(<expL>)`

Purpose:

Enables/disables automatic APPEND BLANK before replacing record at EOF().

Arguments:

ON/OFF enables/disables the automatic APPEND BLANK. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

In some xBase dialects, REPLACEing a record in empty database, or when the record pointer is behind the last record (i.e. when EOF() reports .T.) will automatically invoke APPEND BLANK before REPLACE to avoid RTE (run-time-error) message.

You also may use this feature in your application by setting SET EOFAPPEND ON or the equivalent function SET(_SET_EOFAPPEND, .T.).

Example:

```
USE mydata
GO TOP                             // go to last record
SKIP                               // skip one behind
? EOF(), RecNo(), RecCount()       // .T. 121 120
? SET(_SET_EOFAPPEND)              // .F.

* REPLACE name with "My Name"      // this will cause RTE 334

SET EOFAPPEND ON                   // anywhere before REPLACE
? SET(_SET_EOFAPPEND)              // .T.
REPLACE name with "My Name"        // this invokes APPEND BLANK
? EOF(), RecNo(), RecCount()       // .F. 121 121
```

Classification:

programming

Compatibility:

New in FS6

Related:

APPEND BLANK, Eof(), Set()

SET EPOCH

Syntax:

```
SET EPOCH TO <expN>
```

Purpose:

Controls the interpretation of dates which have no century digits.

Arguments:

<**expN**> specifies the base year of a 100-year period in which all dates containing only two year-digits are assumed to fall.

Description:

SET EPOCH allows the correct interpretation of date strings containing only two year digits, even for dates outside of the 1900..1999 range. When such a string is converted to a date value, its year digits are compared with the year digits of <expN>. If the year digits in the date are greater than or equal to the year of <expN>, the date is assumed to fall within the same century as given in <expN>; otherwise, the date is assumed to fall in the following century.

The default value for SET EPOCH is 1900, causing dates with no century digits to be interpreted as falling within the 20th century.

Staring with the release 4.42.448, the default EPOCH value is set to **1951** to meet all the "Year 2000 Conformity Requirements", see below. This means, when you need compatibility to Clipper or older FlagShip releases, you should set

```
   SET EPOCH TO 1900
```

at program start.

Year 2000 The BSI committee has specified rules for Y2K conformance (see details and the full text on http://www.fship.com/y2k.html). In short: **FlagShip fully meets** all the requirements. But, since the inference rule (3.2.b) says here: "two-digit years with value > 50 imply 19xx, those with a value <= 50 imply 20xx", the default EPOCH value is now set to 1951. This means, the date entered as e.g. 52 imply the year 1952, whilst the entry 49 imply the year 2049. If you want to enable this rule at the begin of year 2000, use

```
   IF YEAR(DATE()) >= 2000
      SET EPOCH TO 1951
   ENDIF
```

instead. This is valid also for FlagShip which imply an immediate availability of the Y2000 Conformance.

Example:

```
? VERSION(), SET(_SET_EPOCH)             && ...4.42.0448...  1951
SET CENTURY OFF
? DATE(), CTOD("12/31/55")               && 05/20/98   12/31/55
SET CENTURY ON
? DATE(), CTOD("12/31/55")               && 05/20/1998 12/31/1955
? DATE()+730, CTOD("12/31/45")           && 05/19/2000 12/31/2045 !

SET EPOCH TO 2000
? DATE(), CTOD("12/31/55")               && 05/20/1998 12/31/2055
SET EPOCH TO 1990
? DATE(), CTOD("12/31/55")               && 05/20/1998 12/31/2055
SET EPOCH TO 1900
? DATE(), CTOD("12/31/55")               && 05/20/1998 12/31/1955

SET EPOCH TO 1951
? CTOD("01/01/00"),   CTOD("02/29/00")   && 01/01/2000 02/29/2000
? CTOD("01/01/50"),   CTOD("01/01/51")   && 01/01/2050 01/01/1951
? CTOD("01/01/1950"), CTOD("01/01/2051") && 01/01/1950 01/01/2051
```

Classification:

programming

Compatibility:

This command is available in FS4 and C5. FlagShip supports date values from 01/01/0001 to 12/31/9999. Warning: starting with FS4.42.448, the default EPOCH value changed to 1951 and is not equivalent to Clipper's default of 1900. To meet the backward compatibility, use SET EPOCH TO 1900 at program begin.

Translation:

SET (_SET_EPOCH, expN)

Related:

SET CENTURY, SET DATE, CTOD(), DATE(), DTOC(), SET(), VERSION()

SET ESCAPE

Syntax:

`SET ESCAPE ON|off|(<expL>)`

Purpose:

Toggles the possibility of terminating a READ with the Esc key.

Arguments:

ON/OFF enables/disables the ESC as a READ exit key. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

SET ESCAPE OFF causes READ to ignore the Esc key.

By default SET ESCAPE is ON, allowing Esc to abort READ commands discarding the changes and by-passing the validation of RANGE and VALID.

The redirection of the Esc key using SET KEY TO is not affected by SET ESCAPE.

Example:

```
SET ESCAPE OFF
SET FORMAT TO authors
USE author
SET KEY 27 TO Esc_react                 && procedure for ESC key
READ
SET KEY 27 TO
SET ESCAPE ON
PROCEDURE Esc_react (p1,p2,p3)
LOCAL getlist := {}, answer := "N"
@ 24,0                                  && clear line 24
@ 24,0 SAY "Do you really want to terminate input (y/n)?" ;
       GET answer PICTURE "!"
READ
if answer = "Y"
   KEYBOARD chr(3)                      && simulate PgDn key
endif
@ 24,0                                  && clear line 24
RETURN
```

Classification:

programming

Translation:

SET (_SET_ESCAPE, .T. | .F.)

Related:

READ, SET KEY, SETCANCEL(), SET()

SET EVENTMASK

Syntax:

```
SET EVENTMASK [TO] <expN>
```

Purpose:

Set event mask for Inkey().

Arguments:

<**expN**> is a INKEY_* constant specified in the inkey.fh file. Default at start-up is INKEY_ALL_BUT_MOVE

Description:

The SET EVENTMASK command specifies which events should be considered and stored in the ahead buffer to be returned by the INKEY() function. All events not matching the event mask are silently ignored. Using this mask you can have INKEY() return only the events in which you are interested.

Also the Inkey() function has it own, optional event mask. The SET EVENTMASK decides which keys or events are generally considered and stored, whilst the Inkey()s event mask is an additional filter to receive specific, already stored events.

Classification:

programming

Compatibility:

New in FS5

Related:

Inkey(), InkeyTrap()

SET EXACT

Syntax:

```
SET EXACT on|OFF|(<expL>)
```

Purpose:

Toggles the way character strings are compared.

Arguments:

ON/OFF enables/disables exact string comparison regarding the length. Alternatively, the parenthesized <expL> may be used, whereby TRUE sis the same as ON.

Description:

SET EXACT specifies how two strings are to be compared by the relational operators (=, >, <, >=, <=, <>, #, !=). When EXACT OFF is set (the default), the following rules for the comparison of <expC1> ? <expC2> apply:

- When <expC2> is a null string "", the result is always TRUE on =, >=, <= comparison and FALSE otherwise.
- When <expC1> is a null string "", the result is always TRUE on <, <=, <>, #, != comparison and FALSE otherwise.
- If LEN(<expC2>) is greater then LEN(<expC1>), the result is FALSE.
- Otherwise, all characters in <expC2> will be compared to <expC1>. It returns TRUE if all characters are equal or only trailing blanks remain in <expC1>; otherwise FALSE.

Note: when SET EXACT is OFF, the comparison results (intentionally) depends on the operands sequence, empty strings and the trailing blanks in the second operand, which is the xBase standard.

When EXACT is ON, the two strings must match exactly, except for the trailing blanks in <expC1> or <expC2>.

A comparison using the double equal == operator is not affected by SET EXACT and returns TRUE only, if all characters and both lengths are exactly the same.

For a true string equality comparison, use a == b or !(a == b) respectively, since both are independent of the SET EXACT status. Note that the !(a == b) syntax is not the same as a != b and therefore the results may differ. The !=, # and <> operators are fully equivalent.

Example 1:

```
//                          SET EXACT OFF         SET EXACT ON
? "123"   = "12345"   //      .F.                     .F.
? "12345" = "123"     //      .T.                     .F.
? "123"   = ""        //      .T.                     .F.
? ""      = "123"     //      .F.                     .F.
? "123"   = "123"     //      .T.                     .T.
? "123"   = "123  "   //      .F.                     .T.
? "123 "  = "123"     //      .T.                     .T.

? "123"  == "123  "   //      .F.                     .F.
? "123 " == "123"     //      .F.                     .F.
```

Example 2:

Search exact matching, including string length:

```
USE custom INDEX name
IF SeekExact("Smith    ")
   ? custno, name
ELSE
   ? "Customer 'Smith    ' is not available"
ENDIF
RETURN

FUNCTION SeekExact (expC)
SEEK PADR (expC, LEN(&(INDEXKEY(0))))
RETURN (FOUND())
```

Classification:

programming

Compatibility:

In FlagShip (and Clipper 5.x), SET EXACT has no effect on operations other than relational operators. This includes the SEEK and FIND commands. If exact SEEK is required in other dialects, use the example above.

Translation:

SET (_SET_EXACT, . T. | . F.)

Related:

DISPLAY, FIND, LIST, LOCATE, SEEK, SET()

SET EXCLUSIVE

Syntax:

```
SET EXCLUSIVE ON|off|(<expL>)
```

Purpose:

Switches the access status for all subsequently opened databases (.dbf) and their associated memo files (.dbt) and indices (.idx) to EXCLUSIVE (only one user at a time) or SHARABLE (multiuser).

Arguments:

ON/OFF disables/enables the multiuser mode. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

If SET EXCLUSIVE is ON, the newly opened databases (with memo files and indices) will be accessible only from this user or program, until the database is closed again. This switch is identical to the option USE <dbfname> EXCLUSIVE. The SET EXCLUSIVE, however, is a general switch, USE...EXCLUSIVE is associated with the specified dbf only. On the other side, USE...SHARED will open the database in multiuser mode, regardless of the SET EXCLUSIVE status.

If SET EXCLUSIVE is OFF, the newly opened databases (including memo files and indices) will share the access with other users or programs. Using the command USE...EXCLUSIVE will override the SET EXCLUSIVE state and open the database in non-shareable mode.

Multiuser:

The multiuser/multitasking mode is active after SET EXCLUSIVE OFF or the consequent use of USE...SHARED. The **AutoLock** feature is effective only in shared mode.

Before each **write** access in multiuser mode, the record or the whole file must be locked using RLOCK() or FLOCK(). The commands REINDEX, PACK and ZAP require an EXCLUSIVEly opened database. The command INDEX ON requires FLOCK() or EXCLUSIVE usage. If the lock is not set by the programmer and SET AUTOLOCK is ON, FlagShip locks the record or file automatically by using the AUTOxLOCK() function.

Check the open success using the function NETERR() or USED(). Opening a database EXCLUSIVEly will succeed only if it is not already in use by some other user.

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

See also SET COMMIT for tuning the flushing of updated records.

Example:

```
SET EXCLUSIVE OFF                       && set multiuser mode
USE address ALIAS adr                   && open: shareable
DO WHILE NETERR()
   USE address ALIAS adr                && see also USE examples
ENDDO
SET INDEX TO name, idno
```

Classification:

programming

Compatibility:

In FlagShip, the EXCLUSIVE or SHARED mode applies also for the same database concurrently, opened in different working areas, see the USE command. The internal locking mechanism of FlagShip conforms to the Unix/Windows standard. The locking mechanism of nearly all other xBASE derivates is mutually incompatible. The **AutoLock** feature is available in FlagShip only.

Translation:

SET (_SET_EXCLUSIVE, .T. | .F.)

Related:

USE, COMMIT, FLOCK(), RLOCK(), NETERR(), SET(), SET AUTOLOCK, SET COMMIT

SET EXTRA

Syntax 1:

```
SET EXTRA TO [<file>|(<expC>) [ADDITIVE]]
```

Syntax 2:

```
SET EXTRA on|OFF|(<expL>) [NEW]
```

Purpose:

Echoes the console output (e.g. of the ?, ?? commands) to an ASCII text file.

Arguments:

TO <**file**> is the name of an ASCII text file to which the output will be redirected and can include a path and an extension. If the file extension is not specified, .txt is assumed. When the TO... clause is not given, the opened extra file (if any) will be closed.

Option:

ADDITIVE causes the specified extra file to be appended to instead of overwritten. If not specified, the specified <file> is truncated.

Arguments:

ON/OFF activates or deactivates the output to the current open extra file. The toggle will not be switched to ON if the extra file is not opened. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

NEW causes the current file contents to be deleted, instead of appended to.

Description:

FlagShip allows the redirection of console commands (such as ?, LIST, REPORT FORM, LABEL FORM) to four different devices/files at a time: the SCREEN device, and the ALTERNATE, PRINTER and EXTRA files or devices.

In commands, which support the TO FILE <file> clause (like LIST, REPORT FORM etc.), this clause is a synonym for SET EXTRA TO <file> ADDITIVE and SET EXTRA ON. When such a command is finished, the previous EXTRA status is restored.

In other commands (like ?, ??, QOUT() etc.), an additional redirection to a text file (or device) using the SET EXTRA command is possible. Full-screen commands' output such as @...SAY cannot be echoed by the SET EXTRA command; use SET DEVICE instead.

When setting the output OFF, the extra file remains open. Closing the extra file with SET EXTRA TO will reset the toggle to OFF. Only one extra file may be opened at a time (in addition to the alternate and printer file).

Tuning:

You may set the new-line character by 9th element in FS_SET("prset") e.g.
```
   #ifdef FS_WIN32      /* here: should apply for Windows only */
     FS_SET("prset", {NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,chr(13,10) } )
   #endif
```
before printing to EXTRA file via ? or QOUT(). The default setting is line-feed = chr(10).

Example:
```
SET PRINTER   TO all.doc                && or: TO /dev/lp0
SET ALTERNATE TO old.doc                && or: TO /dev/tty15
SET EXTRA     TO new.doc                && or: TO /dev/tty24
SET PRINTER ON
USE address
? "All customers:"
DO WHILE .NOT. EOF() .AND. INKEY() # 27
   IF lastdate < DATE() - 60
      SET ALTERNATE ON
   ENDIF
   IF lastdate >= DATE() - 60
      SET EXTRA ON
   ENDIF
   ? Name, Address, Zip, Town, lastdate
   SET ALTERNATE OFF
   SET EXTRA     OFF
   SKIP
ENDDO
SET PRINTER OFF
? "Old customers (last access older than 2 months):"
TYPE old.doc
WAIT
? "New customers (last access within 2 months):"
TYPE new.doc
WAIT
```

Classification:

programming

Compatibility:

The command is available in FlagShip only, but is compatible with the Clipper 5 behavior.

Translation:

SET (_SET_EXTRA, .on.)
SET (_SET_EXTRA, "file", .additive.)

Related:

?, ??, DISPLAY, LIST, LABEL FORM, REPORT FORM, TEXT, TYPE, QOUT(), QQOUT(), SET ALTERNATE, SET PRINTER, SET()

SET FILTER TO

Syntax:

`SET FILTER TO [<condition>]`

Purpose:

Makes a database appear as if it only contains the records meeting the specified condition.

Arguments:

<**condition**> is a logical expression identifying a specific set of records. SET FILTER TO without an argument deactivates the filter.

Description:

Each working area can have an active filter. When set, a filter becomes active on the first movement of the record pointer in the corresponding working area, e.g. using the GOTO TOP command. The current filtering condition can be returned as a character string using the DBFILTER() function.

Most commands and functions that move the record pointer honor the current filter setting. Filters have no effect on indexing. A filtered record can always be accessed with GOTO, or any command specifying the RECORD scope.

Although SET FILTER makes the current working area appear as if it contains a subset of records, it in fact processes all records in the database sequentially. Therefore, setting FILTER and GOTO TOP needs the same time as the LOCATE command. For a large database, the usage of index, SEEK and subsequent DO WHILE <condition> is the much faster alternative.

Example 1:

```
USE salesmen
SET FILTER TO parts_sold >= 10 .and. parts_sold < 1000
GO TOP                                  // locate first match
? "sales for: " + DBFILTER()
DO WHILE !EOF()
   ? name, parts_sold
   SKIP
ENDDO
```

Example 2:

Same example as above (now in multiuser mode), but much faster on a large dbf

```
IF !FILE("partsold" + INDEXEXT())           // if no index
   USE salesmen EXCLUSIVE NEW               // exists,
   WHILE NETERR()                           // create it
      USE salesmen EXCLUSIVE
   END
   INDEX ON parts_sold TO partsold
   USE
ENDIF
USE salesmen SHARED NEW                     // open database in
WHILE NETERR()                              // multiuser mode
```

```
   USE salesmen SHARED
ENDDO
SET INDEX TO partsold                       // and assign index

// SEEK and "filter" applied records

SET SOFTSEEK ON
SEEK 10                                     // first match +
DO WHILE !EOF() .and. parts_sold < 1000 // filter condition
   ? name, parts_sold
   SKIP
ENDDO
SET SOFTSEEK OFF
```

Classification:

database

Translation:

DBCLEARFILTER ()
DBSETFILTER ({||condition}, "condition")

Related:

SET DELETED, DBFILTER(), LOCATE, SEEK, DBSETFILTER(), oRdd:Filter

SET FIXED

Syntax:

`SET FIXED on|OFF|(<expL>)`

Purpose:

Defines whether the SET DECIMALS will control the display of numeric values, or not.

Arguments:

ON/OFF enables/disables the fixed decimal places display specified by SET DECIMALS. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

After a SET FIXED ON, all numeric values are displayed according to the last SET DECIMALS setting (the default is two decimal digits).

When SET FIXED is OFF, the standard display of numeric values depends on the mathematical operation:

- On assignment (`:=` or `=`), the number of decimal digits of the variable or constant is stored in the receiving variable structure.
- On addition and subtraction (`+`, `-`, `++`, `--`, `+=` or `-=`), the number of decimal places of the operand with the greater number of decimal places is stored.
- On multiplication (`*` or `*=`), the sum of decimal places of both operands is stored.
- On division (`/`, `%`, `/=` or `%=`) and exponentiation (`**`, `^` or `**=`) operations, SET DECIMALS value determines the number of decimal places to display. The same also applies for the functions EXP(), LOG() and SQRT().

SET DECIMALS and SET FIXED only affect the way numbers are displayed (or strings created by STR*(), PAD*(), TRANSFORM() etc.) and have no effect on the precision of numeric calculations.

To display the numbers in another format, use the PICTURE clause of @..SAY or @..GET; the STR() or TRANSFORM() function can be used respectively.

Example:

```
LOCAL num
SET FIXED OFF
SET DECIMALS TO 1
? 1.23456 + 1                           && 2.23456
? 2.2 * 2.2                             && 4.84
? EXP(1)                                && 2.7
? num := 10/3                           && 3.3
SET DECIMALS TO 0
SET FIXED ON
? num                                   && 3
? STR(num, 5, 3)                        && 3.333
? TRANSFORM (num, "9.99999")            && 3.33333
```

Classification:

programming

Translation:

SET (_SET_FIXED, .T. | .F.)

Related:

SET DECIMALS, EXP(), LOG(), SQRT(), @..SAY..PICTURE, TRANSFORM(), STR(), SET()

SET FONT

Syntax:

```
SET FONT [TO] [FACE] <family> [, [<sizePt>]
         [SIZE [<sizePt>]]
         [BOLD][UNDERLINE][UPRIGHT|ITALIC][NORMAL]
```

Purpose:

Sets new default font name and/or size and/or attribute, used for all consecutive console output like QOut(), QQOUT(), @..SAY, @..GET etc. Apply also for printer output with SET GUIPRINT ON. Applicable for GUI mode, ignored otherwise.

Default is oApplic:Font, if not set.

Arguments:

<family> is the used font family. The available family depends on the installed fonts. Usually, at least "Helvetica" or "Arial", "Times" and "Courier" fonts are available.

<sizePt> is the font size in points. The common size is 10 (points), larger size is 12, smaller is 8 points.

UNDERLINE is a underlined face

BOLD is thicker boldface than NORMAL

ITALIC is a cursive face.

UPRIGHT is the usual character face and disables ITALIC

NORMAL disables BOLD, ITALIC and UNDERLINE settings

Description:

In GUI, the default font is set at application begin corresponding to the screen manager setting. You may change the font at any time later.

The default character set in GUI mode (when assigning new font) is ISO-8859-15 which is nearly equivalent to ISO-8859-1 (Latin-1) but contains Euro sign. See http://en.wikipedia.org/wiki/Iso-8859-15 for details.

The fixed font is eg. "Courier" where all characters have the same width and hence the application behaves very similarly to terminal based i/o. The "Helvetica", "Arial" or "Times" are proportional fonts, where each character has different width. It mostly looks more pretty, but the handling is slightly aggravated. FlagShip provides several functions to alleviate the handling with proportional fonts, e.g. StrLen2Col(), StrLen2pix(), SET GUIALIGN etc.

In GUI mode, SET FONT access/assign the m->oApplic:Font object. You therefore may retrieve or set additional font properties by using the Font class, documented in section OBJ. To modify the char set, use SET GUICHARSET or m->oApplic:Font properties.

Note that SET FONT and m->oApplic:Font sets the standard font for displaying @..SAY, @..GET, ?, ?? etc. To change fonts of widgets like Listbox(), Achoice(), Tbrowse() etc, you need to set/modify explicitly the m->oApplic:FontWindow object properties.

The low-level font selection is not performed directly by FlagShip, but is handled by the underlying Qt and X11 or MS-Windows font manager. If the requested font and it characteristics is not found exactly "as is", a heuristic (and sometimes costly) search is used:

- a table of comparable typefaces is searched for similar font family,
- if even the replacement family is not found, "helvetica" or "arial" is searched for,
- if that too is not found, as a last resort a specific font to match to, ignoring the attribute settings, is searched through a built-in list of very common fonts
- if nothing apply, an error message displays.

The following attributes are then matched, in order of priority: character set, fixed/variable pitch, point size, weight, italic. If, for example, a font with the correct character set is found, but with all other attributes in the list unmatched, it will be chosen before a font with the wrong character set but with all other attributes correct. The point size is defined to match if it is within 20% of the requested point size. Of course, when several fonts match and only point size differs the closest point size to the one requested will be chosen.

For additional information about font handling, see also chapters LNG.5.3.1 (fonts), LNG.5.3 (difference between terminal and GUI i/ o), LNG.5.4 (national characters), OBJ.FONT (the font class), SET PIXEL, Col(), Row(), Col2pixel(), Row2pixel(), SET GUITRANSL (using semi-graphic PC8 character set), SET GUIALIGN, SET ROWALIGN, SET ROWADAPT, StrLen2Col(), StrLen2pix()

Note that changing the font will result in recalculating of the current row, column and line height, they all depends on the used font. For using different font attributes within current text, use the FONT clause of ?, ?? or @...SAY commands instead of SET FONT command, see second example below.

Hint: when changing the font, you may need to adapt the application window size to fit max. required rows and columns by invoking

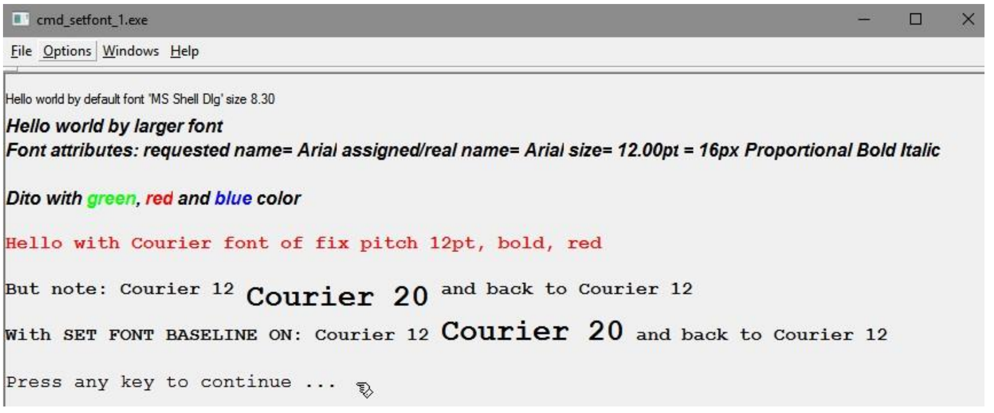
```
   oAplic:Resize(rows, columns, , .T.)
```

to avoid automatic horizontal and/or vertical scroll bars, see the Resize() description in section OBJ.Application class.

Example:

```
? "Hello world by default font '"
?? Set(_SET_FONTNAME)          // e.g. Helvetica
?? "' size "
?? ltrim(Set(_SET_FONTSIZE))   // e.g. 10

SET FONT TO "Arial" SIZE 12 ITALIC BOLD
m->oApplic:Resize(25, 80,, .T.) // resize to 80x25 accord.to font

? "Hello world by larger font"
if AppIoMode() == "G"           // font change apply for GUI mode
  ? "Font attributes: requested name=", m->oApplic:Font:FontName, ;
    "assigned/real name=", m->oApplic:Font:FontFamily, ;
    "size=", ltrim(m->oApplic:Font:Size) + "pt", ;
    "=", ltrim(m->oApplic:Font:SizePixel()) + "px", ;
    FontAttrib(m->oApplic:Font)
endif
?
? "Dito with "
?? "green" GUICOLOR "G+"
?? ", "
?? "red" GUICOLOR "R+"
?? " and "
?? "blue" GUICOLOR "B+"
?? " color"

SET FONT "Courier", 12 BOLD UPRIGHT
?
? "Hello with Courier font of fix pitch 12pt, bold, red" ;
   GUICOLOR "R+"
?
? "But note: Courier 12 "
?? "Courier 20" FONT FontNew("Courier", 20, "B")
?? " and back to Courier 12"
?
SET FONT BASELINE ON
? "With SET FONT BASELINE ON: Courier 12 "
?? "Courier 20" FONT FontNew("Courier", 20, "B")
?? " and back to Courier 12"
SET FONT BASELINE OFF         // reset to default
?
SET FONT "Courier", 12 NORMAL
wait

Function FontAttrib(oFont)
cRet := ""
cRet += if(m->oApplic:Font:Normal,     "NormalWeight ", "")
cRet += if(m->oApplic:Font:Pitch,   "Proportional ", "FixedPitch ")
cRet += if(m->oApplic:Font:Bold,       "Bold "        , "")
cRet += if(m->oApplic:Font:Italic,     "Italic "      , "")
cRet += if(m->oApplic:Font:Underline,  "Underline "   , "")
cRet += if(m->oApplic:Font:StrikeThru, "StrikeThru"   , "")
return cRet
```

Output:



```
Hello world by default font 'MS Shell Dlg' size 8.30
Hello world by larger font
Font attributes: requested name= Arial assigned/real name= Arial size= 12.00pt = 16px Proportional Bold Italic

Dito with green, red and blue color

Hello with Courier font of fix pitch 12pt, bold, red

But note: Courier 12 Courier 20 and back to Courier 12

With SET FONT BASELINE ON: Courier 12 Courier 20 and back to Courier 12

Press any key to continue ...
```

Classification:

screen oriented output in GUI mode and PrintGui() printer output

Compatibility:

New in FS5

Translation:

*_SetDefFont(family, sizePt, lUpright, lItalic, lNormal, ;
lBold, NIL, lUnderline)*

Related:

SetFont(), Set(_SET_FONTNAME | _SET_FONTSIZE | _SET_FONTBOLD | _SET_FONTITALIC), Font class, SET FONT BASELINE, SET GUICHARSET

SET FONT ALIGN
SET FONT BASELINE

Syntax:

`SET FONT [ALIGN] BASELINE on|OFF|(<expL>)`

Purpose:

Enable/disable font alignment to base line. Apply for GUI mode, and for creating printer template via SET PRINTER GUI ON with subsequent printing by PrintGui(), ignored otherwise.

