

FlagShip



**Object Oriented
Database
Development System**

**Cross-Compatible to Unix,
Linux and MS-Windows**

 **MULTISOFT**

Release 8.1

Section

EXT

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 " fsman " contains all above sections, search function, and additionally last changes and extensions



multisoft Datentechnik, Germany

Copyright (c) 1992..2017
All rights reserved



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

Section EXT

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

Copyright © 1992..2017 by multisoft Datentechnik, D-84036 Landshut, Germany. All rights reserved worldwide. Manual authors: Jan V. Balek, Ibrahim Tannir, Sven Koester

No part of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, manual, or otherwise; or disclosed to third parties without the express written permission of multisoft Datentechnik. Please see also "License Agreement", section GEN.2

Made in Germany. Printed in Germany.

Trademarks

FlagShip™ is trademark of multisoft Datentechnik. Other trademarks: dBASE is trademark of Borland/Ashton-Tate, Clipper of CA/Nantucket, FoxBase of Microsoft, Unix of AT&T/USL/SCO, AIX of IBM, MS-DOS and MS-Windows of Microsoft. Other products named herein may be trademarks of their respective manufacturers.

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>

EXT: API C Interface

1. Using the C Interface	4
2. Extend C API System	6
2.1 Structure of Extend Programs	6
2.2 Naming Convention	10
2.3 Access to FlagShip Variables and Arrays	10
2.4 Accessing and Modifying String Values	13
2.5 Access to UNIX/Windows Runtime Libraries	14
2.6 Using C Input/Output Functions	14
2.7 Using the Open C Interface.....	14
2.8 Compiling the C programs	15
2.9 The Extend API System Reference	15
_parc ().....	16
_parcLen () _parcsiz ().....	18
_pards ().....	20
_parinfa ()	22
_parinfo ()	24
_parl ().....	26
_parnD ()	27
_parni () _parnl ()	28
_parscw ().....	29
_ret ().....	30
_retC () _retclen ().....	31
_retds ().....	32
_retl ().....	33
_retnD ()	34
_retni () _retnl ()	35
_retsCw ()	36
_storc () _storclen ()	40
_stords ()	42
_storl ().....	43
_stornd ()	44
_storni () _stornl ()	45
_xalloc () _xgrab ().....	46
_xfree ().....	47
ALENGTH ()	48
ISARRAY ()	49
ISBYREF ()	50
ISCHAR ().....	51
ISDATE ()	52
ISLOG ()	53
ISMEMO ().....	54
ISNUM ().....	55
ISSCREEN ()	56
FSinit ()	57
FSudfname ().....	58
FSreturn	59

3. Inline C Programming	60
3.1 Structure of Inline C Programs	60
3.2 Compiling the Program	61
3.3 Accessing Unix/Windows Libraries	61
3.4 Using an Inline C Function	62
3.5 Prototyping, Casting	63
3.6 Using own main() function	64
4. Open C System	66
4.1 Structure of the Executable Program	66
4.2 Structure of the C program	66
4.3 Modifying the C Source	71
4.4 The FlagShip Open C API System	72
4.5 Using FlagShip Variables	73
4.6 Using FlagShip Standard Functions	78
4.7 Screen and File Input/Output	80
4.8 Using the Open C API	81
4.9 Access to C++ functions and classes	85
4.10 The Open C API Reference	87
IS_VAR_xxx ()	87
IS_VAR_EQ () IS_VAR_EE () IS_VAR_NE () IS_VAR_GT () IS_VAR_GE ()	
IS_VAR_LT () IS_VAR_LE ()	90
OBJ_DECL_METH () OBJ_DECL_METHACCESS () OBJ_DECL_METHASSIGN ()	92
OBJ_METHEXEC () OBJ_ACCEXEC () OBJ_ASSEXEC ()	94
SET_VAR_ADD ()	97
SET_VAR_BLOCK ()	98
SET_VAR_CHR () SET_VAR_CHRLEN () SET_VAR_LOWER () SET_VAR_UPPER ()	99
SET_VAR_COPY ()	101
SET_VAR_DATE ()	102
SET_VAR_LOG ()	103
SET_VAR_MACRO ()	104
SET_VAR_NIL ()	105
SET_VAR_NUM () SET_VAR_NUMDECI () SET_VAR_INT ()	106
SET_VAR_SCR ()	108
SET_VAR_SPECIAL ()	109
UDF_DECL ()	111
UDF_EXEC ()	113
UDF_PROT ()	115
VAR_ARRELEM ()	116
VAR_BLOCK ()	118
VAR_BLOCK_COMPILE ()	119
VAR_BLOCK_EVAL ()	120
VAR_CHR ()	121
VAR_DATE ()	123
VAR_DELETE VAR_DEL_ONEVAR()	124
VAR_DEL_MARK() VAR_DEL_SINCE_MARK()	126
VAR_ISDECI ()	127
VAR_ISDIM ()	128
VAR_ISFLDPOS () VAR_ISFLDWA ()	129
VAR_ISLEN ()	131

VAR_ISMODE ().....	132
VAR_ISTYPE ().....	134
VAR_LOG ().....	136
VAR_NAME_FIELD () VAR_NAME_MEMVAR ().....	137
VAR_NAME_MEMVAR_BYNAME_OR_NEW()	
VAR_NAME_MEMVAR_BYNAME_OR_NULL()	
VAR_NAME_FIELD_BYNAME_OR_ERR(cName)	
VAR_NAME_FIELD_BYNAME_OR_NULL(cName).....	139
VAR_NAME_LOCAL () VAR_NAME_LOCPAR ().....	141
VAR_NAME_STATIC ().....	143
VAR_NEW VAR_NEW_STATIC VAR_NEW_ARGS ().....	145
VAR_NEW_ARRAY () VAR_NEW_STATIC_ARRAY ().....	147
VAR_NEW_COPY () VAR_NEW_STATIC_COPY ().....	149
VAR_NUM () VAR_INT () VAR_INTNUM() VAR_FPNUM ().....	150
VAR_OBJ ().....	152
VAR_SCR ().....	154
VAR_SPECIAL ().....	156
5. Inter-Process communication	158
Index.....	162

1. Using the C Interface

The open architecture of FlagShip uses different levels of API (application program interfaces) which are connected to the C language:

- The Extend C System, almost compatible to Clipper 5.x and Summer'87,
- C lines specified directly within regular .prg code, using the #Cinline directive for Open C API,
- Invoking any internal or external C function using the CALL command.
- Invoking standard functions from the FlagShip library with any C program.
- Modifying intermediate C code produced by the FlagShip compiler.
- Providing your own main() module.

The first is the most common way of including C code into a FlagShip application. It is also suitable for C programmers of little and medium experience. The Extend System includes several checking mechanisms for avoiding mistakes and also has direct access to FlagShip variables.

The C code included allows an experienced programmer to code short program sequences directly into the .prg file and to call other C functions and libraries directly. Access to FlagShip variables and functions is possible.

Modifying the C code produced is not very common, but is possible for very experienced C programmers. Such a modified code will lose its high level of compatibility, but it is possible to port to other UNIX systems can be done on the C FlagShip level.

- **Warning:** Programming in the C language allows you nearly unlimited access to the whole UNIX system. Therefore, it requires a high level of programming discipline to avoid a system or application crash, as compared to the easy, high-level programming and "learning by doing" when using the FlagShip (xBASE) language.

A general discussion of C programming is beyond the scope of this manual. For more information on C programming, refer to your C documentation. As noted above, it is easy to crash the application (or in an extreme case the whole system) by incorrect usage of