Arguments:

ON/OFF enables/disables the automatic font alignment. Alternatively, the parenthesized <expL> expression may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

The default x/y alignment in GUI mode is on the top left character frame (marked with + in the picture below), to allow start the output at 0,0 coordinates. The characters "O-umlaut","h","p" are displayed as

```
--+----------------------------  ----- <- top character frame
  |  *   *   |       |        |     ^
  |   ###    | #     |        |     |
  |  #   #   | #     |        |     | oFont:Ascent
  | #     #  | ###   | ####   |     |
  |  #   #   | #  #  | #   #  |     |
  |   ###   -| #  # -| ####  -|  ---X- <- base line
  |          |       | #      |     |
  |          |       | #      |     | oFont:Descent
  |          |       | #      |     |
  -----------------------------  ---V- <- bottom character frame
-------------------------------  ----- <- line spacing
```

where the size of (bottom - top) is returned by oFont:Height() - or in pixel by oFont:SizePixel() which corresponds to oFont:Ascend plus oFont:Descend. The line spacing is user definable by global variable `_aGlobSetting[GSET_G_N_ROW_-SPACING]`.

When you change the FONT size, the start position remain unchanged, i.e. larger font has it base line located below the former font (or at higher Y position in view of top/down coordinates):

```
   BBBB           11      BBBBB              2222
   B   B  *      1 1      B    B  *         2    2
   BBBBB  i  ggg   1      B    B      ggggg      2
   B   B  i g  g   1      BBBBB   i  g    g     2
_  BBBB   i  ggg  111 _   B    B  i  g    g   2      __ base line 1
               g          B    B  i  g    g  2
             gg           BBBBB   i   ggggg  222222 __ base line 2
                                              g
                                              g
                                           ggg
```

Sometimes you may wish to align characters on it base line, eg. when using the FONT clause to display different fonts in the same output line (similarly to word processor output), e.g.

```
   SET FONT "Arial",12             // set standard font
   ? "Big1"                        // output by standard font
   SET FONT BASELINE ON
   oFont2 := Font{"Arial",20} ; oFont2:Bold := .T.
   ?? "Big2" FONT oFont2           // output by temporary font
```

where the SET FONT BASELINE ON statement causes the second output to be shifted up so that it base line matches the base line of standard font:

```
                          BBBBB              2222
                          B    B  *         2    2
   BBBB           11      B    B      ggggg      2
   B   B  *      1 1      BBBBB   i  g    g     2
   BBBBB  i  ggg   1      B    B  i  g    g   2
   B   B  i g  g   1      B    B  i  g    g  2
_  BBBB   i  ggg  111 _   BBBBB   i   ggggg  222222  __ base line
               g                              g
             gg                               g
                                           ggg
```

The same apply also for printing with enabled SET PRINTER GUI ON. Note that SET FONT BASELINE takes effect only on the **temporary** font, assigned by the FONT clause of ?, ??, Qout(), QQout() and @...SAY; the output by default font (assigned by SET FONT) remain unchanged. See also example in SET FONT command.

The SET FONT BASELINE does not change ROW() or COL() output, nor the default line spacing. Because of the output shift, this behaves with larger font correctly only if you have enough space above, i.e. when the current ROW() is > 0.

Classification:

screen oriented output in GUI mode and PrintGui() printer output

Compatibility:

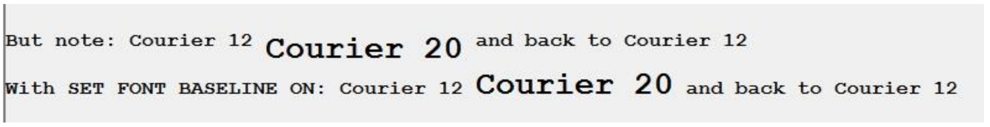
New in FS7

Translation:

Set (_SET_FONT_BASELINE, [.T. | .F.])

Example 1:

See example in SET FONT:

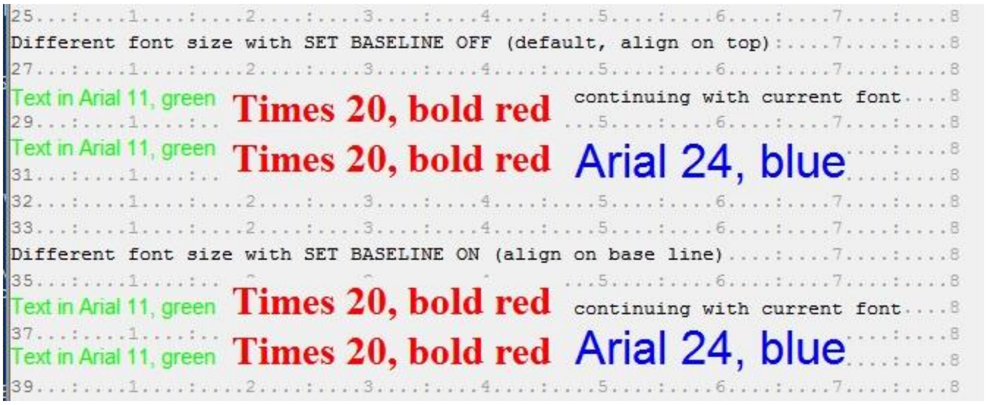


```
But note: Courier 12 Courier 20 and back to Courier 12
With SET FONT BASELINE ON: Courier 12 Courier 20 and back to Courier 12
```

Example 2:

```
See <FlagShip_dir>/examples/printergui.prg
```

Output:



```
25...:....1....:....2....:....3....:....4....:....5....:....6....:....7....:....8
Different font size with SET BASELINE OFF (default, align on top):....7....:....8
27...:....1....:....2....:....3....:....4....:....5....:....6....:....7....:....8
Text in Arial 11, green Times 20, bold red continuing with current font....8
29...:....1....:..                          ...5....:....6....:....7....:....8
Text in Arial 11, green Times 20, bold red Arial 24, blue..........:....8
31...:....1....:..                                        ..........:....8
32...:....1....:....2....:....3....:....4....:....5....:....6....:....7....:....8
33...:....1....:....2....:....3....:....4....:....5....:....6....:....7....:....8
Different font size with SET BASELINE ON (align on base line)....:....7....:....8
35...:....1....:..                          ...5....:....6....:....7....:....8
Text in Arial 11, green Times 20, bold red continuing with current font....8
37...:....1....:..                                        ..........:....8
Text in Arial 11, green Times 20, bold red Arial 24, blue..........:....8
39...:....1....:....2....:....3....:....4....:....5....:....6....:....7....:....8
```

Related:

SET FONT TO, SetFont(), ?, ??, @..SAY, Qout(), Qqout(), Font class, SET PRINTER, PrintGui()

SET FORMAT TO

Syntax:

```
SET FORMAT TO <procname>
```

Purpose:

Specifies a format procedure to be executed before every READ command.

Arguments:

<**procname**> can be a user-defined procedure (UDP) or a file with the .fmt or .prg extension. If <procname> is not specified, the current FORMAT is deactivated.

Description:

The only difference between format procedures and other procedures is the way they are invoked. The format procedures are executed when a READ is encountered after a SET FORMAT.

SET FORMAT is a global setting, which means that there can only be one active format at a time. An other SET FORMAT statement in the format procedure will become active when the current format procedure terminates and will be executed by subsequent READs.

In the FORMAT procedure, all FlagShip commands and functions in addition to @...SAY and @...GET, can be used.

Unlike the interpreted xBASE dialects, format files are not opened at runtime but compiled and linked into the application. When the FlagShip compiler encounters a SET FORMAT command and the name of the procedure is unknown, it searches the current directory for a source file with the same name (and the .frm or .prg extension) in order to compile it. If not found, the object file has to be specified at link time; otherwise an error "unresolved external _bb_<procname>" occurs.

Therefore, the name of the format procedure must be unique in the whole application. It must differ from all the function, procedure and other format names, as well as from all the file names comprising the application when their extension is discarded. This means that, for example "test.fmt" and "test.prg" may not be parts of the same application.

Note that SET FORMAT TO is an obsolete command and is supported for compatibility purposes only.

Example:

```
SET FORMAT TO get_name
USE authors
DO WHILE .NOT. EOF()
   READ
   SKIP
ENDDO
USE
SET FORMAT TO test                      // compiles test.frm
READ
RETURN

PROCEDURE get_name
@ 10,0  CLEAR TO 10,79
@ 10,10 SAY "First name: " GET first_name
@ 10,50 SAY "Last  name: " GET last_name
RETURN
```

Classification:

programming, compiler

Compatibility:

Unlike the interpreted xBASE dialects, the screen is not cleared before executing the format procedure. Multiple-page formats are not supported. Note also the compiler notes above.

Translation:

PROCREQ ("procname") ; __SETFORMAT ({|| procname() })

Related:

@...SAY, @...GET, READ, PROCEDURE, DO...WITH

SET FUNCTION ... TO

Syntax:

```
SET FUNCTION <expN1> TO <expC2>
```

Purpose:

Defines a string that will be pushed into the keyboard buffer when the specified function key is pressed.

Arguments:

<**expN1**> is the function key number (1..48), e.g. 8 for the F8 key.

<**expC2**> is the character string to assign to the function key. If <expC2> is not specified, the current string assignment to a FN key is disabled.

Description:

When the specified function key is pressed, the keyboard buffer is stuffed with the character string which may contain any characters including control characters. The following keys can be assigned with SET FUNCTION (see also section SYS and the FStinfo.src file):

expN1	Function	Key			terminfo
1 - 10		F1 - F10			kf1...kf10
11 - 20	shift	F1 - F10			kf13...kf22
21 - 30	ctrl	F1 - F10			kf25...kf34
31 - 40	alt	F1 - F10	*		kf37...kf46
41 - 42		F11 - F12		**	kf11...kf12
43 - 44	shift	F11 - F12		**	kf23...kf24
45 - 46	ctrl	F11 - F12		**	kf35...kf36
47 - 48	alt	F11 - F12	*	**	kf47...kf48

* The "Alt-FN" keys are seldom available on the Unix terminals and systems, but are often supported by Ctrl+Shift+FN, see terminfo (e.g. FStinfo.src).

** The F11 and F12 key combinations are not supported by all of the DOS derivates. On Unix, the usage is dependent on the terminal capability (see FStinfo.src).

A key redirection to a UDF using SET KEY has precedence over SET FUNCTION. Initially when a program is started, the F1 key is redirected to the HELP procedure, if any. To SET FUNCTION for any key that has been redirected with SET KEY, first the SET KEY redirection must be disabled prior to the SET FUNCTION setting.

To determine the current FUNCTION setting of the specified FN key, the function expC := __GETFUNCTION (<expN1>) can be used; see also getsys.prg.

Example:

```
Each time F9 is pressed, the cursor jumps 4 GETs ahead

four_gets = REPLICATE(CHR(13),4)
SET FUNCTION 9 TO four_gets
SET FORMAT TO Articles
USE Article
READ
SET KEY F9 TO
SET FORMAT TO
USE
RETURN
```

Classification:

programming

Compatibility

Unlike C5, SET KEY does not disable the current SET FUNCTION setting, but hides it only, similar to C87.

The ability of function keys depends on the current setting of the environment variable TERM, the respective terminfo description and the hardware capability. Refer to sections SYS and REF for available function keys according to your terminal.

Translation:

```
__SETFUNCTION (expN1, "expC2")
```

Related:

SET KEY, KEYBOARD, <FlagShip_dir>/system/getsys.prg

SET GOTOP

Syntax:

```
SET GOTOP on|OFF|(<expL>)
```

Purpose:

Enable/disable automatic movement to database top

Arguments:

ON/OFF enables/disables automatic database movement to the first logical record after USE... or USE..INDEX.. or SET INDEX.. command. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default is OFF which enables programmable index integrity check and it silent recovery by using INDEXCHECK() function.

Description:

FlagShip automatically checks the database and index integrity, see chapter LNG.4.5. However, this index integrity checking disables the automatic movement to the database top, so when you are using SET FILTER or SET DELETED ON, you may need to issue GO TOP or DbGoTop() after open the database or assigning new indices. You may force the GO TOP movement automatically by SET GOTOP ON or Set(_SET_GOTOP,.T.) which will then handle same as Clipper, but disables the possibility of index check and it silent recovery.

Note: the SET GOTOP is considered only with the USE.. and SET INDEX commands. Hence if you are using the DbUseArea() or OrdListAdd() functions instead of the commands, you will need to invoke DbGoTop() function (or GO TOP command) thereafter to move to the first logical database record.

Classification:

programming

Compatibility:

New in FS5

Related:

USE, SET INDEX, INDEXCHECK(), SET(_SET_GOTOP), GO TOP, DbGoTop()

SET GUIALIGN

Syntax:

`SET GUIALIGN ON|off|(<expL>)`

Purpose:

Align all @..GET columns in a GetList{} array according to the length of ..SAY.. text during a READ or via GetAlign([GetList]) call. Apply only in GUI when @..SAY..GET is specified.

Arguments:

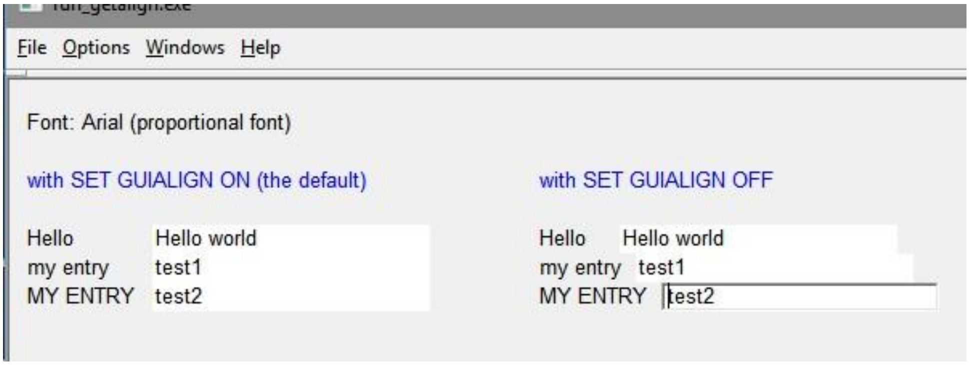
ON/OFF enables/disables the aligning. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is ON.

Description:

In terminal based i/o or in GUI with fixed fonts, these lines

```
   @ 10,1 say "Hello    " GET var1
   @ 12,1 say "my entry" GET var2
   @ 13,1 say "MY ENTRY" GET var3
   READ
```

produces get/read fields all aligned at column 10. In GUI with proportional fonts, the column of these GET fields will vary, since every of this commands says "*display the @..SAY text in the specified row/column and start the GET input fields one character behind the text end*" which is done correctly. But it does not look as you wanted, since the width of these texts is different, where you want to get all the fields among one another. To do so, you may advise the READ (which has the information about the final layout from the objects in GetList array) by SET GUI ALIGN ON to reformat/align these fields at the same column.



Font: Arial (proportional font)

with SET GUIALIGN ON (the default)

```
Hello        Hello world
my entry     test1
MY ENTRY     test2
```

with SET GUIALIGN OFF

```
Hello     Hello world
my entry  test1
MY ENTRY  test2
```

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: The GetAlign() function is available in <FlagShip_dir>/system/getsys.prg

Related: GetAlign(), Set(_SET_GUIALIGN), READ, @..SAY..GET

SET GUICHARSET

Syntax:

```
SET GUICHARSET BOLD|UNDERLINE|ITALIC|STRIKETHRU
          on|OFF|(<expL>)
SET GUICHARSET FONT_ISO8859_1 ... FONT_UNICODE
```

Purpose:

Modifies the current application font (oApplic:Font) in GUI mode.

Arguments:

SET GUICHARSET BOLD on|OFF|(<expL>) sets or clears the bold attribute, equivalent to m->oApplic:Font:Bold := <expL>

SET GUICHARSET UNDERLINE on|OFF|(<expL>) sets or clears the underline attribute, equivalent to m->oApplic:Font:Underline := <expL>

SET GUICHARSET ITALIC on|OFF|(<expL>) sets or clears the italic attribute, equivalent to m->oApplic:Font:Underline := <expL>

SET GUICHARSET STRIKETHRU on|OFF|(<expL>) sets or clears the strike thru attribute, equivalent to m->oApplic:Font:StrikeThru := <expL>

SET GUICHARSET FONT_ISO8859_1 ... FONT_UNICODE sets the requested character set (codec) for current font, equivalent to m->oApplic: Font:CharSet(...). Valid arguments are:

```
FONT_ISO8859_1    = Latin-1, common in much of Europe
FONT_ISO8859_2    = Latin-2, Central and Eastern European
FONT_ISO8859_3    = Latin-3, South European
FONT_ISO8859_4    = Latin-4, North European
FONT_ISO8859_5    = Latin/Cyrillic
FONT_ISO8859_6    = Latin/Arabic
FONT_ISO8859_7    = Latin/Greek with Euro sign
FONT_ISO8859_8    = Latin/Hebrew
FONT_ISO8859_9    = Latin-5, Turkish
FONT_ISO8859_10   = Latin-6, Nordic
FONT_ISO8859_11   = Latin/Thai alphabet
FONT_ISO8859_12   = Latin/Devanagari
FONT_ISO8859_13   = Latin-7, Baltic Rim
FONT_ISO8859_14   = Latin-8, Celtic
FONT_ISO8859_15   = Latin-9, same as Latin-1 but with Euro
FONT_ISO8859_16   = Latin-10, South-Eastern European
FONT_KOI8R        = KOI8-R, Cyrillic - RFC 1489
FONT_KOI8U        = KOI8-U, Cyrillic/Ukrainian - RFC 2319
FONT_SET_JA       = font specific: Japanese
FONT_SET_KO       = font specific: Korean
FONT_SET_TH_TH    = font specific: Thai
FONT_SET_ZH       = font specific: Chinese
FONT_SET_ZH_TW    = font specific: traditional Chinese
FONT_SET_BIG5     = font specific: Chinese
FONT_GBK          = font specific: simplified Chinese
FONT_CP1251       = Microsoft Cyrillic encoding
```

```
FONT_PT154        = Paratype Asian Cyrillic encoding
FONT_UNICODE      = Unicode ISO-10646 (UTF-8 encoding)
FONT_UTF8         = same as FONT_UNICODE
```

The default character set in GUI mode (when assigning new font) is FONT_ISO8859_15 (ISO-8859-15, Latin-9) which is nearly equivalent to FONT_ISO8859_1 (ISO-8859-1, Latin-1) but contains also Euro sign, see details in *http://en.wikipedia.org/wiki/Iso-8859-15*. This default setting can be changed by assigning

```
_aGlobSetting[GSET_G_C_FONTCHARSET] := "FONT_ISO8859_15" // def
```

You may retrieve (or set) the attribute of current font by m->oApplic:Font:Attrib() or m->oApplic:Font:AttribChar() and the charset by m->oApplic:Font:CharSet() or oApplic:Font:CharSetName(), see section OBJ.Font

Description:

In GUI, the default font is set at application begin corresponding to the screen manager setting. You may change the font at any time later SET FONT, SET GUICHARSET, and m->oApplic:Font properties.

Note that SET FONT and m->oApplic:Font sets the standard font for displaying @..SAY, @..GET, ?, ?? etc. The SET GUICHARSET modifies this standard font. To change fonts of widgets like Listbox(), Achoice(), Tbrowse() etc, you need to set/modify explicitly the m->oApplic:FontWindow object properties.

The Unicode is available in GUI mode only and uses UTF-8 encoding. To transfer ASCII PC-8 string to Unicode, use cp437_utf8() function. To convert UTF-16 to UTF-8, use Utf16_Utf8() function. In Linux, you may need to set corresponding Unicode font, e.g. SET FONT "mincho" for Japanese.

Classification:

screen oriented output in GUI mode and PrintGui() printer output

Compatibility:

New in FS7

Translation:

_SetGuiCharset(cMode, [clOnOff]) -> lOk

Related:

SET FONT, SetFont(), Font class, Cp437_utf8()

SET GUICOLORS

Syntax:

`SET GUICOLORS on|OFF|(<expL>)`

Purpose:

Enable colors also in GUI application

Arguments:

ON/OFF enables/disables color support in GUI mode. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

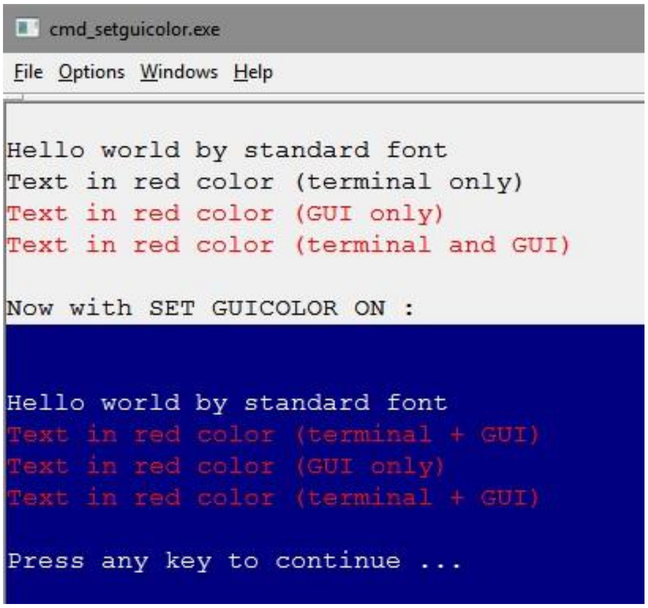
Colors and lines drawing are disabled per default in GUI mode to get proper GUI look & feel. You may enable the color support in GUI mode via SET GUICOLOR ON or `Set(_SET_GUICOLORS,.T.)` to consider the `SET COLOR` values, same as in terminal i/o mode. Most of the display commands supports also GUICOLOR clause for GUI mode, which then temporarily overrides the default or SET COLOR setting.

The SET GUICOLOR influences the ?, ??, qout(), qqout(), @..SAY, @..GET, @..DRAW etc. console output.

Example:

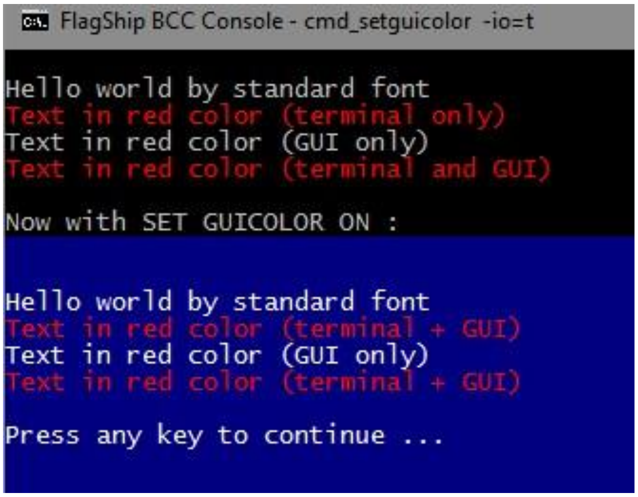
```
SET FONT "Courier", 12       // set default font (GUI only)
oApplic:Resize(25,80,,.T.)   // resize according to font (GUI only)

? "Hello world by standard font"
? "Text in red color (terminal only)" COLOR "R+"
? "Text in red color (GUI only)"      GUICOLOR "R+"
? "Text in red color (terminal and GUI)" COLOR "R+" GUICOLOR "R+"
?
? "Now with SET GUICOLOR ON :"
SET COLOR TO "W+/B,N/W"
SET GUICOLOR ON                      // accept COLOR also in GUI
@ row() +1,0 CLEAR
?
? "Hello world by standard font"
? "Text in red color (terminal + GUI)" COLOR "R+/B"
? "Text in red color (GUI only)"       GUICOLOR "R+/B"
? "Text in red color (terminal + GUI)" COLOR "R+/B" GUICOLOR "R+/B"
?
wait
```

Output in GUI mode:



```
Hello world by standard font
Text in red color (terminal only)
Text in red color (GUI only)
Text in red color (terminal and GUI)

Now with SET GUICOLOR ON :


Hello world by standard font
Text in red color (terminal + GUI)
Text in red color (GUI only)
Text in red color (terminal + GUI)

Press any key to continue ...
```

Output in terminal i/o:



```
Hello world by standard font
Text in red color (terminal only)
Text in red color (GUI only)
Text in red color (terminal and GUI)

Now with SET GUICOLOR ON :


Hello world by standard font
Text in red color (terminal + GUI)
Text in red color (GUI only)
Text in red color (terminal + GUI)

Press any key to continue ...
```

Classification:

programming

Compatibility:

New in FS5

Related:

SET COLOR, Set(_SET_GUICOLORS)

SET GUICURSOR

Syntax 1:

```
SET GUICURSOR on|OFF|(<expL>)
```

Syntax 2:

```
SET GUICURSOR TO <expN>
```

Syntax 3:

```
SET GUICURSOR TO
```

Purpose:

Enable/disable text cursor in GUI mode or specify the GUI cursor shape.

Arguments:

ON/OFF enables/disables the display of text cursor in GUI mode. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF. The corresponding function is SET(_SET_GUICURSOR, [<expL>]).

TO <expN> according to syntax 2 re-defines the default shape CURSOR_UNDERSCORE, same as invoking SET(_SET_GUICURSORTYPE, val). The new shape is then displayed on subsequent screen output when SET GUICURSOR is ON. Valid shape values are:

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_UNDERSCORE	-21	underscore
CURSOR_BOX	-22	box in size of one character
CURSOR_DEFAULT_TEXT	-21	default = CURSOR_UNDERSCORE

TO according to syntax 3 sets the text cursor shape to it default state, i.e. to CURSOR_UNDERSCORE or the value assigned to the global variable _aGlobSetting[GSET_G_N_TEXTSHAPE].

Description:

The behavior of an application in GUI mode with SET GUICURSOR ON is very similar to running it in textual mode. If set, the default (or user set) cursor shape is displayed behind the current screen output, i.e. at the Row(), Col() position, independent on the mouse cursor.

The SET GUICURSOR is considered in ?, ??, qout(), qqout(), @..SAY, SetPos() etc. console output. The text cursor is not displayed in special add-on widgets (controls) like READ, MemoEdit(), Tbrowse(), InfoBox() etc.

If you want to set text cursor shape anywhere on the user screen, independent on the current output, use SetGuiCursor() and best to disable SET GUICURSOR using the OFF clause.

Note: The WAIT command use own shape to signal user's entry. If the SET GUICURSOR display is enabled, your cursor shape will be restored automatically at the return from WAIT.

To set the shape of mouse cursor, use MsetCursor().

The SET GUICURSOR command is accepted also for other than GUI i/o modes, but no action is taken there.

Example:

complete example is available in .../examples/guicursor.prg

Classification:

screen oriented output in GUI mode

Translation:

```
SET(_SET_GUICURSOR     [, <expL> ])
SET(_SET_GUICURSORTYPE [, <expN> ])
```

Compatibility:

New in FS5

Related:

SET CURSOR, SetGuiCursor(), SetPos(), MsetCursor(), WAIT

SET GUIPRINTER

Syntax:

```
SET GUIPRINTER on|OFF|(<expL>)
```

Purpose:

Enable or disable GUI alike printing via selected system driver. Applicable only in GUI mode, ignored otherwise.

Arguments:

ON/OFF activates or deactivates the rendering for printer output.

Description:

With enabled SET GUIPRINTER, the output is (additionally) rendered for selected printer in GUI mode, and subsequently printed by `PrintGui()` or by _oPrinter:ExecGui().

SET GUIPRINTER ON is equivalent to PrintGui(.T.), SET GUIPRINTER OFF is equivalent to PrintGui(.F.)

For further details and example, see function **PrintGui()**

Classification:

programming

Compatibility:

New in FS7

Translation:

SET (_SET_GUIPRINTER, .T. | .F.)

Related:

PrintGui(), SET PRINTER, SET CONSOLE, SET DEVICE, SET GUI*

SET GUITRANSL

Syntax:

```
SET GUITRANSL ASCII    on|OFF|(<expL>)
SET GUITRANSL TEXTDRAW on|OFF|(<expL>)
SET GUITRANSL BOX      on|OFF|(<expL>)
SET GUITRANSL LINES    on|OFF|(<expL>)
SET GUITRANSL LOWCP437 on|OFF|(<expL>)
```

Purpose:

Enable support of semi-graphic characters also in GUI application and/or automatically translate ANSI to ISO code

Arguments:

ON/OFF enables/disables support of semi-graphic characters in GUI mode. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

In GUI mode, colors, boxes and semi-graphic characters are handled upon programmer's request only, since if would (in the most cases) break the look-and-feel of GUI, and the source cross-compatibility to terminal i/o mode.

Character set conversion, national and semi-graphic characters

In GUI mode, the screen-output, and the output to GDI printer (via SET GUIPRINT ON) is in ISO/ANSI mode (internally in Unicode). This ISO/ANSI character set have no semi-graphics, and the byte representation of national characters (i.e. CHR(128..255)) differs to PC8/ASCII/OEM charset, refer to LNG.5.4 for differences and to the ASCII-ISO comparison table in <FlagShip_dir>/manual/charset.pdf file.

The consequence is, that strings with national characters coded in editor supporting ASCII/OEM/PC8 charset differs from strings coded in ISO/ANSI alike editor. For example, the output of ? "München" may or may not be displayed properly, since your code contains different byte-representation of the u-umlaut. Or, the CHR(196) is horiz.line in ASCII/OEM, but A-umlaut in ISO/ANSI mode.

FlagShip however provides automatic conversion between these codes by using SET SOURCE ASCII (default) or SET SOURCE ISO, see details there. For GUI mode, you may additionally/differently control this translation by **SET GUITRANSL ASCII** ON or OFF.

For text coded in PC8/ASCII/OEM character set (assumed by default), an automatic ASCII -> ISO conversion is available via SET SOURCE ASCII and/or SET GUITRANSL ASCII ON. This setting converts automatically ASCII strings passed to i/o commands and functions to ISO character set, same as doing it manually via Oem2Ansi(string). If you wish to draw semi-graphic characters passed in ASCII mode, use additionally **SET GUITRANSL TEXTDRAW ON**.

For text passed in ISO/ANSI mode, SET GUITRANSL ASCII OFF should be used. In this mode, you cannot display semi-graphic characters (by SET GUITRANSL TEXT ON) simultaneously with umlauts (or other special characters), since e.g. the CHR(196) = horizontal line in ASCII is equivalent to A-umlaut in ISO/ANSI mode (see the comparison table). You however may draw semi-graphics by CHR(..) as well, see example below.

The SET GUITRANSL ASCII ON is equivalent to SET CHARSET ASCII, and SET GUITRANSL ASCII OFF is equivalent to SET CHARSET ISO.

The SET GUITRANSL ASCII ON is similar to SET SOURCE ASCII, but the seconds translates also output for terminal i/o and std. printer, whilst SET GUITRANSL affects screen (and SET GUIPRINT) translation only. Equivalently, SET GUITRANSL ASCII OFF is similar to SET SOURCE ISO, but w/o terminal and std.printer influence.

To draw semi-graphic ASCII characters 1..31 in GUI mode, use **SET GUITRANSL LOWCP437 ON** or Set(_SET_GUILOWCP437,.T.) for an automatic conversion for @..SAY CHR(n). Not considered with with ?, ?? etc. commands, but you may display/print it according to example 2 below.

To draw semi-graphic ASCII characters 179..218 in GUI mode, use **SET GUITRANSL TEXT ON** or Set(_SET_GUIDRAWTEXT,.T.) for an automatic translation of text strings to graphic ASCII characters - but not chr(176,177,178,219..223). If both GUITRANSL TEXT and GUITRANSL ASCII are ON, these 179..218 semi-graphic chars are not translated to the ISO equivalence, but drawn as graphic in the ?, ??, @..SAY commands and Qout(), Qqout() functions.

When using Unicode for input/output (see LNG.5.4.5), best to avoid SET GUITRANSL TEXT ON and SET GUITRANSL ASCII ON, since bytes 127.. 255 may conflict with UTF8 encoding for glyphs. See example 2 below

SET GUITRANSL ASCII ON	translates ASCII source to ISO, this is set also by SET SOURCE ASCII or by SET CHARSET ASCII
SET GUITRANSL ASCII OFF	(default) the source is in ISO charset, this is set also by SET SOURCE ISO or by SET CHARSET ISO
SET GUITRANSL TEXTDRAW ON	draws semi-graphic passed in ASCII code, when SET GUITRANSL ASCII is ON
SET GUITRANSL TEXTDRAW OFF	(default) disables semi-graphic drawing to be able print national characters chr(128..255) passed in ISO mode and to ensure GUI look & feel
SET GUITRANSL LOWCP437 ON	draws chr(1..31) by CP437 character set when using @..SAY, but not with ?, ??

Boxes, lines

Boxes and lines drawing are disabled per default in GUI mode, since standard widgets/controls usually have own frames. To enable drawing lines and boxes via @..TO.. and @..BOX in GUI mode too, use SET GUITRANSL LINES ON and/or SET GUITRANSL BOX ON, or the corresponding Set(_SET_GUIDRAWLINE, .T.) and Set(_SET_GUIDRAWBOX, .T.) function. You may draw lines also directly by the @..DRAW.. command.

Colors:

Colors are disabled per default in GUI mode to get proper GUI look & feel. You may enable the color support in GUI mode via SET GUICOLOR ON or Set(_SET_GUICOLORS, .T.). This will use the global SET COLOR setting, or the explicit COLOR clause of many commands. Alternatively, you may use the GUICOLOR command clause for specific color setting, also without SET GUICOLOR ON.

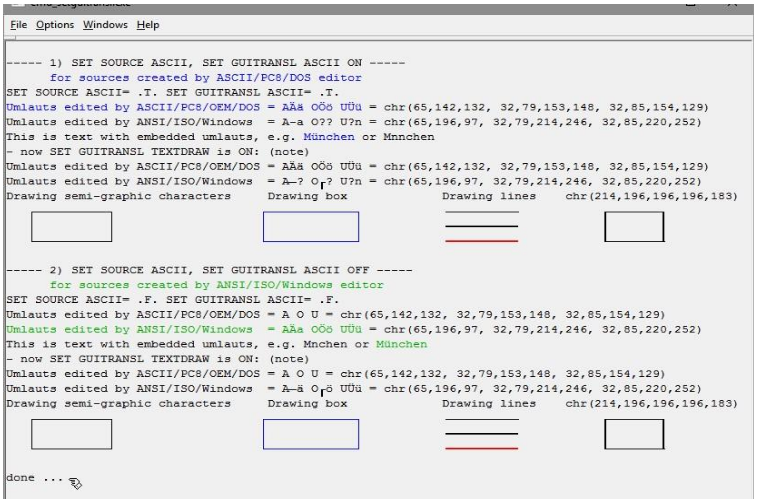
All these commands and functions may remain global in the source code for a hybrid executables, they will be ignored when the application run in Terminal or Basic mode.

Classification:

programming

Example 1:

```
set font "courier", 10
oApplic:resize(35,100,,.T.)
cUmlautPC8  := "AÄä OÖö UÜü" // chr(65,142,132, 32,79,153,148, ;
                             //      32,85,154,129)
cUmlautANSI := "A─a OÍ÷  U▄▄³ " // chr(65,196, 97, 32,79,214,246, ;
                             //      32,85,220,252)
cColorN := "N"
cColor1 := "B+"
cColor2 := "#00AA00"

for ii := 1 to 2
   * ii=1: Text is created with PC8/ASCII/OEM/DOS program editor
   * ii=2: Text is created with ANSI/ISO/Windows  program editor
   *      (See full source in <FlagShip_dir>/examples/setsource.prg)
   if ii == 1
      ? "----- 1) SET SOURCE ASCII, SET GUITRANSL ASCII ON -----"
      ? "       for sources created by ASCII/PC8/DOS editor" ;
        GUICOLOR cColor1
      SET SOURCE ASCII  // implies SET GUITRANSL ASCII ON
   else
      ? "----- 2) SET SOURCE ASCII, SET GUITRANSL ASCII OFF -----"
      ? "       for sources created by ANSI/ISO/Windows editor" ;
        GUICOLOR cColor2
      SET SOURCE ISO    // implies SET GUITRANSL ASCII OFF
   endif
   ? "SET SOURCE ASCII=",set(_SET_SOURCEASCII), ;
     "SET GUITRANSL ASCII=",set(_SET_GUIASCII)
   ? "Umlauts edited by ASCII/PC8/OEM/DOS = AÄä OÖö UÜü "  ;
     GUICOLOR (if(ii==1, cColor1, cColorN))
   ?? "= chr(65,142,132, 32,79,153,148, 32,85,154,129)"
   ? "Umlauts edited by ANSI/ISO/Windows  = A─a OÍ÷  U▄▄³  " ;
     GUICOLOR (if(ii==2, cColor2, cColorN))
   ?? "= chr(65,196, 97, 32,79,214,246, 32,85,220,252)"
   ? "This is text with embedded umlauts, e.g. "
   ?? "München" GUICOLOR (if(ii==1, cColor1, cColorN))
   ?? " or "
   ?? "M³nchen" GUICOLOR (if(ii==2, cColor2, cColorN))
   SET GUITRANSL TEXT ON   // To draw semi-graphic characters
   ? "- now SET GUITRANSL TEXTDRAW is ON: (note)"
   ? "Umlauts edited by ASCII/PC8/OEM/DOS = AÄä OÖö UÜü " + ;
     "= chr(65,142,132, 32,79,153,148, 32,85,154,129)"
   ? "Umlauts edited by ANSI/ISO/Windows  = A─õ OÍ÷ U▄³  " + ;
     "= chr(65,196, 97, 32,79,214,246, 32,85,220,252)"
   ? "Drawing semi-graphic characters"
   rr := row()
   // saveFont := set(_SET_FONTNAME, "courier") // fixed font
   ? space(3) + chr(218) + repli(chr(196),10) + chr(191)
   ? space(3) + chr(179) + space(10) + chr(179)
   ? space(3) + chr(192) + repli(chr(196),10) + chr(217)
   // set(_SET_FONTNAME, saveFont)              // restore font
   SET GUITRANSL TEXT OFF   // To draw semi-graphic characters

   @ rr, 36 say "Drawing box"
   SET GUITRANSL BOX ON                         // for GUI mode
   @ rr+1, 35, rr+3, 48 box color "B+" guicolor "B+"
   SET GUITRANSL BOX OFF                        // for GUI mode
   @ rr, 60 say "Drawing lines"
   SET GUITRANSL LINES ON                       // for GUI mode
   @ rr+1, 60 to rr+1, 70
   @ rr+2, 60 to rr+2, 70 double
   @ rr+3, 60 to rr+3, 70 double color "R+" guicolor "R+"
   SET GUITRANSL TEXT ON                        // for GUI mode
   @ rr, 77 say "chr(214,196,196,196,183)"
   @ rr+1,82 say chr(214,196,196,196,196,196,196,196,183)
   @ rr+2,82 say chr(186, 32, 32, 32, 32, 32, 32, 32,186)
   @ rr+3,82 say chr(211,196,196,196,196,196,196,196,189)
   setpos(rr+4,0)
   SET GUITRANSL LINES OFF ; SET GUITRANSL TEXT OFF // for GUI mode
next
wait "done ..."
```

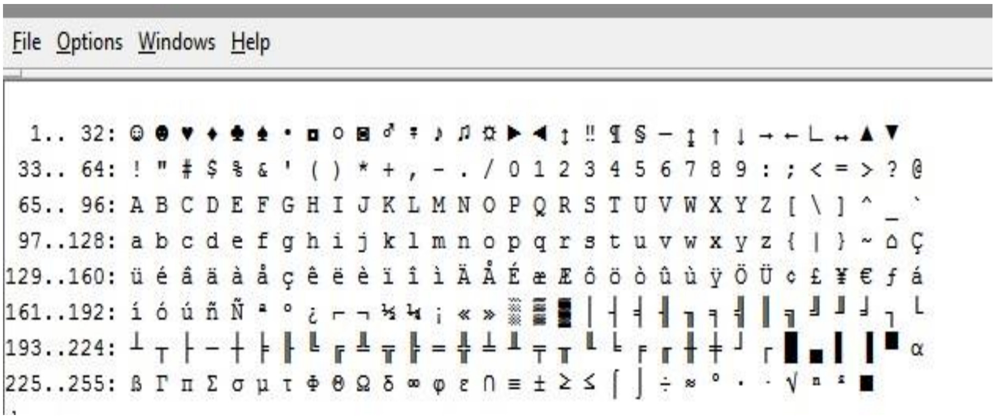


```
----- 1) SET SOURCE ASCII, SET GUITRANSL ASCII ON -----
      for sources created by ASCII/PC8/DOS editor
SET SOURCE ASCII= .T. SET GUITRANSL ASCII= .T.
Umlauts edited by ASCII/PC8/OEM/DOS = AÄä OÖö UÜü = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = A-a O?? U?n = chr(65,196,97, 32,79,214,246, 32,85,220,252)
This is text with embedded umlauts, e.g. München or Mnnchen
- now SET GUITRANSL TEXTDRAW is ON: (note)
Umlauts edited by ASCII/PC8/OEM/DOS = AÄä OÖö UÜü = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = A─? O┌? U?n = chr(65,196,97, 32,79,214,246, 32,85,220,252)
Drawing semi-graphic characters     Drawing box          Drawing lines    chr(214,196,196,196,183)

----- 2) SET SOURCE ASCII, SET GUITRANSL ASCII OFF -----
      for sources created by ANSI/ISO/Windows editor
SET SOURCE ASCII= .F. SET GUITRANSL ASCII= .F.
Umlauts edited by ASCII/PC8/OEM/DOS = A O U = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = AÄa OÖö UÜü = chr(65,196,97, 32,79,214,246, 32,85,220,252)
This is text with embedded umlauts, e.g. Mnchen or München
- now SET GUITRANSL TEXTDRAW is ON: (note)
Umlauts edited by ASCII/PC8/OEM/DOS = A O U = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = A─ä O┌ö UÜü = chr(65,196,97, 32,79,214,246, 32,85,220,252)
Drawing semi-graphic characters     Drawing box          Drawing lines    chr(214,196,196,196,183)

done ...
```

Example 2:

```
* Display full CP437 character set in GUI mode via Unicode

SET FONT "Courier New",10
oFontUni := Font{}
ObjClone(m->oApplic:Font, oFontUni)
oFontUni:CharSet("FONT_UNICODE")

for ii := 1 to 255
   if ii % 32 == 1
      ? str(ii,3)+".."+str(min(ii+31,255),3)+": "
      rr := row()
      cc := col()
   endif
   @ rr,cc SAY cp437_utf8(chr(ii)) FONT oFontUni
*  setpos(rr,cc)                          // alternatively
*  ?? cp437_utf8(chr(ii)) FONT oFontUni  // instd. of @..SAY
   cc += 2
next
wait
```

Output:



```
  1.. 32: ☺ ☻ ♥ ♦ ♣ ♠ • ◘ ○ ◙ ♂ ♀ ♪ ♫ ☼ ► ◄ ↕ ‼ ¶ § ▬ ↨ ↑ ↓ → ← ∟ ↔ ▲ ▼
 33.. 64: ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
 65.. 96: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _ `
 97..128: a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ ⌂ Ç
129..160: ü é â ä à å ç ê ë è ï î ì Ä Å É æ Æ ô ö ò û ù ÿ Ö Ü ¢ £ ¥ € ƒ á
161..192: í ó ú ñ Ñ ª º ¿ ⌐ ¬ ½ ¼ ¡ « » ░ ▒ ▓ │ ┤ ╡ ╢ ╖ ╕ ╣ ║ ╗ ╝ ╜ ╛ ┐ └
193..224: ┴ ┬ ├ ─ ┼ ╞ ╟ ╚ ╔ ╩ ╦ ╠ ═ ╬ ╧ ╨ ╤ ╥ ╙ ╘ ╒ ╓ ╫ ╪ ┘ ┌ █ ▄ ▌ ▐ ▀ α
225..255: ß Γ π Σ σ µ τ Φ Θ Ω δ ∞ φ ε ∩ ≡ ± ≥ ≤ ⌠ ⌡ ÷ ≈ ° ∙ · √ ⁿ ² ■
```

Compatibility:

New in FS5, CP437 is new in FS7

Related:

SET CHARSET TO..., Set(_SET_GUIDRAWTEXT), Set(_SET_GUIDRAWBOX), Set(_SET_GUIDRAWLINE), Set(_SET_GUITRANSASC), SET SOURCE

SET HTMLTEXT

Syntax:

```
SET HTMLTEXT on|OFF|(<expL>)
```

Purpose:

Enables or disables embedded HTML tags within the output text.

Arguments:

ON/OFF enables/disables the support of embedded HTML tags within the console input. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

In GUI mode, FlagShip supports RichText (using a subset of HTML and XML tags) in the standard output via ?, ??, Qout(), Qqout() and @..SAY.

You have two options to interpret HTML tags:

- As long as SET HTMLTEXT is ON, any console text is interpreted in RichText format, considering HTML tags.
- Even when SET HTMLTEXT is OFF, you may preface the text string by "<HTML>" which will force this specific output to be interpreted as HTML/RichText.

Supported HTML tags are:

`<B>text_part</B>`	= print "text_part" in bold
`<I>text_part</I>`	= print "text_part" in bold
`<TT>text_part</TT>`	= print "text_part" in fixed font
`<CENTER>text_part</CENTER>`	= print "text_part" centered
`<PRE>...</PRE>`	= preserve whitespaces in the "..." text

`<FONT color="#rrggbb">text_part </FONT>`

= print "text_part" in color, where rr=red, gg=green, bb=blue RGB fraction given in hex notation (00, 80, FF), e.g.:

`<FONT color="#000000">...</FONT>`	prints black ... text
`<FONT color="#FFFFFF">...</FONT>`	prints white ... text
`<FONT color="#808080">...</FONT>`	prints gray ... text
`<FONT color="#FF0000">...</FONT>`	prints red ... text
`<FONT color="#000080">...</FONT>`	prints blue ... text

and so on. You also may use HTML color names like "yellow", "aqua" etc.

`<FONT size=nn>text_part</FONT>`

= print "text_part" in another font size, nn is the logical size (1 to 7) of the font. The value may either be absolute, for example `size=3`, or relative like `size=-2` or `size=+1`

`<FONT face=name>text_part</FONT>`	= print "text_part" in another font family of the font, for example `face=times`
`<HR>`	= draw horizontal line
`<BR>`	= new line
`<P>` or `<P>...</P>`	= new paragrap
`<IMG src=file>`	= draw image <file>

`<TABLE><TR><TD>colText</TD><TD>colText</TD></TR>...</TABLE>`
is also supported. The “colText” is any column text. You may use following <table> tags: bgcolor, width, border, cellspacing, cellpadding. The <TR> tags are: bgcolor. The <TD> tags are: bgcolor, width, colspan, rowspan, align.

Same as in HTML documents, the tags are case insensitive, i.e. "<B>" and "<b>" are equivalent. You also may combine the tags, e.g.

```
    @ 2,0 say '<B><U>underlined</U>’ + ;
              ‘<FONT color="red">red</FONT>bold</B>'
```

If you wish to display the "<" character, you need to use `"&LT;"` instead. To display the ">" character, use the `"&GT;"` tag instead. Note that the passed string is scanned for RichText tags. You therefore may also split the output into two or more parts, e.g.

```
    ?  "displaying <"
    ?? "b> as is, uninterpreted"
```

to reach the same effect. In some cases, you will need to add the "<html>" at the begin of your string to force the interpretation and/or use "<pre> text </pre>" to preserve spaces.

The Col() is adapted automatically to a larger/smaller font size but the Row() only when SET ROWADAPT is ON (default is OFF). You also may force the adaption manually by invoking RowAdapt(). Both will also consider <BR> and <P> line break tags.

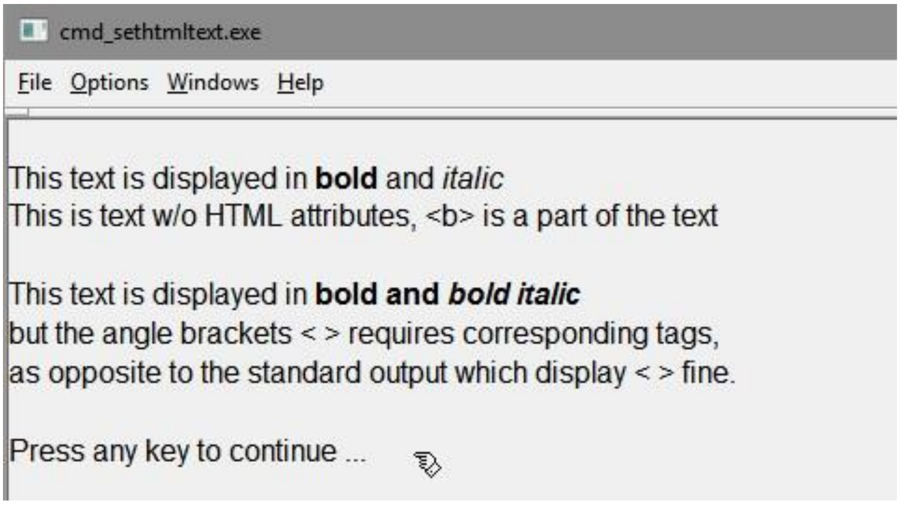
In addition to the RichText console output controlled by SET HTMLTEXT, the RichText is also supported by the MessageBox class and it subclassed functions InfoBox(), TextBox{}, Alert() etc.

Note that the RichText is interpreted in GUI mode only. Regardless the SET HTMLTEXT setting, the HTML tags are printed "as is" in the Terminal and Basic i/o mode or in the output sent to printer/file.

Example:

```
SET FONT "Arial", 12          // set default font (GUI only)
oApplic:Resize(25,80,,.T.)    // resize according to font (GUI only)

? "<html>This text is displayed in <b>bold</b> and <i>italic</i>"
? "This is text w/o HTML attributes, <b> is a part of the text"

SET HTMLTEXT ON
if AppIoMode() != "G"
   ? "Note: HTML formatting is supported for GUI mode only"
endif
? "This text is displayed in <b>bold and <i>bold italic</i></b>"
? "but the angle brackets &lt; &gt; requires corresponding tags,"
SET HTMLTEXT OFF
? "as opposite to the standard output which display < > fine."
```

Output:



```
This text is displayed in bold and italic
This is text w/o HTML attributes, <b> is a part of the text

This text is displayed in bold and bold italic
but the angle brackets < > requires corresponding tags,
as opposite to the standard output which display < > fine.

Press any key to continue ...
```

Classification:

programming, console oriented output

Translation:

Set (_SET_HTMLTEXT [, <expL>])

Compatibility:

New in FS5

Related:

?, ??, @..SAY, Qout(), Qqout(), InfoBox(), OBJ.MessageBox{}

SET INDEX TO

Syntax:

```
SET INDEX TO [<fileList> [EXCLUSIVE]]
```

Purpose:

Opens the specified index files in the current working area in the given order.

Arguments:

<**fileList**> is a comma separated list of up to 15 index file names (.idx) to be opened in the current working area. Each index file can be specified as a literal filename or as a character expression enclosed in parentheses. A file name resulting in either spaces ("") or NIL is ignored. If an extension is not specified, .idx is assumed.

SET INDEX TO without an argument closes all indexes open in the current working area; so behaving as CLOSE INDEXES.

Options:

EXCLUSIVE clause opens all indices specified in the <filelist> exclusively for the current application, regardless of the SHARED status of the database. This is very similar to the status of the index file after performing the INDEX ON command. An attempt to SET INDEX for the same index file from another user, will be denied and NETERR() set to TRUE. To reset the EXCLUSIVE status to SHARED mode, reopen the index file(s) using SET INDEX TO <filelist>.

Description:

When more than one index is opened, the first specified index becomes the controlling index and the database is positioned to the first logical record in that index. SET ORDER changes the order of the controlling index. When assigning an empty index (created by INDEX.. ..FOR), both BOF() and EOF() return TRUE and the record pointer is set beyond the end-of-file.

All open indices are properly updated according to the changes made to the database. To stop the database pointer being repositioned while updating multiple records, issue SET ORDER TO 0.

Index file names may be specified by means of macro variables or parenthesized expressions. Each file name, however, must be in a separate variable.

When the open fails, NetErr() will report .T. 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 NetErr() 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.

During index assignments, the integrity of the index file compared to the database is checked. If the check fails, the first database movement results in a warning if in FS_SET("developer") mode. For more details, see INDEX ON, INDEXCHECK() and LNG.4.5.

Multiuser:

Instead of USE..INDEX.. it is better practice to open the database, check success by USED(), then assign index/indices by SET INDEX TO.. and check success by NETERR() which should be .F.

Tuning:

As noted above, FlagShip do not raise run-time error on failure, so check by NETERR() reports failure or success. You however may force RTE 501 on failure by assigning

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

which then behaves FoxPro conform.

Example:

```
SET EXCLUSIVE OFF                        && multiuser mode
idx1 = "id_numb"
idx2 = "name"
idx3 = "salary"
IF !FILE(idx3 + INDEXEXT())              && indices available?
   USE personal EXCLUSIVE                && no, create them
   DO WHILE NETERR()
      ? "waiting to become exclusive"
      INKEY (3)
      USE personal EXCLUSIVE
   ENDDO
   INDEX ON idnumber    TO (idx1)
   INDEX ON UPPER(name) TO &idx2
   INDEX ON salary      TO salary
   USE
ENDIF

// open database and assign indices

USE personal
DO WHILE NETERR()                        && multiuser: success ?
   USE personal                          && - no, try again
ENDDO
SET INDEX TO &idx1, (idx2), &idx3

SEEK 1234                                && seek ID number
? "ID 1234", FOUND(), name
SET ORDER TO 2                           && index: name
SEEK UPPER("Smith")
DO WHILE !EOF() .and. TRIM(UPPER(name)) == "SMITH"
   ? "Smith:", FOUND(), idnumber
   SKIP
ENDDO
```

Classification:

database

Compatibility:

The index files of FlagShip (.idx) are not compatible to their counterparts in xBASE DOS dialects (.NTX or .NDX), when the default DBFIDX driver is used. For compatible code, use INDEXEXT() or FS_SET ("translext", "ntx", "idx"). Keep in mind the case sensitive file names on Unix or use FS_SET ("lower", .T.). For more details, see compatibility notes in section LNG.9.5.

After porting a DOS application and transferring the required databases (using a binary protocol), execute INDEX ON... to create the index files on the target Unix/Windows system. All .idx and database files created by FlagShip are cross-compatible to different operating systems.

The EXCLUSIVE clause and the integrity check is available in FlagShip only.

Translation:

```
DBCLEARINDEX () ; [ DBSETINDEX("index1") ...]
```

Related:

CLOSE, INDEX, REINDEX, SET ORDER, USE, INDEXEXT(), NETERR(), FS_SET()

SET INPUT

Syntax:

```
SET INPUT ON|off|(<expL>)
```

Purpose:

Enables or disables the console input.

Arguments:

ON/OFF enables/disables the console input. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is ON.

Description:

In special cases, the console input may be disabled. This setting is considered by Inkey(), InkeyTrap(), INPUT, ACCEPT but not in FReadStd(), InStdChar(), InStdString().

When the input is disabled, the input function does not check the event queue or buffer, but returns substitute character (usually ESC = 27), re-definable by Set(_SET_NOINPUTCHAR).

Classification:

programming

Compatibility:

New in FS5

Related:

Inkey(), Set()

SET INTENSITY

Syntax:

`SET INTENSITY ON|off|(<expL>)`

Purpose:

Defines whether the GETs and prompts in MENU TO will be displayed in the "standard" or the "enhanced" color.

Arguments:

ON/OFF enables/disables the enhanced color setting. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

When INTENSITY is ON (the default), the active GET field in READ appears in the enhanced, all other GET fields in the "unselected" color, as specified or default. The light bar in MENU TO marking the current PROMPT selection also appears in the "enhanced" color and the cursor is hidden.

By setting INTENSITY OFF, GETs and the current PROMPT appear in the standard color. The cursor remains visible.

SET INTENSITY has no effect on ACHOICE() and DBEDIT().

Example:

```
IF FS_SET("term") == "dummy"            && which TERM used?
   SET INTENSITY OFF
ENDIF
SET FORMAT TO authors
USE authors
READ
SET FORMAT TO
USE
SET INTENSITY ON
```

Classification:

programming

Translation:

SET (_SET_I NTENSI TY, . T. | . F.)

Related:

@..GET, READ, @..PROMPT, MENU TO, SET COLOR, SETCOLOR(), SET CURSOR, SETSTANDARD, SETENHANCED, SET()

SET KEY ... TO

Syntax:

```
SET KEY <expN> TO [<procname>]
```

Purpose:

Defines a procedure to be executed whenever the specified key is pressed in a wait state.

Arguments:

<**expN**> is the ASCII value of the key, including negative numbers for function keys, see INKEY() values.

Options:

<**procname**> is the name of the procedure to be executed when the key is pressed. If <procname> is not specified, the current redirection of the <expN> key is canceled.

Description:

When the defined procedure is executed from a wait state, FlagShip invokes the UDP similarly as in the usual procedure call

DO <procname> WITH PROCNAME(), PROCLINE(), READVAR() [,lastKey]

but will use an internal code block instead. Using these parameters, context sensitive reactions or help from within the procedure can be implemented. Note that code-block and the some of the procedure names in the call-stack may be filtered out, so the first and second passed parameter may differ from ProcName() and ProcLine(), see details in FUN.HELP().

expN	Associated Key	Notes	Unix terminfo
28	*F1*		*kf1*
-1 ... -9	*F2 - F10*		*kf2...kf10*
-10 ... -19	*shift F1 - F10*		*kf13...kf22*
-20 ... -29	*ctrl F1 - F10*		*kf25...kf34*
-30 ... -39	*alt F1 - F10*	*	*kf37...kf46*
-40 ... -41	*F11 - F12*	**	*kf11...kf12*
-42 ... -43	*shift F11 - F12*	**	*kf23...kf24*
-44 ... -45	*ctrl F11 - F12*	**	*kf35...kf36*
-46 ... -47	*alt F11 - F12*	* **	*kf47...kf48*
18, 3	*PgUp, PgDn*	+	*kpp, kpn*
1, 2, ...	*Ctrl-A, Ctrl-B, ...*		-
65, 66, 67..	*A, B, C, ...*		-

* The "Alt-FN" keys are sometimes not available on Unix terminals, but are then often supported by Ctrl+Shift+FN, see terminfo (e.g. FStinfo.src). In X11 and Windows, several Alt-FN combinations are hard-wired to window manager

** The F11 and F12 key combinations are not supported by all of the DOS derivatives. On Unix, their usage depends on the terminal capability (see FStinfo.src).

+ See INKEY() or the "inkey.fh" file for other numeric codes and their DEFINE equivalents.

The SET KEY redirection is active in ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT but not in INKEY(). For its usage or simulation in INKEY(), see getsys.prg.

A maximum of 32 keys may be defined/redirected at one time. The F1 key is initially redirected to a procedure named HELP, if such exists.

SET KEY has precedence over SET FUNCTION, SET ESCAPE and SETCANCEL(). A maximum of 32 keys can be set at the same time. The SET KEY redirection is a global setting and therefore also remains active during an invocation of other UDFs or UDPs or on returning to a higher program level.

When designing a "background" procedure, it is good programming technique to preserve the current status of the application (i.e., screen appearance, current working area, etc.) and to restore it before exiting. CLEAR should not be used to clear the screen within a "background" procedure since it also clears GETs and therefore terminates READ. Use CLS, CLEAR SCREEN or @...CLEAR instead. To terminate the current READ from a "background" procedure, issue:

Command	Action
CLEAR GETS	Terminate READ, do not save current GET
BREAK	Terminate READ, do not save current GET
KEYBOARD chr(23)	Terminate READ, save the current GET
KEYBOARD chr(27)	Terminate READ, do not save current GET

When using the redirection to a STATIC PROCEDURE (or STATIC FUNCTION), the same rules as in code blocks apply: the SET KEY command must be specified in the same .prg file, where the STATIC procedure is defined; otherwise the <procname> will be invisible/undefined. When using redirection to a global procedure or UDF, the SET KEY can be defined anywhere.

You also may automatically invoke / trigger procedure, function or code block in specific time intervals by using TriggerUdf() from FS2 Toolbox.

The HELP procedure or function (see FUN.HELP) is a special case of "background" procedure. It is already pre-defined and assigned to F1-key at program begin.

Example 1:

```
while lastkey() != K_ESC
  SET KEY K_F1 to help         // F1
  SET KEY -1   to mykey        // F2
  SET KEY K_F5 to mykey        // F5
  SET KEY asc('a') to mykey    // a
  SET KEY asc('B') to mykey    // B
  wait "Press F1, F2, F5, a, B or ESC" NOECHO
enddo
```

```
FUNCTION mykey()
alert("Key " + inkey2str(lastkey()) + " pressed")
return
FUNCTION help()
alert("This is my help")
return
```

Example 2:

Using the SET KEY redefinition to display the available article groups, when the [F3] key is pressed while being located in the entry field "group". In this example, it is assumed that there are just a few records in the artgroup.dbf database (otherwise, use DBEDIT() or TBROWSE instead of ACHOICE()).

```
#include "inkey.fh"
STATIC showarr := NIL

PROCEDURE main
LOCAL menu
USE article INDEX article NEW
menu := menu_choice()
IF menu == 1                     // new entry
   IF new_modif (.T.)
      APPEND BLANK
      REPLACE ...
   ENDIF
ELSEIF menu == 2                 // modify
   IF new_modif (.F.)
      REPLACE ...
   ENDIF
ENDIF

FUNCTION new_modif (newentry)
PRIVATE Xarticle, Xgroup, Xprice

IF newentry
   Xarticle := 0
   Xgroup   := space(20)
   Xprice   := 0
ELSE
   Xarticle := article->article
   Xgroup   := article->group
   Xprice   := article->price
ENDIF
SET KEY K_F3 TO show_group
@ 5, 10 GET Xarticle PICTURE "999999"
@ 6, 10 GET Xgroup   PICTURE "!!!!!"
@ 7, 10 GET Xprice   PICTURE "99,999.99"
READ
RETURN LASTKEY() # K_ESC

PROCEDURE show_group (procName, procLine, varName)
LOCAL savescr := SAVESCREEN (10,50,MAXROW(),79)
LOCAL choice
IF .not. (varName == "XARTICLE" .or. varName == "XGROUP")
   RETURN
```

```
END
@ 10,50 CLEAR TO MAXROW(),79
@ 10,50 TO MAXROW(),79 DOUBLE
IF VALTYPE(showarr) != "A"               // initialized ?
   initArray()                           // no, do it now
ENDIF
choice = ACHOICE (11,51, MAXROW() -1, 78, showarr)
IF choice > 0
   Xgroup := SUBSTR(showarr[choice], 1, 5)
END
RESTSCREEN (10,50,MAXROW(),79, savescr)
RETURN

FUNCTION initArray
LOCAL act_select := SELECT()
IF VALTYPE(showarr) != "A"               // initialized ?
   showarr := {}                         // not yet
   USE artgroup INDEX artgroup NEW
   WHILE !eof()
      AADD (showarr, group + " " + textgroup)
      SKIP
   END
   SELECT (act_select)
END
RETURN NIL

Compile: $ FlagShip test.prg -Mmain -na
```

Classification:

programming

Compatibility:

Most other xBASE dialects do not support F11 and F12 keys and their combinations. In Terminal i/o mode, refer to sections SYS, REF and the current terminfo file (e.g. <FlagShip_dir>/terminfo/ FStinfo.src) for FN keys available according to the currently assigned terminal (by TERM, FSTERM, TERMINFO and FSTERMINFO envir. variables, see section FSC.3.3). The SET KEY command of dBASE IV has another functionality, but its ON KEY..DO.. is very similar to FlagShip's (and Clipper's) SET KEY.

Include:

The INKEY() key numbers <expN> are defined in the #include "inkey.fh" file.

Translation:

SETKEY (expN, {|p1, p2, p3, p4| procName(p1, p2, p3, p4)})

Related:

HELP(), SET FUNCTION, KEYBOARD, LASTKEY(), PROCLINE(), PROCNAME(), READVAR(), SETKEY(), FS2:TriggerUdf()

SET KEYTRANSL

Syntax:

```
SET CHARSET|KEYTRANSL [TO] ISO|ANSI
SET CHARSET|KEYTRANSL [TO] PC8|ASCII|OEM
```

Purpose:

Translates the keyboard values > 127 to Inkey() value and the screen output accordingly. Applicable/considered in GUI mode only.

Arguments:

ANSI|ISO use default keyboard scan codes corresponding to your keyboard setting (which are ISO/ANSI values in GUI mode). The inkey codes are taken from the table in Fsguikey.def, user definable via FS_SET("guikey",file_name)

PC8|ASCII|OEM activates an automatic translation of the inkey value from ISO/ANSI to PC8/ASCII/OEM character set. This will produce same Inkey value as

```
    key := Ansi2oem( Inkey(0) )
```

with SET KEYTRANSL set to ANSI

Description:

In GUI i/o mode, both the screen input and output are handled in ISO/ANSI mode per default.

SET KEYTRANSL is mainly used to map/translate an user input to the same character set/mode used also for output.

● If you are using ISO/ANSI/Windows character set for your source code (i.e. the editor is for GUI mode or MS-Windows character set), you don't need change the default setting SET GUITRANSL ASCII OFF and SET KEYTRANSL ISO. In this mode, the u-umlaut is represented in ISO-8859-1 charset by chr(252) - as opposite to chr(129) in PC8/ASCII mode.

● If you prefer to use PC8/ASCII character set coding (same as in DOS/Clipper or in the most of terminal applications), you may set

```
   SET SOURCE ASCII          // translate output and input
```

which is also set in

```
   #include "fspreset.fh"    // see LNG.9.5
```

The generalized command SET SOURCE ASCII is a shortcut for

```
   SET GUITRANSL ASCII ON      // translate output
   SET KEYTRANSL ASCII         // translate input
   Set(_SET_PRINTASCII,  .F.)  // don't translate printer ISO->ASCII
   Set(_SET_SOURCEASCII, .T.)  // source is in ISO character set
```

It will then display chr(129) as u-umlaut, and Inkey() will return 129 when pressing the u-umlaut key. With separate SET GUITRANSL and SET KEYTRANSL you however may precise control a different behavior.

Example:

```
see example in SET SOURCE
```

Classification:

programming

Translation:

```
SET KEYTRANSL ISO    =  SET(_SET_CHARSET, _SET_CHARSET_ISO)
SET KEYTRANSL ANSI   =  SET(_SET_CHARSET, _SET_CHARSET_ISO)
SET KEYTRANSL ASCII  =  SET(_SET_CHARSET, _SET_CHARSET_PC8)
SET KEYTRANSL PC8    =  SET(_SET_CHARSET, _SET_CHARSET_PC8)
SET KEYTRANSL OEM    =  SET(_SET_CHARSET, _SET_CHARSET_PC8)
```

Compatibility:

New in FS5

Related:

Ansi2oem(), Oem2Ansi(), FS_SET("ansi2oem"), SET ASCII, SET ANSI, SET GUITRANSL ASCII, SET SOURCE

SET LARGEFILE

Syntax:

```
SET LARGEFILE ON|off|(<expL>)
```

Purpose:

Sets or disables the capability of large file support.

Arguments:

ON/OFF enables/disables the capability of large file support for databases over 2 Gigabytes. With LARGEFILE ON, the system limit is increased up to 16 Terabytes (system dependant). The default setting is ON. Set it to OFF to ensure backward compatibility to available databases. Alternatively, the parenthesized <expL> may be used, whereby .T. is the same as ON and .F. same as OFF.

Description:

Enable LARGEFILE to create or manage databases exceeding the 2GB limit. Available only on operating systems supporting large files, see below. You may check the status by Set(_SET_LARGEFILE) after executing SET LARGEFILE; it returns .T. when large files are supported.

The setting is effective at the time of opening the database or accessing other files.

Of course, even with LARGEFILE ON the file size cannot exceed the physical file system limit for file size, which is:

FAT16, VFAT	DOS, Windows, Linux, others	2 GB
FAT32, VFAT	Windows, Linux, others	4 GB
FAT64, exFAT	Windows, Linux, Apple	16 EB
NTFS	Windows32/64	16 EB
ext3	Linux	16 GB to 2 TB
ext4	Linux	16 GB to 16 TB
ReiserFS	Linux	4 GB to 16 TB
Btrfs	Linux, Sun, Oracle	16 EB
JFS	Linux, AIX	4 PB
XFS	Linux, Irix	8 EB
ZFS	Linux, Solaris	16 EB
HFS	Apple Mac OS X	2 GB
HFS+	Apple Mac OS X	8 EB
ISO 9660	CDROM, DVDROM	4 GB to 8 TB

Example:

```
SET LARGEFILE ON
if !set(_SET_LARGEFILE)
   ? "Large file support not available; this operating system"
   ? "does yet not support it"
   wait
   quit
endif
USE myData SHARED
...
```

Classification:

programming, databases, file input/output

Compatibility:

SET LARGEFILE is available in FS6 (and up) only. In VFS6 and VFS7 the default was OFF, in VFS8 and later the default is ON.

Translation:

SET (_SET_LARGEFILE, <expL>)

Related:

USE

SET MARGIN TO

Syntax:

`SET MARGIN TO [<expN>]`

Purpose:

Sets the left margin for all printed output.

Arguments:

<**expN**> is the column number or the margin size in current coordinates to which the left margin is to be set. The default margin value is zero.

Description:

SET MARGIN affects the printer output according to the current i/o mode:

- In Terminal or Basic i/o mode, when SET PRINTER is ON or the ...TO PRINTER clause is used, the <expN> number of spaces is printed in front of a new line. With SET DEVICE TO PRINTER, the <expN> value is added to column during the @..SAY output. The PCOL() value reflects the current print column position, including the margin. You may tune the printer output by FS_SET("prset") which may be advantageous when using proportional character set etc.
- In GUI mode with standard printout, i.e. when SET GUIPRINTER is OFF or SET PRINTER GUI is OFF, and PrintGui(.T.) is not set (all these are defaults), the behavior is the same as in Terminal i/o mode.
- In GUI mode with GUI / GDI printer output, i.e. when PrintGui(.T.) was set (or SET GUIPRINTER is ON), the <expN> represents the number of spaces added in front of any printer line. If (at the time of SET MARGIN) the current font is proportional, the size of letter "M" is used instead of space. The <expN> value may be given also as decimal fraction, the size (internally recalculated into printer dots) is added to oPrinter:MarginLeft, set by printer driver. In this mode, the FS_SET("prset") tuning is not considered, and the PCOL() return value is not affected by <expN>.

SET MARGIN has no effect on SCREEN and FILE or EXTRA output. The SET COORD UNIT is not considered here.

Example 1: print to device or spooler file

```
// SET PRINTER TO LPT3
// SET PRINTER TO /dev/lp2
// SET PRINTER TO ("myprint.txt")
SET MARGIN TO 10             // add 10 spaces at left margin
SET CONSOLE OFF              // disable screen output

USE address
LIST Name, Area, Subarea TO PRINTER   // print to device or file

SET CONSOLE ON               // enable screen output
// SET MARGIN TO             // reset margin to 0
// SET PRINTER TO            // close spooler file if any
```

```
 wait "printed to " + fs_set("printfile") + " - any key..."
```

Example 2: create spooler file and print to GDI printer with preview

```
 SET GUIPRINTER ON            // prints to GDI printer (in GUI mode)
 SET FONT TO "courier",8      // use small printer font
 SET MARGIN TO 5              // left printout margin will be 5
spaces
 SET PRINTER ON               // activate printer output

 // SET CONSOLE OFF           // optional, don't print to screen
 SET EJECT ON                 // autom. EJECT on full page

 USE address
 LIST Name, Area, Subarea     // print to spooler file or GUI printer

 SET PRINTER OFF              // printout is ready
 SET CONSOLE ON               // enable screen output
 // SET PRINTER TO            // close spooler file if set
 // SET MARGIN TO             // reset margin to 0

 if AppIoMode() == "G"        // only if running in GUI mode:
    ok := PrintGui()          // flush printout
 else
    ? "printed to " + fs_set("printfile")
 endif
 wait "done..."
```

Classification:

programming

Translation:

SET (_SET_MARGIN, expN)

Related:

@...SAY, SET DEVICE, SET PRINTER, SET GUIPRINTER, FS_SET("prset"), PCOL(), PrintGui(), OBJ:Printer class

SET MEMOFILE TO

Syntax:

`SET MEMOFILE TO [ DBT | FPT ]`

Purpose:

Specifies the kind of memo file for "M" data field for newly created databases by DbCreate(), CREATE FROM and COPY TO.

Arguments:

SET MEMOFILE TO DBT is default setting, creates file <dbfname>.dbt when "M" field is available in the database structure. If so, the first byte of the database (.dbf) header contains 0x83 = chr(131). SET MEMOFILE TO DBT can co-exist with "V*" variable data field which (additionally) creates <dbfname>.dbv file. In this case the database header contains 0x93 = chr(147) in first byte for .dbt and .dbv, or 0x13 = chr(19) for .dbv only.

SET MEMOFILE TO FPT is optional, it creates Foxbase/FoxPro compatible file <dbfname>.fpt when "M" field is available in the database structure. In this case, the first byte of the database (.dbf) header will contain 0xF5 = chr(245). The SET MEMOFILE TO FPT cannot coexist with "VC*" variable data fields.

SET MEMOFILE TO causes reset to the default SET MEMOFILE TO DBT

Description:

SET MEMOFILE TO... is considered only during creation of new database when "M" memo field is available in the database structure. It has no effect on opening available databases by USE or DbUseArea() where the the first byte of the database (see above) specifies the kind of used memo file. The setting remain active until program ends, or a new SET MEMOFILE is assigned.

Example 1:

```
aStru := {{"name","C",20,0}, {"ID","N",6,0}, {"text","M",10,0}}
dbcreate("dbf1", aStru)
_displarrstd(directory("dbf1.*"))   // dbf1.dbf and dbf1.dbt
SET MEMOFILE TO FPT
dbcreate("dbf2", aStru)
_displarrstd(directory("dbf2.*"))   // dbf2.dbf and dbf2.fpt
dbcreate("dbf3.xyz", aStru)
_displarrstd(directory("dbf3.*"))   // dbf3.xyz and dbf3.fpt
```

Example 2:

See also examples in COPY TO and DbCreate()

Classification: database creation
Compatibility: Available in FlagShip VFS7 and newer only
Translation:

```
SET MEMOFILE TO DBT => _aGlobSetting[GSET_N_DBCREAMEMO] := 1
SET MEMOFILE TO FPT => _aGlobSetting[GSET_N_DBCREAMEMO] := 2
```

Related:

DbCreate(), COPY TO, CREATE FROM

SET MESSAGE TO

Syntax:

`SET MESSAGE TO [<expN> [CENTER|CENTRE] ]`

Purpose:

Defines the row and centering for the display of @...PROMPT messages when executing MENU TO and/or the row for StatusMessage() in Terminal i/o mode.

Arguments:

<**expN**> is the row where the messages will be displayed. If there is no argument, or if <expN> is zero, messages will not be displayed.

Options:

CENTER: When specified, the message texts are centered. Otherwise, each message starts at column zero.

Description:

When the clause MESSAGE is specified by the @... PROMPT command, MENU TO displays this message text on the row, given by SET MESSAGE when the PROMPT item is selected. This can be used for a short context specific help.

The StatusMessage() text in Terminal i/o mode displays only when SET MESSAGE was specified. In GUI mode, the StatusMessage() text displays in StatusBar at bottom of the application window, independent on SET MESSAGE. To remain compatible in GUI to Terminal i/o, you may invoke SET(_SET_MESSAGE_GUI, .T.) which will then display the PROMPT message in the row <expN>, optionally centered.

Example:

```
SET MESSAGE TO 22 CENTER                       // msg on line 22
@ 21,0 TO 21,79 DOUBLE                         // draw line
@  5,30 PROMPT "Append" MESSAGE "Add and edit a new record"
@  6,30 PROMPT "Change" MESSAGE "Edit current selectd. data"
@  8,30 PROMPT "Quit"   MESSAGE "Exit to main menu"
MENU TO choice
```

Classification:

programming (and screen oriented output in MENU)

Compatibility:

SET(_SET_MESSAGE_GUI, log) is available in VFS8 and newer

Translation:

SET (_SET_MESSAGE, expN) [; SET (_SET_MCENTER, .T.|.F.)]

Related:

@...PROMPT, MENU TO, StatusMessage()

SET MULTIBYTE

Syntax:

```
SET MULTIBYTE on|OFF|(<expL>)
```

Purpose:

Enables the capability of multi-byte Unicode input.

Arguments:

ON/OFF enables/disables the capability of READ and other input processes to handle multi-byte character set (like Chinese etc). The default setting is OFF. Alternatively, the parenthesized <expL> may be used, whereby .T. is the same as ON.

Description:

Enabled MULTIBYTE support will handle Unicode (like Asian) glyphs in GET/READ and MemoEdit(). Multi-byte characters contains two to four bytes chr(128..255). Lower ASCII characters chr(1..127) are handled as single-byte and can be intermixed with multi-byte glyphs in the same entry or line.

When enabled and upper ASCII character chr(128..255) is entered, READ and MemoEdit() waits for next characters of the glyph.

On Linux in Terminal i/o mode, you will need to use proper window Terminal (e.g. mlterm or xiterm instead of Gnome/KDE console) for a correct support of multi-byte characters, see example below.

Since SET MULTIBYTE ON interprets chr(128..255) as begin of glyph (if not changed, see Tuning and LNG.5.4.5), you cannot enter international single-byte characters like Umlauts etc. in this mode. With MULTIBYTE ON, you may detect slight flickering on slower computers.

Helpful links:
http://www.xuexizhongwen.de/chinese_t7.htm
http://www.xuexizhongwen.de/index.htm?computing_t20.htm&1
http://www.schaepermeier.de/linux/l_japanisch_d.htm
http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/ Chinese-HOWTO.pdf
http://www.suse.de/~mfabian/suse-cjk.pdf

Tuning:

In GUI mode, all Unicode characters in UTF-8 encoding are supported. In Terminal i/o mode, Unicode UTF-8 is partially supported in Linux too. In the READ and MemoEdit() handler, Inkey() collects separate UTF-8 bytes, waiting max a specified time period, which is set per default to 50 milliseconds but can be changed by assigning

```
_aGlobSetting[GSET_N_MEMO_GET_MULTIW] := 0.05  // default
```

Example:

```
SET MULTIBYTE ON
myData := space(40)
@ 1,1 SAY "enter Chinese chars" GET myData
READ
```

Example of start-up script in Linux:

```
#!/bin/sh
# shell requirement for Linux in Terminal i/o mode:
export LC_CTYPE=zh_TW.Big5
export LANG=zh_TW.Big5
export LC_ALL=zh_TW.Big5
## xcin &
## scim &
skim &
XMODIFIERS="@im=skim"; export XMODIFIERS
echo "---chinese"
echo "LANG=$LANG"
## echo ".. opening xiterm"
## xiterm &
echo ".. opening mlterm"
## mlterm &
mlterm --bg=black --fg=white --term=mlterm &
```

Classification:

programming, screen input/output

Compatibility:

SET MULTIBYTE is available in FS6 (and up) only.

Translation:

SET (_SET_MULTI BYTE, <expL>)

Related:

@..GET, READ, MemoEdit(), Inkey(), Unicode LNG.5.4.5

SET MULTILOCKS

Syntax:

`SET MULTILOCKS on|OFF|(<expL>)`

Purpose:

Sets or disables the capability of multiple record locking.

Arguments:

ON/OFF enables/disables the capability to RLOCK() multiple records. The default setting is OFF. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

When a database is open in SHARED (multiuser) mode, any write access requires a record or file lock. Usually, the RLOCK() locks the current record only, and FLOCK() is used for multiple record replacements. For a large application with many users, locking a specified region of the database for a transaction may be more efficient, allowing the remaining records to be replaced also by others.

When MULTILOCKS is OFF (the default), any attempt to RLOCK(), AUTORLOCK(), FLOCK() or APPEND BLANK will release all previous locks.

When MULTILOCKS is set ON, any consecutive attempt to RLOCK() or AUTORLOCK() (but not APPEND BLANK or DBAPPEND()) will add the record number to an internal list of locked records. FLOCK() releases all previous record locks first.

The UNLOCK (or UNLOCK ALL) command will release all locked records, regardless of the SET MULTILOCKS state. You may release a specific RLOCK by using DBRUNLOCK().

Note, the oRDD:RLOCK() does not consider the SET MULTILOCKS state, but determines it from the parameter passed.

Example:

```
USE mydbf SHARED
goto 5
RLOCK()                             // locked: recno 5 only
RLOCK(10)                           // locked: recno 10 only
? RECNO()                           // 5 (remains unchanged)
SET MULTILOCKS ON
goto 3
RLOCK()                             // locked: recno 3 and 10
RLOCK(7,8)                          // locked: recno 3, 7, 8 and 10

oRdd := DBOBJECT()
? "List of locked records:"
aeval (oRdd:RlockList, {|x| qout("RecNo", x)} )
UNLOCK                              // release all locks
```

Classification:

database

Compatibility:

SET MULTILOCKS is not available in C5 and VO, but compatible to FoxPro.

Note for FoxPro users: SET MULTILOCKS will not perform an automatic UNLOCK ALL. If such behavior is required, you may add the statement

```
#command SET MULTILOCKS <x:ON,OFF,&> => ;
   DBUnlockAll() ; SET(_SET_MULTILOCKS, <x>)
```

in your source file, or at the end of the std.fh file.

Translation:

SET (_SET_MULTILOCKS, <expL>)

Related:

RLOCK(), UNLOCK, DBRUNLOCK(), SET(), oRdd:RLOCK(), oRdd:RLOCKLIST

SET NFS

Syntax:

```
SET NFS on|OFF|(<expL>)
SET NFS_FORCE on|OFF|(<expL>)
SET NFSLOCK on|OFF
```

Purpose:

Enable additional security handling and buffer flushing for databases and indices mounted via NFS or SAMBA

Arguments:

ON/OFF enables/disables special handling of index files on NFS file system. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

In some NFS versions (e.g. Linux 2.4.x NFS server), and in SAMBA, the buffer caching is over-optimized, so the standard FlagShip locking and file flushing does not force the server to update all buffers to the hard disk, especially on heavy loaded server. In some cases, the insufficient system cache flushing may corrupt the database or index(es) located on the server.

As long as SET NFS is ON, FlagShip's DbfIdx RDD issues additional actions like internal locks and forced data flushing to fix the above described NFS or SAMBA server problem. Since this may slow-down the performance for pure server or local based access, best to enable SET NFS ON only in applications that access remotely mounted databases and indices via NFS or SAMBA.

For your convenience, the USE command has optional NFS clause too; this will invoke SET NFS ON, so the special NFS handling remains active also for all subsequent database actions, until SET NFS is set OFF.

Note: To mount the remote filesystem via NFS, you will need to use at least nfsvers=3,rw,lock,sync mount options. But you will get better performance, when the executable run on the server (because of local HD access) with SET NFS OFF (default) by "shuffle" only the user i/o through the network. You may execute the application via ssh, telnet, emulators, X11 redirection or X11 emulator for MS-Win, CGI, mirroring etc. see also http://www.fship.com/emulators.html for details.

Note: to mount the remote filesystem via SAMBA (e.g. for concurrent Unix/Linux and MS-Windows access), you need to specify/enable at least "path = /your_share_path", "read only = no", "create mask = 0664", "directory mask = 0775" in the [your_share_drive] section of /mnt/server/etc/samba/smb.conf and "workgroup = your_windows_group", "unix extensions = Yes", "security = user", "encrypt passwords = Yes", "client code page = 850", "character set = ISO8859-15" in the [global] section of the same Samba config file. You will then access the <your_share_path> via usual mount as nfs (or as smbfs) type and from MS-Windows as usual network drive.

For concurrently use of Unix/Linux and MS-Windows based FlagShip applications (usually on SAMBA system), you often need to use SET NFS ON to avoid flushing problems.

Classification:

programming, database

Compatibility:

New in FS5

Translation:

```
SET ( _SET_NFS_FORCE, <expL> )
```

Related:

USE, DbUseArea(), SET INDEX TO, IndexCheck()

SET OPENERROR

Syntax:

```
SET OPENERROR ON|off|(<expL>)
```

Purpose:

Display database and/or index open failure.

Arguments:

ON/OFF enables/disables raising run-time-error message when the database and/or index file could not be opened, e.g. because the file is not available or there is not sufficient permission. The default is ON, which means the open i/o RTE are displayed. If set OFF, you need to check the USE or SET INDEX TO success manually via Used() and NetErr(). Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

The SET OPENERROR command allows you to control and to react on database and index open failure manually checking the Used() and NetErr() status.

Classification:

programming

Compatibility:

New in FS5

Translation:

```
Set( _SET_OPEN_ERROR, <expL> )
```

Related:

USE, Used(), DbUseArea(), SET INDEX, DbSetIndex(), NetErr()

SET ORDER TO

Syntax 1:

```
SET ORDER TO [<expN>]
```

Syntax 2:

```
SET ORDER TO TAG <expC1> [IN <expC2>]
```

Purpose:

Identifies the specified index number as the (master) controlling index.

Arguments:

<**expN**> specifies which index number, according to the position in the list of open indices, will become the controlling index. The valid range is 0 to 15. If <expN> is not specified, ORDER is set to zero.

TAG <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.

IN <expC2> clause is ignored.

Description:

When assigning the associated indices to the working area using SET INDEX TO or USE...INDEX, all these indices are automatically updated while changing the database fields.

By using SET ORDER, the required index is declared as the controlling one. Changing the index order does not change the database record position.

SET ORDER TO 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.

Tuning:

To avoid resetting FOUND() value to .F. when changing index order, set the global switch to

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

which then behaves similarly to Clipper.

Example:

```
USE employees
SET INDEX TO id, name, born, salary
? RECNO(), Idno, Lastname, Salary       &&  1, Jones, 25000
SET ORDER TO 2
? RECNO(), Idno, Lastname, Salary       &&  1, Jones, 25000
GO TOP
? RECNO(), Idno, Lastname, Salary       && 52, Aaron, 23500
SET ORDER TO 0
REPLACE ALL lastname WITH "Mueller" ;
    FOR TRIM(lastname) == "Müller"
SET ORDER TO TAG "name"
SEEK "Müller"                           && not found
SEEK "Mueller"                          && found
```

Classification:

database

Translation:

DBSETORDER (expN)

Related:

INDEX, REINDEX, SET INDEX, USE, INDEXORD(), INDEXEXT(), INDEXKEY(), INDEXCHECK(), DBSETORDER(), FOUND(), oRdd:SetOrder()

SET OUTMODE

Syntax:

```
SET OUTMODE [TO] [<expN>]
```

Purpose:

Designates how to print zero-bytes and unprintable characters < 32 on the screen.

Arguments:

<**expN**> is the desired output mode:

0:	print all "as is", chr(0) may terminate the string
1:	replace chr(0) by character specified in SET(_SET_ZEROBYTEOUT) which is per default "?", print all other characters "as is"
2:	print characters < 32 as "^x", i.e. chr(0) -> ^@, chr(3) -> ^C
3:	same as 2, except chr(7), chr(10), chr(13)
4:	print characters < 32 as backslash-escaped octal value -> \nnn
5:	same as 4, but enclosed in curly brackets -> {\nnn}
6:	print characters < 32 as hexadecimal value -> 0xNN
7:	same as 6, but enclosed in curly brackets -> {0xNN}
8:	print characters < 32 as CHR(nn)
9:	same as 8, but enclosed in curly brackets -> {CHR(nn)}

The default setting is 1. This is also set when <expN> is not given.

Description:

The standard screen output via OutStd(), OutErr() or the ?, ?? commands and QOUT(), QQOUT() functions cannot print all characters below CHR(32) to the screen. With SET OUTMODE or SET(_SET_OUTMODE) you may decide how to handle them. When the output is redirected to another device than console/screen, SET OUTMODE is ignored.

Example:

```
str := "ab" + chr(0) + "cd" + chr(3) + "ef" + chr(7) + "gh"
? Set(_SET_OUTMODE)       // 1
? Set(_SET_ZEROBYTEOUT)   // "?"
? str                     // "ab?cdefgh"
SET OUTMODE 0 ; ? str     // "ab"  or  "abcdefgh"
SET OUTMODE 1 ; ? str     // "ab?cdefgh"
SET OUTMODE 2 ; ? str     // "ab^@cd^Cef^Ggh"
SET OUTMODE 3 ; ? str     // "ab^@cd^Cefgh"
SET OUTMODE 4 ; ? str     // "ab\000cd\003ef\007gh"
SET OUTMODE 5 ; ? str     // "ab{\000}cd{\003}ef{\007}gh"
SET OUTMODE 6 ; ? str     // "ab0x00cd0x03ef0x07gh"
SET OUTMODE 7 ; ? str     // "ab{0x00}cd{0x03}ef{0x07}gh"
SET OUTMODE 8 ; ? str     // "abCHR(0)cdCHR(3)efCHR(7)gh"
SET OUTMODE 9 ; ? str     // "ab{CHR(0)}cd{CHR(3)}ef{CHR(7)}gh"
SET OUTMODE TO
wait
```

Classification:

programming

Compatibility:

New in FS5

Translation:

Set (_SET_OUTMODE, <expN>)

Related:

?, ??, QOUT(), QQOUT(), OUTSTD(), OUTERR(), SET CONSOLE, SET DEVICE

SET PATH TO

Syntax:

```
SET PATH TO [<pathList>|(<expC>)]
```

Purpose:

Sets the path that FlagShip will search when attempting to open files.

Arguments:

<**pathList**> is a list of paths that FlagShip is to search if a specified file is not located in the current directory. The list of paths is separated by commas or semicolons. Other separators, like the usual Unix colon (`:`) or space separator, can be specified using FS_SET("pathdelim"). Each path gives the absolute or relative directory name separated by slashes or backslashes. "\" will be automatically translated to the Unix syntax "/". The line continuation of a SET PATH command with a semicolon (`;`) is not supported. For long path names, use a character expression enclosed in parentheses.

SET PATH TO with no argument releases the path list.

Description:

When a file is to be accessed, and the path is not given as a part of the file name, FlagShip searches for existing files

- in the current Unix/Windows directory,
- in the path given by SET DEFAULT,
- in all path names specified by SET PATH.

Note that low-level file functions and the SAVE TO, RESTORE FROM or RUN commands do not respect either the DEFAULT or PATH settings. The RUN command considers the PATH= variable of the Unix or Windows shell.

To **create** a new file outside the current directory, either an absolute path or a SET DEFAULT must be specified.

The path (and file) names are case-sensitive in Unix. FlagShip offers different levels of automatic conversion of DOS names, executed during a file or directory access:

- FS_SET("pathlower"|"pathupper",.T.) converts any given path to lower or upper case,
- FS_SET("lower"|"upper",.T.) converts any given file name and extension to lower or upper case,
- FS_SET("translext","ntx","idx") translates the specified extension to another,
- FS_SET("pathdelims",",;: ") to specify the path delimiters for the SET PATH command,

- x_FSDRIVE environment variable substitutes the used DOS drive selector "x" (like C:, D: etc.) in the program path with a Unix directory.

Example:

```
LOCAL path1 := "C:\data1"           // DOS/Windows Syntax
LOCAL path2 := "/usr/data2"         // Unix Syntax
LOCAL path3 := "../../data3"        // relative path
FS_SET ("PathDelim", ",;: ")        // set path delimiters
FS_SET ("PathLower", .T.)           // path translation
FS_SET ("Lower", .T.)               // data translation

IF EMPTY(GETE("C_FSDRIVE"))         // check C: substitut.
   ? "set C_FSDRIVE path first"
   QUIT
ENDIF

SET PATH TO .\data;/usr/data
IF .not. FILE("address1" + INDEXEXT())
   SET PATH TO (path1 + ";" + path2 + ":" + path3)
ENDIF
IF .not. FILE("address.dbf")
   SET PATH TO (GETENV("PATH"))      // Unix/Windows environment
END
USE address INDEX address1
```

Classification:

programming, file access

Translation:

SET (_SET_PATH, "pat h")

Related:

SET DEFAULT, CURDIR(), FS_SET(), (FSC)environment

SET PIXEL

Syntax:

```
SET PIXEL on|OFF|(<expL>)
```

Purpose:

Set default pixel or row/col coordinates

Arguments:

ON/OFF enables/disables the specification of coordinates in pixels or in col/row values in GUI mode. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

Description:

In GUI mode, all widgets (or controls in MS-Windows terminology) are pixel based. To enable the common Xbase (FlagShip, Clipper, FoxPro etc) compatibility, FlagShip internally recalculates the col/row coordinates to pixels according to the used font.

The calculation of one row is the line height (font specific) + line inter-spacing. One column is assumed the largest width of the characters "358AMX". You may change this calculation by setting the global variable

```
  _aGlobSetting[GSET_G_C_COL_MAXCHAR ] := "358AMX"
  _aGlobSetting[GSET_G_N_ROW_SPACING ] := 2
```

correspondingly, see also system/initio.prg and include/set.fh

The most commands or functions using coordinates accept optional clause PIXEL|NOPIXEL or an logical/NIL argument which temporarily overrides the global SET PIXEL declaration. An alternative to other coordinate units is SET CORRD TO ROWCOL / PIXEL / MM / CM / INCH.

One pixel is a "dot on the screen", i.e. smallest single component of a digital image. The character size in pixel depends on the used font (see SET FONT and LNG.5.3.1 & 5.3.2) and can be determined by Row2pixel(), Col2pixel() and Strlen2pix(). For example, with SET FONT "Arial",12 the width of letter "X" is 11 pixel, but "i" occupy only 4 pixel; the row height is here 21 pixels, and column stepping is 13 pixels = width of "M" (data depends on the screen resolution, here for WUXGA desktop monitor with resolution 1920x1200 pixel).

The SET PIXEL apply for GUI mode only and is ignored in Terminal and Basic i/o, which both assumes 1 pixel == 1 column or row.

Another alternative to specify coordinate units in mm, cm and inch is by SET UNIT or SET COORD command.

Classification: programming
Compatibility: New in FS5
Related:

Set(_SET_PIXEL), SET FONT, Col2pixel(), Row2pixel(), Pixel2col(), Pixel2row(), StrLen2col(), StrLen2pix(), SET UNIT

SET PRINTER

Syntax 1:

```
SET PRINTER TO [<file>|<device>|(<expC1>)[ADDITIVE]]
SET PRINTER TO [PIPE <expC2>]
```

Syntax 2:

```
SET PRINTER on|OFF|(<expL>) [NEW]
```

Syntax 3:

```
SET PRINTER GUI on|OFF|(<expL>)
```

Purpose:

Echoes console output (e.g. of the ?, ?? commands) to a printer file or device.

Arguments:

TO <**file**> is the name of an ASCII file, path and an extension included, to which the output will be redirected. If the file extension is not specified, .prn is assumed.

ADDITIVE causes the specified printer file to be appended to, instead of being overwritten. When omitted, the specified <file> is truncated. The <file> is created in the SET DEFAULT directory when given, or in the current one otherwise.

TO <**device**> is any valid Unix character device, like /dev/tty05, /dev/lp0 etc. If the <device> name starts with "/dev/", no default .prn extension is added. In MS-Windows, you may specify LPT1, LPT2, LPT3...LPT9, PRN, COM1...COM9 as direct printing device, which of course must be available on your system. In Linux, device names /dev/lp0 and /dev/lp1 etc. corresponds to LPT1 and LPT2 etc in DOS; /dev/ttyS0 and /dev/ttyS1 etc. corresponds to COM1 and COM2 in DOS.

TO PIPE <**expC2**> streams the PRINTER output to be an input of the Unix executable given in <expC2>, similar to the shell invocation e.g. a.xx | b.sh. The <expC2> expression may also contain more complex piping, like "tee out1.txt > out2.txt" which sends the output into both files. Note: the executable given in <expC2> remains active as a child process until SET PRINTER TO with no arguments is executed. Thereafter, the child process has "zombie" status, which means the process slot remains occupied until the current executable ends. Supported in Linux/Unix only.

When SET PRINTER TO is specified without an argument, the default spooler file is selected ADDITIVE.

ON/OFF activates or deactivates the output to the specified file, device, or the default printer file. Alternatively, the parenthesized <expL> may be used, whereby logical TRUE is the same as ON. When PrintGui(.T.) or SET PRINTER GUI ON is active, the ASCII output is additionally redirected to file or device when SET PRINTER is ON.

NEW causes the current printer file contents to be deleted, instead of appended to.

GUI ON/OFF activates or deactivates GUI alike output. SET PRINT GUI is equivalent to SET GUIPRINTER command and PrintGui(.T./.F.) function, see details in FUN.PrintGui().

Description:

Because of the usual multiuser printer sharing on Unix and Windows, FlagShip redirects by default the printer output to a "spooler" file, see LNG.3.4 and LNG.5.1.6. When starting a program, the default printer file is opened in the current (or by the environment variable FSOUTPUT assigned) directory; the SET DEFAULT path does not affect the default spooler file.

The name of the default spooler file is <main_procedure>.<process_id> and can be retrieved by the FS_SET ("printfile") function. The data from a printer file can be printed either from the application, or any time offline using the default Unix/Windows spooler. To spool the printer data directly from the application,

- issue SET PRINTER OFF or SET PRINTER TO
- retrieve the file name <printfile> := FS_SET("print")
- activate the output using e.g. RUN "lp -d... <printfile>" or RUN "cp <printfile> /dev/..." in Linux, or RUN "copy <printfile> LPT2" etc. in Windows etc.,

  a. delete the file using ERASE <printfile>, if the subsequent printer output is not required,

     issue SET PRINTER ON for the subsequent output.

  b. issue SET PRINTER ON NEW for the subsequent output, which deletes the previous output.

In special cases, if a spooled output is not required, even direct device (such as printer, other terminal etc.) output is supported by FlagShip using SET PRINTER TO <device>.

The SET PRINTER ON command is equivalent to the ...TO PRINTER clause of console commands like LIST, REPORT etc. To suppress the console output, SET CONSOLE OFF may be used. To redirect the @..SAY command to the printer file or device, issue a SET DEVICE TO PRINTER.

Printer output from **GUI based application** can be done w/o any programming activities nearly automatically via the "File->Print ..." menu item, see additional description in LNG.5.1.6 to 5.1.8 and examples in <FlagShip_dir>/examples/printer.prg and printergui.prg

To print graphically in GUI mode, you may use **PrintGui()** function instead, which also supports drawing, different fonts and selecting available printer via common printer dialog, see above examples.

Tuning:

You may tune the printer driver by FS_SET("prset") which may be advantageous when using proportional character set etc. Note that some printers requires CR + LF (= carriage return + line feed) for line break instead of LF (line feed) sent by default. In such a case add the statement

```
FS_SET("prset", { chr(13)+chr(10) } )
```

before your printer output statements.

Example 1:

```
SET PRINTER ON
SET CONSOLE OFF
? "This is a printer output only"
SET CONSOLE ON
? "This goes to both the printer and the screen"
SET PRINTER OFF

USE stock
REPORT FORM invent TO PRINT NOCONSOLE
#ifdef FS_WIN32
   RUN ("COPY " + FS_SET("print") + " LPT2")    // Windows
#else
   RUN ("lp -dlaser -m -s " + FS_SET("print"))  // Linux
#endif
```

Example 2:

Create printout spool file, then send it to printer

```
SET PRINTER ON
SET CONSOLE OFF
SET DEVICE TO PRINT

? "Hallo world"
for ii := 3 to 20
   @ ii,10 say "Line " + ltrim(ii)
next
eject

SET DEVICE TO SCREEN
SET CONSOLE ON
SET PRINTER OFF

prFile := fs_set("printfile")
? "Printing Spool-File", prFile
// _aGlobSetting[GSET_L_RUNDISPLAY] := .T.   // optional
#ifdef FS_WIN32
  RUN("COPY " + prFile + " PRN")     // Windows default printer
#else
  RUN ("cp " + prFile + " /dev/lp0") // Linux default printer (aka
LPT1:)
#endif
wait
```

Example 3:

Print directly to specified device

```
#ifdef FS_WIN32
  SET PRINTER TO LPT1              // or COM1 or PRN etc.
#else
  SET PRINTER TO /dev/lp0          // or /dev/lp1 etc.
#endif
SET PRINTER ON
SET CONSOLE OFF
SET DEVICE TO PRINT
... Printer control codes and output via ?, ??, @...
SET DEVICE TO SCREEN
SET CONSOLE ON
SET PRINTER OFF
SET PRINTER TO
```

Example 4:

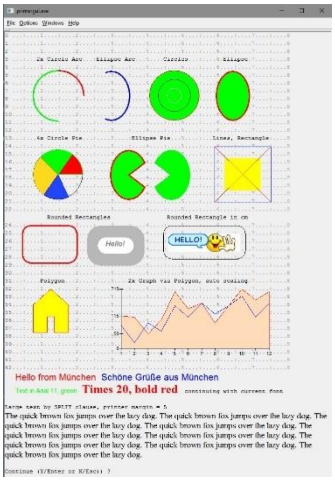
Send the printer output simultaneously to the file "xyz.txt " and the device "tty5c":

```
SET PRINTER TO PIPE "tee xyz.txt > /dev/tty5c"
SET PRINTER ON
? "output to text file and other device"
SET PRINTER (.F.)
? "output to the screen only"
SET PRINTER ON
? "output continued to text file and other device"
SET PRINTER TO
```

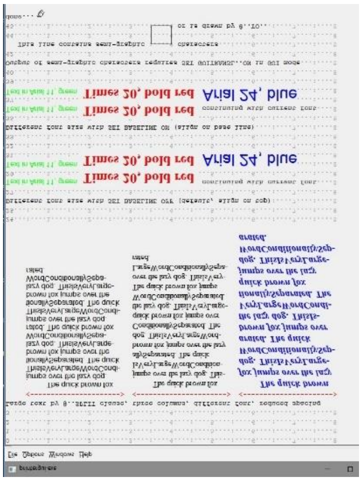
Example 5:

In GUI mode, you may choose the printer driver in a common dialog and have different formatting methods. See/run the complete example in <FlagShip_dir>/examples/printergui.prg (and printer.prg).

Output:



```
Hello from München  Schöne Grüße aus München
Text in Arial 11, green  Times 20, bold red  continuing with current font

Large text by SPLIT clause, printer margin = 5
The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The
quick brown fox jumps over the lazy dog.

Continue (Y/Enter or N/Esc)) ?
```



Example 6:

Printer output via Ethernet: when you want to redirect the local printout (at the Unix server) to a remote printer which has an Ethernet printserver module installed, you simply set lp (or lpr) to the printer IP address. Printer output via Terminal emulator: you may also print remotely via terminal emulator (eg. from MS-Windows 9x/NT) when the emulator support transparent printer redirection eg via VT escape sequences. Here an example, tested with the CRT 2.3 terminal emulator (http://www.vandyke.com) and PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/) from local MS-Windows 32/64bit and a FlagShip application running on remote Unix, Linux or Windows server.

Compile: FlagShip testprint.prg -io=t -o testprint
Execute via terminal emulator: ./testprint or newfsterm ./testprint

```
** file testprint.prg
local i, cSpoolDef, cSpoolTmp, nRow

/* 1. create some printer output into default spool file, you
 *    of course may use any ?, ??, @..SAY etc output instead
 */
printTestData()                 // print some test data to spooler
cSpoolDef := FS_SET("print")    // get default spool file name

/* 2. print the plain spool file to remote printer thru
 *    terminal emulator, using VT100 escape sequences
 *    Note: this may fail with some Terminal Emulators
 */
? "printing file", cSpoolDef, "to remote printer ..."
Send2print(cSpoolDef)
// DELETE FILE (cSpoolDef)

/* 3. switch back to console output
 */
wait
nRow := ROW() +1
for i := nRow to nRow +5
   ? "displaying line #", LTRIM(i), "on terminal"
next
wait "done, press any key..."

/* 4. alternative print to remote printer via shell.
 *    The output file already contains the re-rooting sequences.
 *    Note: this usually work with any terminal emulator which
 *    supports printer redirection via DEC VT escape sequences
 */
?
cSpoolTmp := TempFileName(,"my_") + ".prn"
? "creating another printer output file", cSpoolTmp
printTestData(.T., cSpoolTmp) // create with prefix and postfix

? "copying printer output to remote printer ..."
#ifdef FS_WIN32
  RUN ("type " + cSpoolTmp)   // print in VFS for MS-Windows
```

```
#else
  RUN ("cat " + cSpoolTmp)    // print in VFS for Unix/Linux
#endif
// DELETE FILE (cSpoolTmp)

wait "done, press any key..."
quit

// defines used below
#define REROUTE_ON  chr(27) + "[5i"    /* VT100 sequence */
#define REROUTE_OFF chr(27) + "[4i"    /* VT100 sequence */
#define CRLF        chr(13) + chr(10)
#define MY_STDOUT   2

**********************************************
* print some test data into spool file
* lAddReroute: if .T., add re-rooting prefix and postfix
* cFile: if not empty, use this out file instd.standard
*
FUNCTION PrintTestData(lAddReroute, cFile)
local ii
if valtype(lAddReroute) != "L"
   lAddReroute := .F.
endif

if !empty(cFile)           // specified spooler file
   set printer to (cFile)  // otherwise default spool file
endif
set device to print  // set output to printer
set printer on
set console off

if lAddReroute       // add esc sequence to
   ?? REROUTE_ON     // Start re-routing to remote printer
endif

// send some data to printer
@ 2,0 say "line 2, col 1"
@ 5,5 say "line 5, col 5"
@ 7,25 say "line 7, col 25"
for ii := 1 to 5
   ? ; ?? "printing line#", ltrim(prow())
next
?
if lAddReroute       // add esc sequence to
   ?? REROUTE_OFF    // end re-routing to remote printer
endif

if !empty(cFile)     // close spooler file
   set printer to
endif
set console on       // set output back to console
set printer off
set device to screen
return NIL
```

```
**********************************************
* Sends the given file to remote printer
* adding redirection prefix/postfix esc-sequences
*
FUNCTION Send2print(cFile)
local fh, str, iErr := 0, lAbort := .F.

fh := fopen(cFile,0)
if fh <= MY_STDOUT .or. ferror() != 0
   wait "cannot open printer file '" + cFile + "' ..."
else
   fwrite(MY_STDOUT, REROUTE_ON)   // start redirection
   while iErr == 0
      str  := freadtxt(fh)
      iErr := ferror()
      fwrite(MY_STDOUT, str + CRLF)
      if inkey() == K_ESC
         lAbort := .T.
         exit
      endif
   enddo
   fclose(fh)
   fwrite(MY_STDOUT, CRLF)
   fwrite(MY_STDOUT, REROUTE_OFF)  // end redirection
   if lAbort
      ? "Aborted by user..."
   endif
endif
return NIL
```

Classification:

programming (sequential printer/file output and system file access is used due printing, i.e. executing the output command/function)

Compatibility:

This command is compatible to the xBASE dialects on DOS, but only FlagShip supports the printer spooling by default. If a DOS device name (like LPT2, PRN, COM3) is used in 1:1 ported programs to Linux, make a link to the Unix device before invoking the executable, e.g.:

```
$ ln LPT2.prn /dev/lp1
```

The clauses GUI, PIPE, NEW and ADDITIVE are available in FS only.

Translation:

SET (_SET_PRINTER, .T.|.F.)
SET (_SET_PRINTFILE, "file", .add.)

Related:

EJECT, SET CONSOLE, SET DEVICE, SET ALTERNATE, SET EXTRA, PrintGui(), FS_SET(), (FSC) environment, Unix: man lp, ls -l /dev

SET PROCEDURE TO

Syntax:

`SET PROCEDURE TO [<file>]`

Purpose:

Informs the compiler that all UDFs and procedures in the specified <file> are to be compiled together with the current source file.

Arguments:

<**file**> is the name of the source file. If no extension is specified, the default is .prg. The <file> can optionally include a path designator. SET PROCEDURE TO without an argument has no practical meaning in FlagShip and is simply ignored.

Description:

The SET PROCEDURE statement directs the FlagShip compiler to compile an additional source <file> into C and object file. A file can contain any number of procedures and UDFs. The same occurs, if a procedure call DO... is encountered during the compiling, the name is yet unknown and the same source file exists. If the -m compiler switch is specified, the SET PROCEDURE statement is ignored.

In FlagShip, SET PROCEDURE can be omitted giving the name of the procedure <file> in the command compiler line; see section FSC.

With the -m compiler switch, this command is ignored, i.e. the <file> is not compiled automatically and needs to be added in the compiler command line or in Makefile.

Example:

```
*** file test.prg
SET PROCEDURE TO procfile                && needed for myProc
proc = "myProc"
DO &proc                                 && indirect call by macro
QUIT
*** eof test.prg

*** file procfile.prg                    && not called directly,
PROCEDURE myProc                         && therefore SET PROCEDURE
RETURN                                   && or: FlagShip test.prg \
                                         &&     procfile.prg
FUNCTION other()
return "hello"
*** eof procfile.prg
```

Classification:

compiler/linker

Translation:

PROCREQ ("file")

Related:

DO, FUNCTION, PROCEDURE, SET FORMAT, #include

SET RELATION

Syntax:

```
SET RELATION [ADDITIVE] [MULTIPLE]
     TO [<parentKey1>|<recno> INTO <childAlias1>]
     [, [TO] <parentKey2>|<recno> INTO
        <childAlias2>]
```

Purpose:

Relates two or more working areas using a key expression or record number.

Arguments:

<**parentKey**> is an expression used to perform a SEEK in the child area each time the record pointer moves in the parent working area. This is usually a field of the parent area. The child area must have an index in use, with key expression corresponding to <parentKey> value.

<**recno**> is a record number or an numeric expression (typically the RECNO() function) used to perform a GOTO to the record number in the child working area matching the record number of the parent area. For this type of relation, the child area need not have an index in use or the indices are disabled by SET ORDER TO 0

<**childAlias**> identifies the child working area (or file name). The child database may also be opened by different RDD driver, than the current (parent) database.

SET RELATION TO without arguments removes all relations from the current working area.

Options:

ADDITIVE adds the specified child relations to existing relations already set in the current working area. If this clause is not specified, existing relations in the current working area are released before the new child relations are set. You may determine the number of relations for each parent by DbRelCount().

MULTIPLE specifies that the child database is processed as 1:n relation, otherwise it is 1:1 relation. Only the first relation in each parent is considered as 1:n by SKIP. You may set or clear the 1:n relation also later by DbRelMultiple().

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 (see also Note below to set more than two relations).

A relation causes the record pointer in the child area to move in accordance with the movement of the record pointer in the parent area. If a match is not found, the child area record pointer is positioned to the end-of-file (LASTREC() +1), EOF() returns .T. and FOUND() returns .F.

The typical sequence is SELECT <parent>; SET RELATION TO <parentKey> INTO <childAlias>, where the <childAlias> is indexed on a key matching the <parentKey>, e.g.

```
USE mydb NEW ALIAS master                // has field IdKey
* INDEX ON anything TO master            // optional
USE subdb NEW ALIAS child                // has field IdChild
INDEX ON IdMaster TO subdb               // and field IdMaster
* USE subsub NEW ALIAS childOfChild      // has field IdChild
* INDEX ON IdChild TO subsubdb
...
SELECT master
SET RELATION TO IdKey INTO child             // opt: MULTIPLE
* SELECT child
* SET RELATION TO IdChild INTO childOfChild  // opt: MULTIPLE
SELECT master
// LIST IdKey, child->IdChild, child->IdMaster, child->Data, ;
//      childOfChild->IdChild, childOfChild->data
// other processing ...
SELECT master
SET RELATION TO                          // clear relation
* SELECT child
* SET RELATION TO                        // clear relation
```

If the MULTIPLE clause is specified, SKIP process the child in 1:n manner instead of 1:1. This means, if the child contains more than one corresponding key for this relation, the child is skipped instead of parent. For 1:n:n:..:n relations, the last child is skipped first (as long as in the relation), then the last-1 child with all it childs in relation, and so forth. See also LNG.4.7 and example below.

Although SET RELATION obeys SET FILTER and SET DELETED in the child working areas, it does not obey SET SOFTSEEK, thus always behaving as if SET SOFTSEEK were off. In most cases, conditional index (INDEX ON..FOR..) is also faster then SET FILTER.

Although the SET RELATION is a comfortable database link, it may slow the execution significantly; especially if the movement in the child area(s) is not needed for each movement (SEEK, SKIP, GOTO, REPLACE...FOR etc.) in the parent area. In such a case, use a "soft link", SEEKing the child record explicitly when required only. For 1:N:N relations, you will need (in worst case) to skip (records-in-parent * records-in-child1 * records-in-child2) -times to reach eof().

RELATing a database directly or indirectly to itself will usually result with unpredictable results, possibly endless loops!

FlagShip tries to keep the relation integrity by repositioning the dependent database, e.g. on movements, SELECT, by reaching break in the GUI debugger. You may force the integrity by issuing SKIP 0 on the parent database. A manual SEEK is hence suggested when you will manipulate the record pointer of the related database.

Note: with the SET RELATION command, you can set one or two relations at a time. When more relations are required for the same parent, use subsequent settings

```
   SET RELATION TO KeyChild1 INTO alias1, TO KeyChild2 INTO alias2
   SET RELATION ADDITIVE TO KeyChild3 INTO alias3
   SET RELATION ADDITIVE TO KeyChild4 INTO alias4  //...etc...
```

or use multiple DbSetRelation() which does not have any restriction for the number of relations.

Example 1:

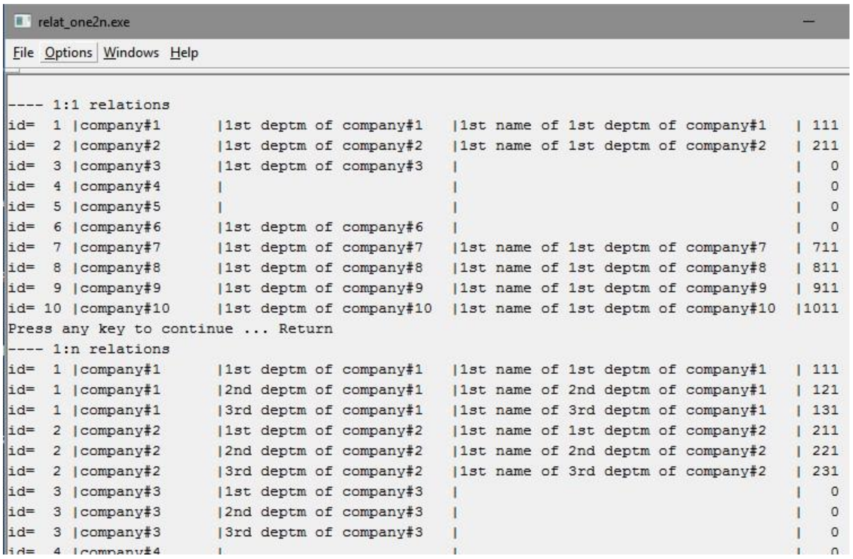
```
SELECT 2                                    // child relat.
USE employee
INDEX ON emplidno TO empl_id
SELECT 1                                    // parent relat.
USE families
? FIELD(3)                                  // ID_NUMB
SET RELATION TO families->id_numb INTO employee
LIST Name, Employee->Name, Employee->Lastname

// The same output using a manual "soft" 1:1 relation:

USE employee NEW INDEX empl_id              // child
USE families NEW                            // parent
DO WHILE !EOF()
   employee->(DBSEEK (families->id_numb))   // SEEK in child
   ? Name, Employee->Name, Employee->Lastname
   SKIP
ENDDO
```

Example 2:

```
* the complete code is available in .../examples/relat_one2n.prg
select company
// index on ID to company_id     // not required
select departm
index on IDcomp to depart_id     // required
select names
index on IDdepart to names_id    // required
/*
 * set 1:n:n relations
 */
select company
set relation to ID into departm MULTI     // company -> departm 1:n
select departm
set relation to IDdepart into names MULTI // departm -> names 1:n
select company
set filter to company->zip >= 2000 .and. company->zip < 3000
// better: INDEX ON zip FOR zip >= 2000 .and. zip < 3000 TO zip2
go top
while !eof()
   ? "id="+str(id,4),"|"+company+"|"+departm->deptm+"|"+ ;
     names->name+"|"+names->first+"|"+str(names->IDpers,5)
   SKIP
enddo
select departm
set relation to     // clear relation(s) from departm to child(s)
select company
set relation to     // clear relation(s) from company to child(s)
set filter to       // clear filter
wait

// the same 1:n:n output without relations:
```

```
// use ... index ... from above
select company
set filter to company->zip >= 2000 .and. company->zip < 3000
go top
while !eof()                                   // on company
   SELECT departm
   SEEK company->ID
   if eof()
      ? "id=" + str(company->id,4)+ "|" + company->company + ;
        "|no departments"
   endif
   while !eof() .and. IDcomp == company->ID    // on departm
      SELECT names
      SEEK departm->IDdepart
      if eof()
         ? "id=" + str(company->id,4)+ "|" + company->company + ;
            "|" + departm->deptm + "|no names"
      endif
      while !eof() .and. IDdepart == departm->IDdepart  // on names
         ? "id=" + str(company->id,4)+ "|" + company->company + ;
            "|" + departm->deptm + "|" + names->name + "|" + ;
            names->first + "|pers.id=" + str(names->IDpers,5)
         SKIP    // next name for the same IDdepart
      enddo      // while... on names
      SELECT departm
      SKIP       // next departm for the same IDcomp
   enddo         // while... on departm
   SELECT company
   SKIP          // next company
enddo            // while.. on company
wait
```

Output:



```
---- 1:1 relations
id=  1 |company#1     |1st deptm of company#1   |1st name of 1st deptm of company#1   | 111
id=  2 |company#2     |1st deptm of company#2   |1st name of 1st deptm of company#2   | 211
id=  3 |company#3     |1st deptm of company#3   |                                     |   0
id=  4 |company#4     |                         |                                     |   0
id=  5 |company#5     |                         |                                     |   0
id=  6 |company#6     |1st deptm of company#6   |                                     |   0
id=  7 |company#7     |1st deptm of company#7   |1st name of 1st deptm of company#7   | 711
id=  8 |company#8     |1st deptm of company#8   |1st name of 1st deptm of company#8   | 811
id=  9 |company#9     |1st deptm of company#9   |1st name of 1st deptm of company#9   | 911
id= 10 |company#10    |1st deptm of company#10  |1st name of 1st deptm of company#10  |1011
Press any key to continue ... Return
---- 1:n relations
id=  1 |company#1     |1st deptm of company#1   |1st name of 1st deptm of company#1   | 111
id=  1 |company#1     |2nd deptm of company#1   |1st name of 2nd deptm of company#1   | 121
id=  1 |company#1     |3rd deptm of company#1   |1st name of 3rd deptm of company#1   | 131
id=  2 |company#2     |1st deptm of company#2   |1st name of 1st deptm of company#2   | 211
id=  2 |company#2     |2nd deptm of company#2   |1st name of 2nd deptm of company#2   | 221
id=  2 |company#2     |3rd deptm of company#2   |1st name of 3rd deptm of company#2   | 231
id=  3 |company#3     |1st deptm of company#3   |                                     |   0
id=  3 |company#3     |2nd deptm of company#3   |                                     |   0
id=  3 |company#3     |3rd deptm of company#3   |                                     |   0
id=  4 |company#4     |                         |                                     |   0
```

Classification:

database

Compatibility:

The ADDITIVE clause is new in FS4. For FS4 and before, the number of child areas was restricted to 8, VFS5 and later supports any number of relations. 1:N relations are available in VFS7 and later.

Translation:

```
IF ( ! .add. ) ; DbClearRel() ; ENDIF
DbSetRelation ( "alias1", {key1}, "key1" [, .multi.] )
[ DbSetRelation( "alias2", {key2}, "key2" [, .multi.] ) ...]
```

Related:

INDEX, SET INDEX, SET ORDER, UPDATE, USE, SEEK, SKIP, GOTO, REPLACE, Recno(), DbSetRelation(), DbRelation(), DbClearRel(), DbRselect(), DbRelCount(), DbRelMultiple(), oRdd:SetRelation(), oRdd:ClearRelation(), oRdd:Info()

SET ROWADAPT

Syntax:

```
SET ROWADAPT on|OFF|(<expL>)
```

Purpose:

Enables or disables the automatic ROW() adaption when the screen output includes HTML tags or different FONTs.

Arguments:

ON/OFF enables/disables the automatic ROW() adaption. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

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 <BR> 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.

The final cursor position may be affected also by the current status of SET ROWALIGN, see examples there.

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 in GUI mode

Compatibility:

New in FS5

Translation:

```
Set( _SET_ROWADAPT [, <expL>] )
```

Related:

?, ??, @..SAY, SET ROWALIGN, SET HTMLTEXT, SET FONT, Qout(), Qqout(), RowAdapt()

SET ROWALIGN

Syntax:

```
SET ROWALIGN [TO] BASELINE|DEFAULT
SET ROWALIGN TO
SET ROWALIGN (<expL>)
```

Purpose:

Enables or disables the automatic ROW() alignment on baseline of the standard font.

Arguments:

BASELINE enables the automatic ROW() alignment.

DEFAULT disables the automatic ROW() alignment.

<expL> is an alternative syntax, whereby (.T.) is same as SET ROWALIGN TO BASELINE and (.F.) is same as SET ROWALIGN TO DEFAULT.

SET ROWALIGN TO is equivalent to SET ROWALIGN TO DEFAULT

Description:

The default x/y alignment in GUI mode is on the top left character frame (marked with + in the picture below), to allow start the output at 0,0 coordinates. The characters "O-umlaut","h","p" are displayed as

```
--+------------------------------ <- top character frame
  |  *   *   |       |        |
  |   ###    | #     |        |
  |  #   #   | #     |        |
  | #     #  | ###   | ####   |
  |  #   #   | #  #  | #   #  |
  |   ###   -| #  # -| ####  -|   <- base line
  |          |       | #      |
  |          |       | #      |
  -----------------------------   <- bottom character frame
--------------------------------- <- line spacing
```

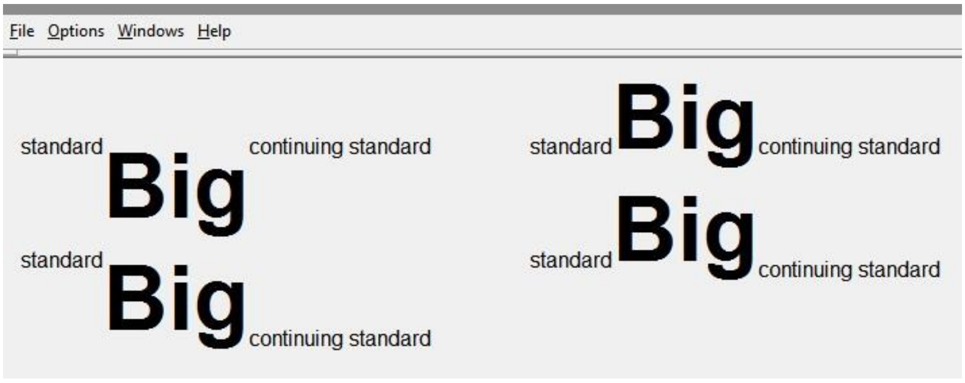
where the size of (bottom - top) is returned by oFont:Height() or in pixel by oFont:SizePixel(); the line spacing is user definable by global variable _aGlobSetting[GSET_G_N_ROW_SPACING].

When you change the FONT size, the start position remain unchanged, i.e. larger font has it base line located at higher Y position (in view of top/down coordinates). Sometimes you may wish to align characters on it base line, e.g. when using the FONT option to display different fonts (from the standard) in the same output line similarly to word processor output, e.g.

```
oFont2 := Font{"Arial",50} ; oFont2:Bold := .T.
// SetPos(4,10)
?? "Start "
?? "Big" FONT oFont2 GUICOLOR "R+"
?? " Continue" GUICOLOR "B+"
```

Depending on SET ROWALIGN and SET ROWADAPT setting, you will get:

```
with  SET ROWALIGN TO DEFAULT  | with  SET ROWALIGN TO BASELINE
 and  SET ROWADAPT OFF         |  and  SET ROWADAPT OFF
                               |
Start  BBBB         Continue   |       BBBB
       B   B  *                |       B   B  *
       BBBBB  i  ggg           |       BBBBB  i  ggg
       B   B  i g  g           |       B   B  i g  g
       BBBB   i  ggg           | Start BBBB   i  ggg  Continue
                   g           |                   g
                 gg            |                 gg


with  SET ROWALIGN TO DEFAULT  | with  SET ROWALIGN TO BASELINE
 and  SET ROWADAPT ON          |  and  SET ROWADAPT ON
                               |
Start  BBBB                    |       BBBB
       B   B  *                |       B   B  *
       BBBBB  i  ggg           |       BBBBB  i  ggg
       B   B  i g  g           |       B   B  i g  g
       BBBB   i  ggg           | Start BBBB   i  ggg
                   g           |                   g
                 gg            |                 gg
                    Continue   |                    Continue
```



Note that the base line alignment for larger font size than current default will display desired results only when the current Row() is > 0, since you cannot print on negative coordinates :-)

Instead of using SET ROWALIGN TO BASELINE, you of course may control the alignment and coordinates manually by SetPos(), e.g.

```
oFont2 := Font{"Helvetica",60} ; oFont2:Bold := .T.
?? "Start "
ySave := Row(.T.)
yNew  := max(0, ySave - oFont2:SizePixel() + ;
               m->oApplic:Font:SizePixel() )
SetPos(yNew,  Col(.T.), .T.) ; ?? "Big" FONT oFont2
SetPos(ySave, Col(.T.), .T.) ; ?? " Continue"
```

The SET ROWALIGN alignment 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 in GUI mode

Compatibility:

New in FS5

Translation:

```
Set( _SET_ROWALIGN_BASE [, <expL>] )
```

Related:

?, ??, @..SAY, SET ROWADAPT, SET HTMLTEXT, SET FONT, Qout(), Qqout(), RowAdapt()

SET SCRCOMPRESS

Syntax:

```
SET SCRCOMPRESS on|OFF|(<expL>)
```

Purpose:

Enable/disable compressing screen image for SaveScreen() and RestScreen() in GUI mode.

Arguments:

ON/OFF enables/disables the compressing of screen images in GUI mode. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON. The default setting is OFF.

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 as opposite to Curses "window" structure in Terminal i/o.

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.

The SET SCRCOMPRESS setting apply for GUI mode only and is ignored in Terminal and Basic i/o.

Classification:

programming

Compatibility:

New in FS5

Related:

Set(_SET_SCRCOMPRESS), SaveScreen(), RestScreen()

SET SCOREBOARD

Syntax:

`SET SCOREBOARD ON|off|(<expL>)`

Purpose:

Defines whether the messages issued by READ and MEMOEDIT() are displayed or not.

Arguments:

ON/OFF enables/disables the display of messages on line zero. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

When SCOREBOARD is ON, the messages are displayed at the uppermost line of the display. The messages are: the indication of the INSERT mode, the RANGE error message, and an abort query message in MEMOEDIT().

In FlagShip, the message text can be user-specified, for example for foreign languages, using FS_SET ("loadlang") and FS_SET("setlang"). The query messages for MEMOEDIT() are re-definable in the file <FlagShip_dir>/system/scor_mem.prg.

Example:

```
SET SCOREBOARD OFF
SET FORMAT TO authors
USE authors
READ
SET FORMAT TO
USE
SET SCOREBOARD ON
```

Classification:

programming

Compatibility:

The user definable messages are available in FlagShip only.

Translation:

SET (_SET_SCOREBOARD, .T. | .F.)

Related:

@...GET, READ, MEMOEDIT(), FS_SET(), SET()

SET SOFTSEEK

Syntax:

`SET SOFTSEEK on|OFF|(<expL>)`

Purpose:

Defines whether SEEK and FIND will have softer criteria in searching or whether they will be strict.

Arguments:

ON/OFF enables/disables soft searching. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

When SOFTSEEK is ON, and a matching index key is not found while SEEKing, the first higher value key is reported as found and the record pointer is located on it. If SOFTSEEK is OFF, only the exact match will be found.

SEEK with SOFTSEEK ON	FOUND()	EOF()
Index key found	.T.	.F.
Next index key found	.F.	.F.
Next index key not available	.F.	.T.

SEEK with SOFTSEEK OFF	FOUND()	EOF()
Index key found	.T.	.F.
Index key not found	.F.	.T.

The current state of SET FILTER and SET DELETED is obeyed in SEEK, regardless of the SOFTSEEK setting.

Note: SEEKing the child record of a SET RELATION specified database ignores the current SET SOFTSEEK switch.

Example:

```
LOCAL searchname
SET SOFTSEEK ON
USE address INDEX adrname
DO WHILE .T.
   ACCEPT "enter name to search (or <┘) " TO searchname
   IF EMPTY(searchname)
      EXIT
   ENDIF
   SEEK searchname
   DO CASE
   CASE FOUND()
      ? "Found: ", name, first
   CASE .not. EOF()
      ? "Next name found: ", name, first
   OTHERWISE
      ? "The same or next name not available"
   ENDCASE
ENDDO
```

Classification:

database

Translation:

SET (_SET_SOFTSEEK, .T. | .F.)

Related:

SEEK, FIND, SET INDEX, SET ORDER, SET RELATION, EOF(), FOUND(), SET(), oRdd:Seek()

SET SOURCE

Syntax:

SET SOURCE ASCII | PC8 | OEM
SET SOURCE ISO | ANSI

Purpose:

Enable support of national character sets, i.e. source strings containing special characters coded in either in PC8/ASCII or in ISO/Ansi. For a full internationalization discussion refer to section LNG.5.4

Description:

To support national character sets coded in ASCII (e.g. by using DOS source editor) in GUI i/o mode (which is usually ISO oriented) and for printer output, an automatic ASCII -> ISO conversion during the output is available via SET SOURCE ASCII. This is very similar to converting the string output via Oem2Ansi() like

```
  Qout(Oem2Ansi("M"+chr(129)+"nchen")) // u-umlaut in PC-8
```

Note: SET SOURCE ASCII is issued in #include "fspreset.fh" to enable an easy port of DOS applications.

To support national character sets coded in ISO/Ansi in GUI mode, you may inform the system that no automatic ASCII -> ISO conversion during the output is required using SET SOURCE ISO. This is very similar to the string output Qout("M"+chr(252)+"nchen") // u-umlaut in ISO-8859-1

In Terminal i/o mode, the input and output is assumed to be in ASCII, i.e. u-umlaut as chr(129), same as in DOS and Clipper. If you are using an ISO or Windows source-code editor (which will code chr(252) for u-umlaut), you may preferably use the -iso compiler switch, which will translate ANSI/ISO strings to ASCII.

The #include "fspreset.fh" statement (see LNG.9.5) sets SET SOURCE ASCII and SET GUITRANSL TEXT ON to enable backward compatibility to DOS and terminal i/o oriented source.

SET SOURCE ISO is not equivalent to set(_SET_SOURCEASCII). The SET SOURCE command sets also additional flags for your convenience, whilst Set(_SET_-SOURCEASCII) only reports if SET SOURCE was invoked. See also the #command SET SOURCE in std.fh for details.

SET SOURCE ASCII (or PC8 or OEM) will set

```
   Set(_SET_SOURCEASCII, .T.)   // source is in ASCII charset
   Set(_SET_GUIASCII,    .T.)   // translate screen output
   Set(_SET_ANSI,        .F.)   // read/write database 1:1
   Set(_SET_CHARSET, _SET_CHARSET_PC8) // translate Inkey()
   Set(_SET_PRINTASCII,  .F.)   // printer output 1:1
```

SET SOURCE ANSI (or ISO) will set

```
   Set(_SET_SOURCEASCII, .F.)   // source is in ISO charset
   Set(_SET_GUIASCII,    .F.)   // GUI screen output 1:1
   Set(_SET_ANSI,        .T.)   // translate read/write database
   Set(_SET_CHARSET, _SET_CHARSET_ISO)  // Inkey() is 1:1
   Set(_SET_PRINTASCII,  .T.)   // translate printer output
```

Note: The automatic support of IBM-PC8/ASCII semi-graphic characters in GUI mode is enabled when SET GUITRANSL TEXT ON is set. Without changing the translation tables heavily - see fs_set("ansi2oem"), the semi-graphic output work properly only with SET SOURCE ASCII, since since glyphs of chr(176..223) equivalence are not available in the ISO/ANSI character set.

In Terminal i/o mode, the IBM-PC8/ASCII semi-graphic characters support is enabled per default (with SET SOURCE ASCII). To display glyphs with current SET SOURCE ISO, enable temporarily SET SOURCE ASCII before the output of chr(176..223).

Note other GUI defaults, modified by SET GUITRANSL command:

```
Set(_SET_GUIDRAWTEXT)  = .F. // don't draw PC-8 charset
Set(_SET_GUIDRAWBOX)   = .F. // don't draw PC-8 boxes
Set(_SET_GUIDRAWLINES) = .F. // don't draw PC-8 lines
```

Classification:

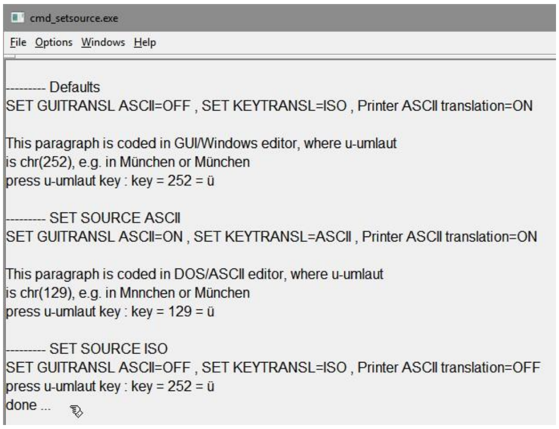
programming, screen and printer output

Example 1:

see <FlagShip_dir>/examples/setsource.prg, printer.prg and printergui.prg

Example 2:

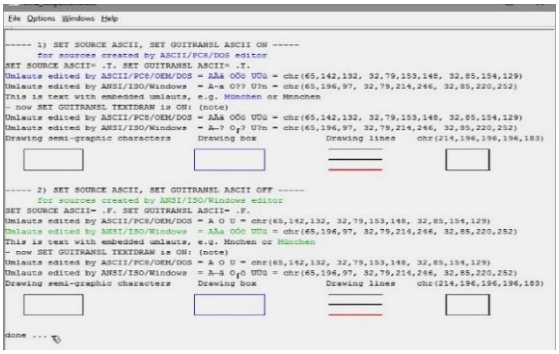
```
? "--------- Defaults"
? "SET GUITRANSL ASCII=" + if(set(_SET_GUIASCII),"ON","OFF"), ;
  ", SET KEYTRANSL=" + if(set(_SET_CHARSET) == _SET_CHARSET_PC8, ;
  "ASCII", "ISO"), ", Printer ASCII translation=" + ;
  if(set(_SET_SOURCEASCII), "ON", "OFF")
?
? "This paragraph is coded in GUI/Windows editor, where u-umlaut"
? "is chr(252), e.g. in München or M" + chr(252) + "nchen"
? "press u-umlaut key :"
key := inkey(0)
?? " key =", ltrim(key),"=", chr(key)  // key = 252 in ISO-8859-1

?
SET SOURCE ASCII
? "--------- SET SOURCE ASCII"
? "SET GUITRANSL ASCII=" + if(set(_SET_GUIASCII),"ON","OFF"), ;
  ", SET KEYTRANSL=" + if(set(_SET_CHARSET) == _SET_CHARSET_PC8, ;
  "ASCII", "ISO"), ", Printer ASCII translation=" + ;
  if(set(_SET_SOURCEASCII), "ON", "OFF")
?
? "This paragraph is coded in DOS/ASCII editor, where u-umlaut"
? "is chr(129), e.g. in München or M" + chr(129) + "nchen"
? "press u-umlaut key :"
key := inkey(0)
?? " key =", ltrim(key),"=", chr(key) // key = 129
?

SET SOURCE ISO
? "--------- SET SOURCE ISO"
? "SET GUITRANSL ASCII=" + if(set(_SET_GUIASCII),"ON","OFF"), ;
  ", SET KEYTRANSL=" + if(set(_SET_CHARSET) == _SET_CHARSET_PC8, ;
  "ASCII", "ISO"), ", Printer ASCII translation=" + ;
  if(set(_SET_SOURCEASCII), "ON", "OFF")
? "press u-umlaut key :"
key := inkey(0)
?? " key =", ltrim(key),"=", chr(key) // key = 252
wait "done ..."
```



```
--------- Defaults
SET GUITRANSL ASCII=OFF , SET KEYTRANSL=ISO , Printer ASCII translation=ON

This paragraph is coded in GUI/Windows editor, where u-umlaut
is chr(252), e.g. in München or München
press u-umlaut key : key = 252 = ü

--------- SET SOURCE ASCII
SET GUITRANSL ASCII=ON , SET KEYTRANSL=ASCII , Printer ASCII translation=ON

This paragraph is coded in DOS/ASCII editor, where u-umlaut
is chr(129), e.g. in Mnnchen or München
press u-umlaut key : key = 129 = ü

--------- SET SOURCE ISO
SET GUITRANSL ASCII=OFF , SET KEYTRANSL=ISO , Printer ASCII translation=OFF
press u-umlaut key : key = 252 = ü
done ...
```

Example 3:

see example 1 in SET GUITRANSL :



```
----- 1) SET SOURCE ASCII, SET GUITRANSL ASCII ON -----
      for sources created by ASCII/PC8/DOS editor
SET SOURCE ASCII= .T. SET GUITRANSL ASCII= .T.
Umlauts edited by ASCII/PC8/OEM/DOS = AÄä OÖö UÜü = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = A-a O?? U?n = chr(65,196,97, 32,79,214,246, 32,85,220,252)
This is text with embedded umlauts, e.g. München or Mnnchen
- now SET GUITRANSL TEXTDRAW is ON: (note)
Umlauts edited by ASCII/PC8/OEM/DOS = AÄä OÖö UÜü = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = A-? O┌? U?n = chr(65,196,97, 32,79,214,246, 32,85,220,252)
Drawing semi-graphic characters     Drawing box            Drawing lines    chr(214,196,196,196,183)

----- 2) SET SOURCE ASCII, SET GUITRANSL ASCII OFF -----
      for sources created by ANSI/ISO/Windows editor
SET SOURCE ASCII= .F. SET GUITRANSL ASCII= .F.
Umlauts edited by ASCII/PC8/OEM/DOS = A O U = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = AÄä OÖö UÜü = chr(65,196,97, 32,79,214,246, 32,85,220,252)
This is text with embedded umlauts, e.g. Mnchen or München
- now SET GUITRANSL TEXTDRAW is ON: (note)
Umlauts edited by ASCII/PC8/OEM/DOS = A O U = chr(65,142,132, 32,79,153,148, 32,85,154,129)
Umlauts edited by ANSI/ISO/Windows  = A-ä O┌ö UÜü = chr(65,196,97, 32,79,214,246, 32,85,220,252)
Drawing semi-graphic characters     Drawing box            Drawing lines    chr(214,196,196,196,183)

done ...
```

Translation:

SET SOURCE ASCII or PC8 or OEM
== Set(_SET_SOURCEASCII, .T.) + Set(_SET_GUIASCII, .T.) +
Set(_SET_CHARSET, _SET_CHARSET_PC8)
SET SOURCE ISO or ANSI
== Set(_SET_SOURCEASCII, .F.) + Set(_SET_GUIASCII, .F.) +
Set(_SET_CHARSET, _SET_CHARSET_ISO)

Compatibility:

New in FS5

Related:

SET GUITRANSL, SET ANSI, Set(_SET_GUIDRAWTEXT), Set(_SET_GUIDRAWBOX), Set(_SET_GUIDRAWLINE), Set(_SET_GUITASCII)

SET TYPEAHEAD TO

Syntax:

`SET TYPEAHEAD TO <expN>`

Purpose:

Sets the size of the keyboard buffer.

Arguments:

<**expN**> is the number of characters that the keyboard buffer can hold. It is an integer in the range from zero up to 2 GB. If not specified, or if a negative value is specified, the buffer is set to default of 10000 bytes.

Description:

FlagShip stores user key stokes in an internal type-ahead buffer, which enables to pre-enter input; see more in chapter LNG.5.2.

If the keyboard buffer's length is set to zero, keyboard polling is suspended and NEXTKEY() will always return zero. The LASTKEY() values are not affected by SET TYPEAHEAD.

The SET TYPEAHEAD buffer size does not affect the number of characters that can be pushed in by a program using the KEYBOARD command.

If you wish to copy-and-paste large text (e.g. to MemoEdit), you may need to increase the TYPEAHEAD buffer accordingly. The default size is sufficient for ca. 2 pages of fully printed paper sheets.

Example:

```
? "working, please do not disturb..."
SET TYPEAHEAD TO 0
USE accounts
COUNT FOR turnover = 0 TO zero
SET TYPEAHEAD TO
? zero, "customers with no turnover"
```

Classification:

programming

Compatibility:

On function keys F2 to F48, 2 bytes for each keystroke are needed. In DOS dialects, the buffer length is limited to 4KB.

Translation:

SET (_SET_TYPEAHEAD, expN)

Related:

ACCEPT, INPUT, KEYBOARD, READ, SET KEY, INKEY(), LASTKEY(), NEXTKEY(), SET()

SET UNIT

Syntax:

```
SET UNIT [TO]
SET UNIT [TO] ROWCOL | PIXEL | MM | CM | INCH |
        ( <expN> )
```

Syntax:

```
SET COORDINATE [UNIT] [TO]
SET COORDINATE [UNIT] [TO] PIXEL | MM | CM | INCH |
        ( <expN> )
```

Purpose:

Sets the unit for subsequently given screen (and printer with active PrintGui() output) coordinates. Applicable in GUI mode only.

Arguments:

ROWCOL :	all subsequent coordinates are in common rows and columns.
PIXEL :	all subsequent coordinates are in pixels
MM :	all subsequent coordinates are millimeter
CM :	all subsequent coordinates are centimeter (ea 10 mm)
INCH :	all subsequent coordinates are in inch (ea 25.4 mm)
<expN> :	parenthesized numeric value, e.g. UNIT_ROWCOL, UNIT_MM, UNIT_CM, UNIT_INCH, UNIT_PIXEL, UNIT_DOTS (specified in the set.fh include file)

Description:

In GUI mode, all widgets (or controls in MS-Windows terminology) are pixel based. To enable the common Xbase (FlagShip, Clipper, FoxPro etc) compatibility, FlagShip internally recalculates the col/row coordinates to pixels according to the used font. The mm, cm, and inch coordinates are re-calculated according to the screen resolution and size, returned (or set) by oApplic:DesktopHeight() and oApplic: DesktopWidth().

SET UNIT TO PIXEL is equivalent to SET PIXEL ON, SET UNIT TO ROWCOL is equivalent to SET PIXEL OFF.

Classification:

programming, screen and printer coordinates

Compatibility:

Available in VFS7 and later only.

Translation:

```
SET ( _SET_COORD_UNIT, expN)
```

Related:

SET PIXEL, SET()

SET UNIQUE

Syntax:

`SET UNIQUE on|OFF|(<expL>)`

Purpose:

Defines whether only unique keys will be included while indexing or not.

Arguments:

ON/OFF enables/disables the UNIQUE indexing. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

When UNIQUE is ON, and a new index is created with INDEX ON...TO, only unique keys are included in the index file, ignoring all subsequent keys of the same values. This is the same as creating an index with the INDEX...UNIQUE command.

Since the UNIQUE setting is stored in the index header, the index retains uniqueness regardless of the UNIQUE settings at later REPLACE, REINDEX, PACK or other database operations.

If a unique key is changed to a value of a key already in the index, the changed record is lost from the index. If there is more than one instance of a key value in a database file, changing the visible key value does not bring forward another record with the same key until the index is rebuilt with REINDEX, PACK, or INDEX...UNIQUE.

Example:

List all magazine names from a large article database:

```
SET UNIQUE ON
USE article
INDEX ON Magazine TO magname           && UNIQUE
LIST magazine

SET UNIQUE OFF
REINDEX                                && remains UNIQUE
INDEX ON Magazine TO magnames          && not UNIQUE
```

Classification:

database

Translation:

SET (_SET_UNIQUE, .T. | .F.)

Related:

FIND, INDEX, REINDEX, SEEK, SET INDEX, USE, SET(), oRdd:OrderIsUnique(), oRdd:CreateIndex(), oRdd:CreateOrder()

SET WRAP

Syntax:

`SET WRAP on|OFF|(<expL>)`

Purpose:

Toggles wrapping of the light bar in MENU TO.

Arguments:

ON/OFF enables/disables the wrapping. Alternatively, the parenthesized <expL> may be used, whereby TRUE is the same as ON.

Description:

Wrapping means that when the light bar is at the last option in MENU and the down-arrow or right-arrow key is pressed, the lightbar moves to the first choice; if the light-bar is at the first choice and up-arrow or left-arrow key is pressed, the light-bar moves to the last choice of the MENU.

When WRAP is OFF, pressing up-arrow or left-arrow at the first menu item or down-arrow or right-arrow at the last menu item does nothing.

Example:

```
@ 10,20 PROMPT "First item"
@ 11,20 PROMPT "Second item"
SET WRAP ON
MENU TO choice
SET WRAP OFF
```

Classification:

programming

Translation:

SET (_SET_WRAP, .T. | .F.)

Related:

@...PROMPT, MENU TO

SETSTANDARD SETENHANCED SETUNSELECTED

Syntax:

`SETSTANDARD`

Syntax:

`SETENHANCED`

Syntax:

`SETUNSELECTED`

Purpose:

Selects the required color attribute for screen output.

Description:

The color set with SET COLOR or SETCOLOR() can include three different color pairs: the "standard", used in all screen output statements, the "enhanced" used in @..GET, READ, MENU, ACHOICE and "unselected", used in the READ command.

In screen output commands (such as @...SAY, @...BOX, ?, QOUT() etc.), only the "standard" color pair will be used. The SETENHANCED and SETUNSELECTED commands switch the corresponding color pair to become the "standard" one, SETSTANDARD resets the original state.

Example:

```
Simulates the READ output:

SET COLOR TO "W+/B,R+/BG,,,GR+/BG"
@ 1, 2 say "Name, First"
SETENHANCED
@ 1,20 say "Smith    "
SETUNSELECTED
@ 1,40 say "Peter    "
SETSTANDARD
```

Classification:

programming

Translation:

_SETSTANDARD( ) | _SETENHANCED( ) | _SETUNSELECTED( )

Related:

SET COLOR, SETCOLOR(), READ, @..SAY, ?, ??

SET ZEROBYTEOUT

Syntax:

`SET ZEROBYTEOUT [TO] <expC>`

Purpose:

Set the output char for \0 byte in [q]qout() if fs_set("zero") is active.

Arguments:

<expC> is the character displayed instead of \0 (binary zero) by ? and ?? commands or Qout() and Qqout() functions. The default setting is chr(63) = "?"

Description:

The embedded zero byte in strings cannot be displayed at all. To be able to see this character in the output, the \0 byte is replaced by this substitute during the console output when FS_SET("zerobyte") is set .T.

SET ZEROBYTEOUT is considered also by SET OUTMODE which specify how to display other unprintable characters < 32

Classification:

programming

Compatibility:

New in FS5

Related:

Set(_SET_ZEROBYTEOUT), FS_SET("zero"), ?, ??, Qout(), Qqout(), SET CONSOLE

SKIP

Syntax:

```
SKIP <expN1> [ALIAS <alias>|(<expN2>)]
```

Purpose:

Moves the record pointer in the specified working area relative to the current pointer position.

Arguments:

<**expN1**> specifies the number of records to move the record pointer from the current position. A positive value moves the pointer forwards, while a negative value moves the pointer backwards. SKIP 0 flushes the current working area buffers, equivalent to DBCOMMIT() or similar to COMMIT.

SKIP without an argument moves the record pointer to the next record in the current working area, having the same effect as SKIP 1.

Options:

ALIAS causes movement of the record pointer in the designated working area specified by the <**alias**> name or by the numeric expression <**expN2**>.

Description:

SKIP 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, SKIP moves the specified number of positions according to the index keys.

SKIP also obeys SET FILTER and SET DELETED when calculating the movement of the record pointer.

Skipping beyond the end-of-file positions the record pointer at RECCOUNT() +1, and EOF() returns .T. Skipping backwards beyond the beginning-of-file moves the record pointer to the first record, and BOF() returns .T.

Skipping on an empty index (created by INDEX...FOR), both BOF() and EOF() return TRUE and the record pointer is set beyond the end-of-file.

Multiuser:

Any record movement command, including SKIP, will make changes in the current working area visible to other applications, if the current file is shared and changes were made.

To force an update to become visible without changing the current record position, or to update the current FIELD variables, use SKIP 0 or COMMIT (or DBCOMMIT(), DBCOMMITALL() respectively). For further details, see chapter LNG.4.8.

Tuning:

see SET COMMIT

Example:

```
USE employee
? RECNO(), name                              &&   1  Miller
SKIP
? RECNO(), name                              &&   2  Johnson
SKIP 1 + MAX(3,2)
? RECNO(), name                              &&   5  Smith
SKIP -10
? RECNO(), BOF(), name                       &&   1 .T. Miller
SELECT 2
SKIP 5 ALIAS employee
? employee->(RECNO()), employee->name   &&   5  Smith
```

Classification:

database

Translation:

DBSKIP (expN1)   -or-   alias->(DBSKIP (expN1))

Related:

BOF(), EOF(), RECNO(), COMMIT, GOTO, SEEK, FIND, LOCATE, CONTINUE, SET COMMIT, DBCOMMIT(), DBCOMMITALL(), DBSKIP(), oRdd:Skip()

SORT ...ON...TO

Syntax:

```
SORT ON <field1> [/ [A|D] [C]]
               [,<field2> [/ [A|D] [C]] ...]
          TO <file>|(<expC>)
               [<scope>]
               [FOR <condition>] [WHILE <condition>]
```

Purpose:

Sorts records from the database in use to a new database file according to the specified key fields.

Arguments:

ON <**field1**...**fieldn**> are the fields to be used as sorting criteria.

TO <**file**> is the name of the target database file. Unless otherwise specified, the new file is assigned a .dbf extension. The given path or the SET DEFAULT is obeyed.

Options:

/A /D /C or **/AC** or **/DC** specifies the order of the <field> sorting:

/A	sorts records in ascending order from smallest to greatest value. This is the default setting.
/D	sorts records in descending order from greatest to smallest value.
/C	in case of a character field, ignores the character case.

<**scope**> is the part of the current database file to sort. The default scope is ALL.

<**condition**> specified by the FOR and/or WHILE clause restricts the range of the source database to be sorted and copied to the target file.

Description:

The SORT command is similar to INDEXing and COPYing one database to another. Therefore, the result of the sorting also depends on the current UNIQUE, FILTER and DELETED setting.

Character fields are sorted by the ASCII value of each character (obeying the sorting order set by FS_SET("loadlang")), date fields chronologically, numeric fields are sorted in numeric order, and logical fields with the FALSE value first. Memo fields cannot be sorted, but are copied to the target.

After replacing the sort key or adding a new record, the database is usually not sorted properly any more. Therefore it is more usual to use INDEX instead of SORT, because indices are always updated automatically when assigned to the database.

Multiuser:

In a multiuser environment, the source database file must be opened in EXCLUSIVE mode.

Performance:

For large databases, there is often much faster to use indices instead of SORT, since the SORT copies the whole database (or at least all records matching FOR/WHILE/REST criteria). See example below.

Example 1:

Outputs a list of magazine articles, grouped by the theme, from the latest to the oldest

```
USE article
SORT ON theme/AC, publ_date/D TO art_sort
USE art_sort
LIST theme, publ_date, author
```

Example 2:

The same example, using an index:

```
USE article
INDEX ON UPPER(theme) + DESCEND(DTOS(publ_date)) TO artsort
LIST theme, publ_date, author
```

Classification:

database

Compatibility:

The new database carries the same access rights as the source database does, see LNG.3.3.4.

Translation:

```
 __DBSORT ("file", {"fields"}, ;
          {for}, {while}, next, rec, .rest. )
```

Related:

INDEX, ASORT(), SET EXCLUSIVE, USE..EXCLUSIVE, oRdd:Sort()

STATIC

Syntax:

```
STATIC <memvar> [:= <exp>] [, ...]
```

Purpose:

Declares and optionally initializes STATIC variables and arrays.

Arguments:

<**memvar**> is the name of a FlagShip variable or array, to be declared in the (lexically scoped) STATIC class. The name may be of any length, but only the first 10 character are significant (see more LNG.2.6). Variable names in the FlagShip language are not case sensitive.

If the <memvar> is followed by square brackets [ ], an array is created. The number of elements for each array dimension can be specified as [dim1,dim2, ..,dimn] or [dim1][dim2][dimn]. The maximum number of dimensions and of the elements per dimension in FlagShip is 65535.

Options, Initializing:

<**exp**> is any valid FlagShip expression including a literal (constant) array to initialize the variable. If the initializer (:= <exp>) is not given, the variable (or all array elements) will be set to NIL.

The STATIC variable will be created on program start with a NIL value. The time of initialization with the <exp> value depends on the variable scope, see below.

Scope, Visibility:

The lifetime of STATIC variables is the entire program execution time. The scope and visibility is restricted to the containing procedure or .prg file, depending on where the declaration statement is placed:

- **UDF wide scope**: if the STATIC declaration is given within a procedure or function body, the variables are visible there only. The variable is initialized by the <exp> value when first entering the module.
- **File-wide scope**: if the declaration is placed before the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the variable is visible for all UDFs or UDPs within the .prg file. The initialization with the <exp> value is done when first entering any of the modules in the file.

The last value of a STATIC variable is available on subsequent entries into the module (or .prg file). If a procedure or UDF is invoked recursively (calls itself), each recursive activation may change the static variables.

The static variables can be passed by value or by reference to other UDFs or UDPs called at the same level. In code blocks, only STATIC variables of the module where the block is declared are visible; see LNG.2.3.3.

STATIC variable declarations hide all inherited PRIVATE, PARAMETERS, PUBLIC or FIELD variables with the same name. If the variable name is already declared in the same module by using another declarator (LOCAL, GLOBAL, MEMVAR, FIELD), a compiler error is generated.

For more information, refer to section LNG.2.6.

Description:

STATIC is a declaration statement that declares one or more variables or arrays static to the current procedure or user- defined function or the whole .prg file.

In FlagShip, the STATIC declarator may be placed anywhere in the function body; the scope and visibility for the compiler starts from this declaration on.

The variable names are known at compile-time only. Therefore, a STATIC variable can be evaluated by simple macros, but it cannot be used as composed macros or **within** the macro string; see also LNG.2.10. Static variables cannot be SAVEd and RESTOREd from .mem files, nor released by CLEAR or RELEASE.

To determine the type of a STATIC variable, only the standard function VALTYPE(varname) can be used; since the TYPE("varname") tries to evaluate the string using a macro and the variable is invisible during string evaluation.

Example:

```
*** File test1.prg ***
STATIC array1[20,10], array2[20][10]     // file-wide scope
STATIC array3 := { 1, 2, 3 }             // file-wide scope

PROCEDURE test1                          // not automatic.
LOCAL var1 := "test"
STATIC var2 := "test1"                   // UDP wide scope
STATIC array4 := {DATE(), TIME()}        // UDP wide scope
? array3[1], array4[2]
? test2 (var2), var2
RETURN

FUNCTION test2 (par)
STATIC var2 := VAL(TIME())               // UDF wide scope
? var2++, par
RETURN var2

Compile: $ FlagShip -na test1.prg [-m -c ...]
```

Classification:

programming

Compatibility:

The lexical scope is new in FS4, and is compatible to Clipper 5x. Clipper has a fixed order of the declaration and cannot use expressions to initialize the STATIC variable, but can use only a constant. Also the time of the initialization and the maximal array size is different in FlagShip and C5.

Related:

STATIC..AS, LOCAL, GLOBAL, PRIVATE, PUBLIC, FIELDS, DO, FUNCTION, TYPE(), VALTYPE(), CONSTANT

STATIC ... AS

Syntax 1:

```
STATIC <tvarList> [:= <expN>] AS <C-type>
```

Syntax 2:

```
STATIC_<C-type> <tvarList> [:= <expN>]
```

Purpose:

Declares and initializes C-TYPED STATIC variables.

Arguments:

<**tvarList**> is a comma separated list specifying the names of variables, to be declared as TYPED STATIC. The name may be of any length, but only the first 10 character are significant (see more LNG.2.6). The variable names in the FlagShip language are not case sensitive; when accessing them from #Cinline statements, use lowercase.

AS <**C-type**> is the alternate syntax to STATIC_<type> where <type> is one of the C-like type keywords listed in LOCAL...AS.

Example of valid syntax:

```
STATIC iVar := 4, ipos := 0, iCount AS INT
STATIC_LONG iOther := 5, myCount
```

Options, Initializing:

<**expN**> is any valid expression returning a numeric value within the <type> range to initialize the variable at the declaration time. If the initializer (:= <expN>) is not given, the TYPED STATIC variables is initialized with zero.

The TYPED STATIC variable are created and initialized in the same way as the STATIC variables, except that the initial value is zero instead of NIL.

Scope, Visibility:

The scope, visibility and lifetime of TYPED STATIC variables is identical to the usual, lexical STATIC variables. The only difference is the fixed storage type, which allows faster runtime access and the direct usage in #Cinline statements. The lifetime is the entire executable, the scope and visibility depends on the declaration placement:

- UDF wide scope: if the STATIC...AS declaration is given within the procedure or function body, the variables are visible in this entity only. The variable is initialized with the <exp> value when first entering the module.
- File-wide scope: if the declaration is placed prior to the first FUNCTION or PROCEDURE statement **and** the compiler switch -na is used, the variable is visible for all UDFs or UDPs within the .prg file. The initialization with the <exp> value is done when first entering any of the modules in the file.

The last value of the TYPED STATIC variable is available on the subsequent entries into the module (or .prg file). If a procedure or UDF is invoked recursively (calls itself), each recursive activation may change the static variables.

Typed variables can be passed by value to other UDFs or UDPs called at the same level. In code blocks, only STATIC variables of the module where the block is declared are visible; see LNG.2.3.3.

Like with all other lexical variables, the STATIC...AS declarations hide all inherited dynamic variables. For more information, refer to the section LNG.2.6.

Description:

STATIC..AS is a declaration statement that declares TYPED lexical variable, very similar to STATIC, but:

- The type and storage range is fixed during compile time and cannot be changed at runtime. Since additional runtime type checking may be omitted, the usage results in faster program execution.
- The variables occupy only 1, 4 or 8 bytes, compared to approx. 28 bytes for standard FlagShip variables.
- The programmer must consider the storage range of the variable's <type>. Otherwise, the resulting value will be truncated to the (last) available bytes.
- The typed variables can be accessed directly in #Cinline statements (giving the name in lowercase).
- A TYPED variable cannot be used for any macro evaluation, but are usable in code blocks. The function VALTYPE(varname) will return "N"; TYPE("varname") cannot be used.
- The TYPED variables will always be passed to a UDF and UDP by value, regardless of the calling convention used (@ prefix or the DO...WITH procedure call).
- If typed variables are intermixed with non-typed variables within an operation or command, they will be internally converted temporarily to non-typed ones. Therefore, use only typed variables or constants within the e.g. FOR... declaration to maintain the speed advantages.

The visibility is static to the current procedure, user-defined function or the whole .prg file. In FlagShip, the STATIC..AS declarator may be placed anywhere in the function body; the scope and visibility for the compiler start from this declaration.

Example:

See also examples in chapter LOCAL...AS and GLOBAL...AS

```
*** File test.prg
LOCAL angle
LOCAL radian, sine, cosine AS DOUBLE
STATIC_DOUBLE pi := 3.1415926535, deci := 2
DO WHILE .T.
   INPUT "Please enter angle 0..360 or <┘ only:" TO angle
   IF angle == NIL
      RETURN
   ENDIF
   radian := 2.0 * pi * angle / 360.0

#Cinline
   sine   = sin (radian);           /* std. math library */
   cosine = cos (radian);
#endCinline

   SET FIXED ON
   SET DECIMALS TO (deci++)
   ? "sin(" + ltrim(str(angle)) + ")=", sine , ;
     "cos(" + ltrim(str(angle)) + ")=", cosine

   angle := NIL                     // for the next entry
ENDDO
```

Classification:

programming

Compatibility:

Typed variables are available in FlagShip and VO only. To remain compatible to Clipper 5, use syntax 2 and #defines such as:

```
#ifndef FlagShip
 #define STATIC_BYTE STATIC
 #define STATIC_LONG STATIC
 #define STATIC_DOUBLE STATIC
#endif
```

Related:

STATIC, LOCAL...AS, LOCAL, GLOBAL, GLOBAL..AS, PRIVATE, PUBLIC, FIELDS, DO, FUNCTION, TYPE(), VALTYPE(), #Cinline, #define, #ifdef

STORE

Syntax 1:

`STORE <exp> TO <memvarList>`

Syntax 2:

`<memvar1> := [<memvar2> := ...] <exp>`

Purpose:

Initializes and/or assigns a value to one or more memory variables.

Arguments:

<**exp**> is a value of any data type (constants, expression, memory variables, database fields) that is to be assigned to the target memory variable(s).

<**memvarList**> are the memory variables of any class to initialize and/or assign values. Their names can have any length, only the first 10 characters are significant (see more LNG.2.6).

Description:

STORE is identical to the simple assignment operators = and *:* =, and refers to the syntax 1 and 2. STORE assigns the same value to a set of memory variables. If the variable name is unknown or invisible, a new autoPRIVATE is created.

When assigning a <memvar>, the same named memory variable takes precedence over a field variable. To assign a value to a database field (same as REPLACE), the variable must be declared as FIELD or the alias-> or the FIELD-> pseudo alias must precede the variable name.

On the other hand, when assigning <exp> to a variable, the field variable <exp> takes precedence over a dynamic memory variable with the same name, unless the declarator MEMVAR or the aliasing M-> or MEMVAR-> is used; see LNG.2.6 to LNG.2.9 and LNG.4.4.

To assign a value to an entire array, use the AFILL() function or the {...} literal array, see LNG.2.6.4 and LNG.2.7.

Example:

Create PRIVATE variables var1..var3 and a..c:

```
STORE "String" TO var1, var2, var3
a := b := c := 22
? var1, var3, a, c                      // String String 22 22
```

Classification:

programming

Compatibility:

The := assignment is available in FlagShip and C5 only. FlagShip allows an unlimited number of memory variables to exist at one time. The only physical limitation is the available RAM memory plus the swap space of the operating system.

Translation:

<var 1> := [<var 2 := . . .] <exp>

Related:

REPLACE, LOCAL, STATIC, GLOBAL, CLEAR MEMORY, PRIVATE, PUBLIC, RELEASE, SAVE, RESTORE

SUM

Syntax:

```
SUM [<scope>] <expList>
     TO <memvarList>
        [FOR <condition>] [WHILE <condition>]
```

Purpose:

Sums a list of numeric expressions to specified memory variables for a range of records in the current database file.

Arguments:

<**expList**> is a list of numeric expressions (typically database fields) to SUM for each processed record.

<**memvarList**> specifies the set of variables in which the results of summing are to be stored. If a variable does not exist or is invisible, a new autoPRIVATE is created. The <memvarList> must have the same number of elements as the <expList>.

Options:

<**scope**> is the part of the current database to SUM. The default scope is ALL.

<**condition**> specified by the FOR and/or WHILE clause restricts the range of the database records to be calculated, see general command description.

Description:

SUM totals a series of numeric expressions for a range of records in the current working area and assigns the results to a series of variables.

Example:

```
USE employee
present = 29.95
SUM no_child * present TO total_spend FOR EMPTY(leavedate)
? "We'll spend ", total_spend, ;
  " on Christmas presents for"
? "the employees' children"
```

Classification:

database

Translation:

```
var1 [ := var2 ... ] := 0
DBEVAL ({|| var1 += exp1 [, var2 += exp2... ]}, ;
       {for}, {while}, next, rec, .rest. )
```

Related:

AVERAGE, TOTAL, DBEVAL(), oRdd:Sum()

TEXT ... ENDTEXT

Syntax:

```
TEXT [TO PRINT]|[TO FILE <file> [ADDITIVE]]
     <text>...
ENDTEXT
```

Purpose:

Displays a block of text on the screen optionally echoing output to the printer and/or a text file.

Arguments:

<**text**> is the block of literal characters to be displayed on the screen, exactly as formatted. The <text> may contain any number of lines, separated by the NEWLINE (LF or CR/LF) character.

Options:

TO PRINT: echoes the display to the printer.

TO FILE: echoes the display to the specified <file>. Extension .txt is automatically added if no other is specified. If the clause **ADDITIVE** is specified, the text is appended to, instead of overwriting the <file>.

Description:

TEXT...ENDTEXT is a console command construct that displays a block of text lines to the screen, optionally echoing output to the printer and/or a ASCII file. To suppress the screen output, use SET CONSOLE OFF.

The text lines are displayed exactly as formatted in the .prg file, including any indentation. Macro variables encountered within <text> block are expanded in the same way as the macro text substitution).

Example:

```
USE address
DO WHILE .not. EOF()
   TEXT TO PRINTER
                           &first &last
                           &address
 Dear &title &lastname :
   ...
 Sincerely,
   ENDTEXT
   EJECT ; SKIP
ENDDO
```

Classification: programming, sequential output

Compatibility: The ADDITIVE clause is available in FlagShip only.

Related: ?, ??, @...SAY, MEMOEDIT(), MLCOUNT(), MEMOLINE(), LNG.2.10

TOTAL

Syntax:

```
TOTAL ON <keyExp> [<scope>]
         [FIELDS <fieldList>]
      TO <file>|(<expC>)
         [FOR <condition>] [WHILE <condition>]
```

Purpose:

Summarizes records by key value by summing specified fields and copying summary records to a new database file.

Arguments:

ON <**keyExp**> defines a group of records that produce, one after another, a new record in the target database. To make the summarizing operation accurate, the source database should be indexed or sorted on the same <keyExp>.

TO <**file**> is the target file where the summary records are to be copied. The default extension, if not otherwise specified, is .dbf.

Options:

FIELDS <**fieldList**> specifies the list of numeric fields to total. If the clause is not specified, the fields are not totaled; while the target record contains the values of the first record matching the <keyExp>.

<**scope**> is the part of the current database file to total. The default scope is ALL.

<**condition**> specified by the FOR and/or WHILE clause restricts the range of the database records to be totaled, see general command description.

Description:

The TOTAL command sequentially processes the current database summarizing records by the specified key value and copying them to a new database file.

When the TOTAL starts, it copies first the structure of the source database into the target; but without memo fields. It then sequentially scans the current database within the <scope>.

As each record with a unique <keyExp> is encountered, that record is copied to the target database. Otherwise, if the <fieldList> is specified, the values of source fields from the <fieldList> are added to the target fields.

Remember that numeric fields in the source database file must be large enough to hold the largest possible total for that field.

TOTAL considers the SET DELETED and SET FILTER status. If DELETED is OFF, the deleted records are copied to the target, but the delete flag will be not set.

Example:

```
LOCAL total := 0
USE employee
INDEX ON UPPER(lastname) TO emplname
TOTAL ON UPPER(lastname) FIELDS salary TO summary ;
      FOR YEAR(salarydate) = YEAR(DATE())

USE summary
SET CENTURY ON
? "Salaries for the year", YEAR(DATE()), "to", DATE()
WHILE .not. EOF()
   ? lastname, salary
   total += salary
   SKIP
ENDDO
? "Total:", total
```

Classification:

database

Translation:

__DBTOTAL ("file", {keyexp}, { "field1"... }, ;
{for}, {while}, next, rec, .rest.)

Related:

AVERAGE, SUM, INDEX, SORT, oRdd:Total()

TYPE

Syntax:

```
TYPE <file1>|(<expC>)
        [TO PRINT]
        [TO FILE <file2>|(<expC>) [ADDITIVE]]
```

Purpose:

Displays the contents of a text file to the screen, printer and/or to another file.

Arguments:

<**file1**> is the name of the ASCII file, including extension, that is to be displayed.

Options:

TO PRINT sends the display to the printer file/device. The clause is equivalent to SET PRINTER ON.

TO FILE <**file2**> sends the display to the <file> specified in this clause. If no extension is specified, .txt is added. When using the **ADDITIVE** clause, the text is added to, instead of overwriting the file.

Description:

TYPE is a console command that displays the contents of a text file to the screen, optionally echoing the display to the printer and/or another text file. To suppress the screen output, use SET CONSOLE OFF.

Ctrl-S is used to pause output. ESC has no effect on interrupting the listing. For a similar, but paged output, RUN "cat file1 **|** pg" or MEMOEDIT() can be used.

Example:

```
TYPE test.prg

@ MAXROW(),0 SAY "Scroll by PgUp/PgDn, ESC to continue"
MEMOEDIT(MEMOREAD("test.prg"),1,0,MAXROW()-1,MAXVOL(), .F.)
RUN MESSAGE "press any key..." cat test.prg | pg
INKEY (0)
REFRESH
```

Classification:

sequential output

Compatibility:

The ADDITIVE clause is available in FlagShip only.

Translation:

__TYPEFILE ("file1", .print., ["file2"], [.add.])

Related:

COPY FILE, SET DEFAULT, SET PATH, SET PRINTER, FS_SET()

UNLOCK

Syntax:

```
UNLOCK [<expN> | ALL]
```

Purpose:

Releases file or record locks, which were set by the current program.

Options:

<**expN**> is the physical record number, that is to be unlocked. When SET MULTILOCKS is ON and this argument is given (e.g. UNLOCK RECNO() or UNLOCK 5), only the specified record is unlocked, if such was previously RLOCKed. Otherwise, all record/file locks are removed.

ALL: if this clause is specified, all locks set by the current program in all used working areas are released. If the ALL clause is not given, only locks of the currently selected database are released.

Description:

UNLOCK frees all records and file locks of the current or of all databases used in the multiuser/multitasking mode.

Multiuser:

In multiuser mode (or when using a concurrent databases accesses 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 (like REPLACE, DELETE, RECALL), the record or the database must be locked using RLOCK() or FLOCK(). Otherwise, if the lock is not set by the programmer, FlagShip invokes the AUTOxLOCK() function, if this is not disabled by SET AUTOLOCK OFF. The only exception is APPEND BLANK which locks the new record automatically.

Note: global changes of the physical record storage (PACK and ZAP) or rebuilding the index files (REINDEX and usually INDEX ON) requires an EXCLUSIVE open mode; for more see LNG.4.8.

When the write access is finished, UNLOCK will release the previously set record and file locks, so that another user may lock the file or record. If the clause ALL is given, locks of all active working areas are released.

UNLOCK does not automatically release a record lock along a RELATION chain unless UNLOCK ALL or alias-> (DBUNLOCK()) is used.

If the AUTOxLOCK() function is invoked by FlagShip, it releases the lock automatically after the write access, by using the AUTOUNLOCK() function.

Using another RLOCK(), FLOCK() or APPEND BLANK will also release the previous locking of the current database. Closing the database and ending or aborting a program automatically releases all locks set by this executable.

The UNLOCK command in FlagShip implies updating the current working area buffers to Unix/Windows, which makes changes visible to other applications. To flush the Unix or MS-Windows buffers physically to the file, use COMMIT, DBCOMMIT() or DBCOMMITALL(), see section LNG.4.8. When SET AUTOCOMMIT is ON (default is OFF) or the `_aGlobSetting[GSET_N_DBCOMMIT_UNLOCK]` is > 0, every UNLOCK will also commit (flush) the current work area physically to hard disk, same as COMMIT - avoid this in a large loop since it will decrease the performance significantly.

See also SET COMMIT for additional tunings.

Example:

```
LOCAL new_name := "Smith", new_first := "Peter", count
SET EXCLUSIVE OFF
USE address NEW
WHILE NETERR(); USE ADDRESS; END
SET INDEX to adr1, adr2
FOR count = 1 TO 10          // try RLOCK() up to 10 times
   IF RLOCK()
      REPLACE name  WITH new_name
      REPLACE first WITH new_first
      UNLOCK
      EXIT
   ELSEIF count < 10
      ? "waiting to lock this record"
      INKEY(1)
   ELSE
      ? "update failed"
   ENDIF
NEXT
```

Classification:

database

Compatibility:

This command and its usage is fully compatible to other xBASE dialects. The internal locking mechanism conform to the Unix and Windows (Posix) standard; the locking mechanism of other xBASE derivates are mostly not compatible.

In FlagShip, the multiuser mode also applies to the same database concurrently open in different working areas; cf. the USE command.

The AutoLock and multiple record locking feature is new in FS4 and not available in Clipper.

Translation:

DBUNLOCK([expN]) | DBUNLOCKALL()

Related:

SET EXCLUSIVE, SET AUTOLOCK, SET MULTILOCKS, SET COMMIT, USE, FLOCK(), RLOCK(), APPEND BLANK, AUTORLOCK(), AUTOFLOCK(), oRdd:Unlock(), oRdd:RlockList()

UPDATE

Syntax:

```
UPDATE ON      <keyExp>
       FROM    <alias>|(<expC>)
       REPLACE <field1> WITH <exp1>
               [, <field2> WITH <exp2>...]
               [RANDOM] [APPEND <expB>]
```

Purpose:

Updates the current database file from another database file based on a key expression.

Arguments:

ON <**keyExp**> is an expression defining the FROM records to be read.

FROM <**alias**> specifies the source working area which updates the current (target) database.

<**fieldn**> is the target field to be updated.

<**expn**> is the value or expression which updates the <fieldn>. Any field contained in the FROM working area must be referenced with the <alias>-> selector.

Options:

RANDOM clause is used, when the FROM database is not indexed or sorted on <keyExp>, but only the current database has such an index active. If the RANDOM clause is not used, both the target and source must be indexed or sorted by the <keyExp>.

APPEND <**expB**> clause specifies, that, in case the <keyExp> is not found in the current database, a new record should be added, the code block evaluated, and the fields replaced by the WITH clause(s). If the current database has not active index, no action is performed, since unpredictable results may occur.

Description:

UPDATE replaces fields in the current working area with values from another working area based on the specified key expression.

The UPDATE command supports 1:n and n:1 logical relations between the current and the FROM area. When in the current (target) working area there is more than one instance of the <keyExp> value, only the first record with the key value is updated. However, the FROM work area can have duplicate key values, which multiply replace the target.

UPDATE works in two different ways, depending on the RANDOM clause:

- If RANDOM is specified, the current database must have an active index matching <keyExp>. The FROM database is skipped, while the matching record is SEEKed in the current database and updated only if found; or a new record is added in the current database, if the APPEND clause is given.

- If the RANDOM clause is **not** specified, the current database is SKIPped according to the FROM sequence. The target record is updated only if the <keyExp> values exactly match in both the target and source record; or a new record is added in the current database, if the APPEND clause is given.

Multiuser:

In multiuser mode, the current database file must be locked with FLOCK() or used EXCLUSIVEly; otherwise AUTORLOCK is used, if possible. The FROM database file may be opened in any mode.

Example:

```
SELECT 1
USE custom INDEX custom EXCLUSIVE
? INDEXKEY()                           && UPPER(name)
USE invoice INDEX inv_cust ALIAS inv NEW
? INDEXKEY()                           && UPPER(name)
SELECT 1
UPDATE ON UPPER(name) FROM inv ;
   REPLACE debit WITH debit + inv->amount
USE orders NEW
SELECT 1
FIELD name, id
UPDATE ON UPPER(name) FROM orders ;
   REPLACE debit WITH debit + orders->amount ;
   APPEND {|| name := orders->name, id := orders->idnum }
```

Classification:

database

Compatibility:

The APPEND clause is available in FlagShip only.

Translation:

```
__DBUPDATE ("alias", {keyExp}, .random, ;
 {|| _FIELD->fld1 := exp1 [, _FIELD->fld2 := exp2...] }, ;
 [appendBlock] )
```

Related:

APPEND FROM, REPLACE, JOIN, INDEX, SORT, SET AUTOLOCK. oRdd:Update()

USE

Syntax:

```
USE <file>|(<expC>)
       [ALIAS <alias>|(<expC>)]
       [INDEX <fileList>|(<expC>)]
       [EXCLUSIVE | SHARED]
       [READONLY]
       [NEW]
       [NFS]
       [VIA <driver>]
```

Syntax 2:

```
USE
```

Purpose:

Opens the specified database file, its associated memo file when memo fields exist, and, optionally, associated index files in the selected working area.

Arguments:

<**file**> is the name of the database file to open in the current (or a NEW) working area. If an extension is not specified, the default .dbf extension is assumed. The <file> can optionally include drive and/or path. If only file name is given, the database file name is searched for:

- in the current directory (see also note below)
- in the path specified by SET DEFAULT statement (if any)
- in all paths specified by SET PATH command (if any)

If the file could not be opened (file not found or access denied), no error message occurs when SET OPENERROR is OFF. You therefore should test the success or failure of USE command by subsequent USED() which returns .T., or by NETERR() which returns .F. on success.

If <file> is not specified (syntax 2), the current working area is closed, equivalent to the CLOSE command.

Options:

ALIAS <**alias**> is the name to be associated with the working area. If not specified, the main part of the <file> name is assigned to <alias>. If the given alias is invalid, USE will try to generate a valid name and will display corresponding warning.

INDEX <**fileList**> specifies up to 15 index files to be opened in the current working area. Each index file may be specified either as a literal filename or as a character expression enclosed in parentheses. If the <expC> returns NIL or an empty string, it is ignored. The first index file in the list becomes the controlling index. It is not recommended to use this clause in multiuser mode. See also SET INDEX, INDEX ON and LNG.4.5.

EXCLUSIVE opens the database file for non-shared use in a network or multitasking environment. Other users cannot access the database until it is closed. It is a synonym for USE with SET EXCLUSIVE ON. This clause overrides the SET EXCLUSIVE state.

SHARED opens the database file for shared use in a multiuser, multitasking, network or concurrent mode. It is a synonym for USE with SET EXCLUSIVE OFF. This clause overrides the current SET EXCLUSIVE state.

NEW selects an unused working area making it the current one, and opens the database <file> there. The clause is equivalent to SELECT 0 prior to the USE... command. If this clause is not given, the database is opened in the currently SELECTed working area.

NFS clause enables the global switch SET NFS ON, which then remain active for all subsequent database actions and USEs until SET NFS is set OFF. This clause can be used for databases and indices located on NFS mounted file system, when the NFS server does not flush all buffers correctly - resulting sometimes in corrupted index reported by IndexCheck(). See further details in the SET NFS description.

READONLY opens the database for read-only purposes. The Unix access rights -r- (or Windows read-only) are sufficient for the database and memo <file> (but not for index files, which must be always -rw- or read+write). An attempt to REPLACE or APPEND a record brings a run-time error.

VIA <**driver**> defines the replaceable database driver (RDD) to use for the current working area. The default is the "DbfIdx" RDD. You may need to add the statement "EXTERN <driver>" (e.g. EXTERN myRdd) to force the linker to add your RDD driver into the executable, if the object file containing "CLASS myRdd" is not linked explicitly. To set another driver globally, use RddSetDefault("myRdd")

Description:

USE opens an existing database .dbf file, its associated memo .dbt (or .dbv) file, and optionally associated index .idx files in the current or the next available working area. Before USE opens a database file and its associated files, it closes any active files already open in the working area.

Note: when only <file> name is given (without path), the database is tried to open in the current directory (and then in SET DEFAULT, SET PATH directories when specified). This means "access in current directory" work fine when you start the executable from the working directory containing your data. However when you (or the end-user) invoke the application from other drive or directory or by searching the PATH environment variable, or via link on desktop, or via file manager (like Windows Explorer, NC, Konqueror etc.), the used "current directory" is most probably not what you really meant :-) In such a case, either use fully qualified file names, or (better) specify the current directory by CURDIR("/my/data") or use SET DEFAULT TO "D:\my\data" or SET DEFAULT TO (getenv("MYDATA")) which reads the setting from user's environment variable etc. In doubt, you may check/display the current directory by CURDIR() and the availability of files by IF !FILE("mydata.dbf") ; SET DEFAULT ... see example below.

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.

When opening an empty database or an empty index (created by INDEX...FOR), both BOF() and EOF() return TRUE and the record pointer is set beyond the end-of-file.

USE without an argument closes the active database file and the associated memo and index files, if any, equivalent to the CLOSE command. To close the database files in all work areas, use CLOSE ALL or CLOSE DATABASES.

When the open fail, NetErr() will report .T. and Used() .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 NetErr() or Used() status.

Additional warnings are available by using FS_SET("devel", .T.)

In FlagShip, up to 65534 working areas may be opened simultaneously, each with up to 65534 indices and relations.

Each working area has the following attributes:

Attribute/Action	Retrieving Command/Function
Open/close work area	USE, CLOSE DATA
Change work area	SELECT wano, SELECT alias
Indices	USE..INDEX, SET INDEX
Relations	SET/CLOSE RELATION
Filtering	SET FILTER, SET DELETED
Searching	SEEK, LOCATE, FIND
Moving	GOTO, SKIP
Alias name	ALIAS()
Database file	DBF(), INDEXDBF()
Working area no.	SELECT()
Index file ext, names	INDEXEXT(), INDEXNAMES()
Index key, contrl.no.	INDEXKEY(), INDEXORD()
Index integrity check	INDEXCHECK()
Record number	RECNO()
Record count	LASTREC(), RECCOUNT()
Field count	FCOUNT()
Field name	FIELD()
Field description	AFIELDS()
Beginning-of-file flag	BOF()
End-of-file flag	EOF()
Filter condition	DBFILTER(), DELETED()
Locate/Seek result	FOUND()
Relation	DBRELATION(), DBRSELECT()
Header size	HEADER()

Network cmd result	NETERR()
Locking	RLOCK(), FLOCK(), UNLOCK, SET AUTOLOCK, AUTOxLOCK(), SET MULTILOCKS

Multiuser:

If a multiuser, multitasking and/or network access is required, database files can be opened EXCLUSIVEly or SHARED. The exclusive status stops the database from being used by other users (or in other working areas concurrently) until the file is closed; the shared mode allows other users to use the database and its associated files for concurrent access.

The open status of the database is determined by the SET EXCLUSIVE command, or the EXCLUSIVE or SHARED clause respectively:

- If SET EXCLUSIVE is ON (the default), the database is open exclusively; the given SHARED clause will override the current global setting.
- If SET EXCLUSIVE is OFF, the database is open in sharable mode. Specifying the EXCLUSIVE clause on the USE command will override the default setting.

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 another user (or concurrently in another working area).

Instead of USE..INDEX.. it is better practice to open the database, check success by USED(), then assign index/indices by SET INDEX TO.. and check success by NETERR() which should be .F.

In the SHARED mode, any **write** attempt to the database or memo file (like REPLACE, DELETE, RECALL, or alias->name := ...) requires that the current record or the whole file is locked beforehand using RLOCK() or FLOCK(). This will ensure data integrity denying other users a write access to the same record or database. When the write access is finished, use UNLOCK or UNLOCK ALL to release the previously set record and file locks, so that another user may lock the file or record. If the lock is not set by the programmer and SET AUTOLOCK is ON, FlagShip locks the record or file automatically by using the AUTOxLOCK() function.

Global changes to the physical record storage order (PACK and ZAP) or rebuilding the index files (INDEX, REINDEX) requires an EXCLUSIVE open mode.

Concurrent Databases:

For special purposes, FlagShip allows the same database to be USEd simultaneously in different working areas, when the given ALIAS names are different. Note: FlagShip distinguishes between database equivalents on the same inode number, not only on the DBF() name itself. When performing operations on the SAME physical database (used concurrently in different working areas), see also chapter LNG.4.8.7.

The handling of concurrent databases is the same, as the usage of shared databases in multiuser mode. Therefore, using concurrent databases in the same application requires their SHARED use, NETERROR() checking and RLOCK() or FLOCK() on write access.

Large Files:

FlagShip supports also large files >> 2Gigabytes. If you need to use this feature, enable SET LARGEFILE ON at the begin of your application, latest before open the database (it is enabled automatically in VFS8 and later). See additional details in CMD.SET LARGEFILE.

Tuning:

As noted above, FlagShip do not raise run-time error on failure, so check by 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.

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 1:

Open 3 databases (and their indices) in single user mode:

```
SELECT 22
USE address                               // address in WA 22
SELECT 1                                  // customer in WA 1
USE "\data\customer" ALIAS cust INDEX customer
if neterr()
   alert("cannot open customer database or index")
endif
USE invoices NEW                          // invoices in next WA
IF .not. FILE("inv_1" + INDEXEXT())       // inv_1.idx
   INDEX ON customno TO inv_1
   INDEX ON invdate  TO inv_2
ENDIF
SET INDEX TO inv_1, inv_2
```

Example 2:

Open a database and its indices in multiuser mode:

```
SET EXCLUSIVE OFF                         // multiuser
USE address
count := 0
while !used() .and. file("address.dbf")   // error ?
   sleepms(100)                           // wait 0.1 sec
   USE address                            // yes, try again
   if ++count > 20                        // but max.
      exit                                //  for 2 seconds
   endif
ENDDO

if !used()
   alert("cannot open address.dbf")
   quit
```

```
endif
SET INDEX TO adr_name, adr_zip
if NETERR()
   alert("could not open index ...")
   quit
endif
```

Example 3:

Open a database and indices SHARED with RTE on failure

```
LOCAL dbfname := "address", idx1 := "adr_name", idx2 := "adr_zip"
_aGlobSetting[GSET_L_DBUSEAREA_ERR ] := .T. // force RTE on failure
_aGlobSetting[GSET_L_DBSETINDEX_ERR] := .T. // force RTE on failure

USE (dbfname) SHARED NEW INDEX (idx1), (idx2)
```

Example 4:

Check and set the "working directory" from the current location, or from user environment variable MYDATA, or from location of the executable:

```
#include "fspreset.fh"              // optional, see LNG.9
LOCAL dbfname := "address"
LOCAL cWorkDir := "", lFound := .F.
// check current directory
if file(dbfname + ".dbf")
   lFound := .T.                    // everything is ok
endif
// check directory specified in environment variable
cWorkDir := getenv("MYDATA")
if !lFound .and. !empty(cWorkDir)
   if file(cWorkDir + PATH_SLASH + dbfname + ".dbf")
      SET DEFAULT TO (cWorkDir)     // or: CURDIR(cWorkDir)
      lFound := .T.
   endif
endif
// check the directory of executable (i.e. of current .exe file)
cWorkDir := left(execname(.T.), rat(execname(.T.),PATH_SLASH) -1)
if !lFound .and. !empty(cWorkDir)
   if file(cWorkDir + PATH_SLASH + dbfname + ".dbf")
      SET DEFAULT TO (cWorkDir)     // or: CURDIR(cWorkDir)
      lFound := .T.
   endif
endif
if !lFound
   alert("Sorry, cannot locate the databases, set MYDATA envir.")
   quit
endif
// now you can handle it w/o worrying about current directory
USE (dbfname) ...
```

Example 5:

Example for multitasking, support of the DOS file and path names also in Unix/Linux, the usage of a general open routine, including checking for success:

```
SET EXCLUSIVE OFF                      // multiuser mode
#ifdef FlagShip
   FS_SET ("lower", .T.)               // auto file transl.
   FS_SET ("pathlower", .T.)           // auto path transl.
   IF GETENV("C_FSDRIVE") == ""        // C: drive substitution
      ? "set the environment var C_FSDRIVE first !"
      QUIT
   END
#endif
SET DEFAULT TO C:\Data\Adr          // DOS path support is avail.

SELECT 23
IF .not. FILE("adr_name" + INDEXEXT()) .OR. ;
   .not. FILE("adr_idno" + INDEXEXT())
   my_use ("Address",, .T., .F.)       // USE..EXCLUSIVE
   INDEX ON UPPER(name) + STR(zip,1,5) TO adr_name
   INDEX ON custno TO adr_idno
END
my_use ("address", "adr" .F., .F.)     // USE..SHARED
SET INDEX TO adr_name, adr_idno
SELECT adr
SEEK "SMITH"                           // seek name
SEEK "SMITH      54321"                // seek name + zip
SET ORDER TO 2
SEEK 12345                             // seek id number

FUNCTION my_use (dbf, alias, excl, new)
***************************************
* [dbf]   (C) dbf name
* [alias] (C) alias name or NIL for alias=dbf
* [excl]  (L) .T. use exclusive, NIL for SET EXCLUSIVE flag
* [new]   (L) use in a new working area

LOCAL timebeg := SECONDS(), ii := 0
LOCAL shared  := IF (excl != NIL, !excl, NIL)
new   := IF (new   == NIL, .F., new)
alias := IF (alias == NIL .or. EMPTY(alias), NIL, alias)
#define TIMELIMIT 10
IF EMPTY(dbf)
   USE
ELSE
   DBUSEAREA (new, , dbf, alias, shared)
   WHILE NETERR() .AND. (SECONDS() - timebeg) <= TIMELIMIT
      @ 0,0 SAY str(++ii) + ". try to open database " + dbf
      INKEY(1)                          // error, try again
      DBUSEAREA (new, , dbf, alias, shared)
      @ 0,0
   ENDDO
ENDIF
RETURN USED()
```

Classification:

database

Compatibility:

The clause NEW, SHARED, EXCLUSIVE, READONLY and VIA are available in FS4 and C5 only. FlagShip is able to handle 65534 working areas simultaneously, each with additional memo files and up to 15 indices. Clipper'87 supports 255, Clipper 5.x and VO up to 250 working areas only. Refer to the section SYS for the Unix kernel settings for open file limits (often in /proc/sys/fs/file-max).

Large file support (over 2 Gigabytes) depends on the used operating system and is available in VFS 6.1 and newer. Once the file exceeds the 2 GB limit, it is **incompatible** to DOS!

See also chapter LNG.9.5 describing how to maintain full compatibility to the DOS written programs running on Unix. All databases and memo files must be transferred from or to DOS using a **binary** protocol. If the text mode is used the database is corrupted !

Translation:

```
DBUSEAREA(.new., "rdd", "dbfName", "alias", .shared., .ronly.)
[ DBSETINDEX (indexname1) ...]
```

Related:

CLOSE, SELECT, SET AUTOLOCK, SET MULTILOCKS, SET INDEX, NETERR(), SELECT(), RLOCK(), FLOCK(), UNLOCK, USED(), DBF(), FS_SET(), DBUSEAREA(), RddSetDefault(), SET LARGEFILE, OBJ.6, RDD.3

WAIT

Syntax:

```
WAIT [POPUP | WINDOW]
     [TIMEOUT  <expN2>]
     [COLOR    <expC3>]
     [GUICOLOR <expC4>]
     [GUISHAPE <expN5>] [NOSHAPE]
     [ECHO|NOECHO]
     [<expC1>]

WAIT [<expC1>]
  TO <memvarC>
     [POPUP | WINDOW]
     [TIMEOUT  <expN2>]
     [COLOR    <expC3>]
     [GUICOLOR <expC4>]
     [GUISHAPE <expN5>] [NOSHAPE]
     [ECHO|NOECHO]
```

Purpose:

Displays a prompt and waits for a key to be pressed.

Options:

<**expC1**> is the user prompt which is displayed if specified. It can be an expression of any data type. If <expC1> is not specified, the default prompt "Press any key to continue..." will be displayed. Note: this default string is pre-defined in the global variable _aGlobSetting[GSET_C_WAITPROMPT] (see also the ininit.prg source) and may hence be simply re-defined to your preferred prompt text. If a null string is specified, only NEWLINE is printed. When SET CONSOLE is OFF, neither newline nor the prompt is displayed.

<**memvarC**> is the memory variable to contain the character entered. If the variable does not exist or is not visible, a new autoPRIVATE one is created.

POPUP displays the message in Popup window (MessageBox) instead of the next console row. The equivalent **WINDOW** clause is supported for FoxPro compatibility.

TIMEOUT <expN2> waits max. for <expN2> seconds. If not given, wait until user key press.

COLOR <**expC3**> specifies the color for displaying the <expC1> data. Only the first color pair (standard) is significant. If this clause is not given, the current color setting is used. In GUI mode, first the GUICOLOR clause is checked. If not set, the COLOR <expC3> 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 COLOR and SET GUICOLOR setting.

GUICOLOR <**expC4**> specifies the color for displaying the <expC1> data considered in GUI mode. Only the first color pair (standard) is significant. If GUICOLOR is set, it is used regardless the current SET GUICOLOR on/off setting. If omitted and SET GUICOLOR is ON, either the COLOR <expC3> is used if given, or the current SetColor() is used. The GUICOLOR clause apply for GUI mode only, and is ignored otherwise.

GUISHAPE <**expN5**> specifies the text cursor shape displayed at the end of the prompt message <expC1> and signaling an user input. Apply for GUI mode only, ignored otherwise. The default shape is CURSOR_HAND, but may be re-defined by any other value assigned to global variable _aGlobSetting[GSET_G_N_WAITSHAPE]. You may override it temporarily by setting your own shape using the CURSOR_* constant or it corresponding value:

mouse.fh constant	value	Description
	0	same as CURSOR_INVISIBLE
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	same as CURSOR_UNDERSCORE

NOSHAPE disables the text cursor displayed at the end of the prompt message and is equivalent to GUISHAPE 0 or GUISHAPE CURSOR_INVISIBLE clause. In Termnal i/o mode, it is equivalent to SET CURSOR OFF.

ECHO or **NOECHO** clause temporarily overrides the current setting of Set(_SET_WAIT_ECHO) and indicates or suppress displaying the pressed key. Echo is displayed only when SET CONSOLE is ON.

Description:

WAIT is a console command with wait state. The specified or default prompt is displayed after a NEWLINE. The command then waits for a user input or reads one from the type-ahead buffer. The input key is echoed on the screen if not disabled by the NOECHO clause or by global setting SET(_SET_WAIT_ECHO,.F.).

When the TIMEOUT <expN2> is specified and a key is not pressed within the given time frame, WAIT exits returning "".

When a key assigned via SET KEY or ON KEY is pressed, the UDF is executed and WAIT waits for next key input. When the Escape (K_ESC) key was assigned to an UDF by SET KEY or ON [ANY] KEY, you need to press Escape key twice to terminate WAIT - this avoids a possible infinite loop.

When a FN key is assigned to a string by SET FUNCTION and this FN key was pressed, WAIT exits returning the assigned string in the <memvarC> variable.

For backward compatibility to other xBase dialects, function keys are ignored if not associated to SET KEY, ON KEY or SET FUNCTION. When function keys (i.e. inkey() values 28 and less than 0) should exit WAIT too, use SET(_SET_WAIT_IGNFUN,.F.) - the default setting is .T.

When you don't wish echo the input key, use Set(_SET_WAIT_ECHO,.F.) where the default setting is .T. You also may use the ECHO | NOECHO clause to temporarily override the global status for this WAIT.

WAIT displays the <expC1> or standard prompt per default also with SET CONSOLE OFF. You may disable this feature by assigning
`   _aGlobSetting[GSET_L_WAIT_PROMPT] := .F.   // default is .T.`
which then considers current SET CONSOLE setting. Without this set, you also may use WAIT "" to avoid prompt, or WAIT "" NOECHO which is then equivalent to Inkey(0). The above setting however does not affect waiting for user key press, which always apply.

Example:

```
WAIT TO key
//
WAIT NOECHO "press any key to continue, ESC to abort"
IF lastkey() = 27
   QUIT
ENDIF
//
@ 10,0 say "Press any key: "
key := INKEY(0)
IF key > 32
   ?? CHR(key)
ENDIF
```

Classification:

sequential screen output, waiting keyboard input

Compatibility:

The support of foreign language prompts is available in FlagShip only. The COLOR, GUICOLOR, POPUP, TIMEOUT, ECHO, NOECHO, GUISHAPE and NOSHAPE clauses are new in FS5. For FlagShip 4 compatible behavior, use Set(_SET_WAIT_IGNFUN,.F.), see text above.

Translation:

[var :=] __WAIT ([exp], ...)

Related:

@..GET, READ, ACCEPT, INPUT, INKEY(), FS_SET(), ?, ??, QOUT(), SET GUICURSOR, SetGuiCursor()

ZAP

Syntax:

ZAP

Purpose:

Removes all records from the currently selected database file.

Description:

ZAP permanently removes all records from the database, memo file and associated indices in the current working area. The disk space previously occupied by the allocated files is released.

ZAP performs similar operation as COPY STRUCTURE and REINDEX commands. It is therefore significantly faster, than the similar DELETE ALL followed by PACK.

Multiuser:

ZAP requires an exclusively opened database using SET EXCLUSIVE ON or USE...EXCLUSIVE. If not so, RTE (Run-Time-Error) displays and ZAP is not performed. See also LNG.4.8.

Tuning:

ZAP creates temporary database newNNNNN.dbf (and .dbt, dbv, .fpt if required) in the same directory where the database resides. The NNNNN is the current process ID number of the executable. If such a file exist, newHHMMSSUUUU.dbf is created, the format is of Time(1). These files are deleted after completing the ZAP. You may assign any other directory for these temporary files by environment variable FSPACKDIR, e.g. SET FSPACKDIR=[drive:]\path in Windows, or export FSPACKDIR=/path in Linux.

Example 1:

```
SET EXCLUSIVE ON                // default setting
USE taxes INDEX tax             // assuming index exists
? RECCOUNT()                    // 34
ZAP
? RECCOUNT()                    // 0
```

Example 2: *The same example as multiuser with check:*

```
USE taxes INDEX tax EXCLUSIVE   // open exclusive, index exists
while !used() .or. neterr()     // check
   ok = alert("could not open 'taxes' exclusive;close by other", ;
              {"Retry", "Quit"} )
   if ok != 1
      quit
   endif
   sleep(2)
enddo
? RECCOUNT()                    // 34
ZAP                             // clear database and index
USE taxes INDEX tax SHARED      // re-open multiuser
// optionally check again whether open...
? RECCOUNT()                    // 0
```

Classification:

database

Translation:

__DBZAP( )

Related:

DELETE, PACK, USE, COPY STRUCTURE, oRdd:Zap()

Index

Notes

MULTISOFT

multisoft Datentechnik
Schönaustr. 7
D-84036 Landshut

http://www.fship.com
sales@multisoft.de
support@flagship.de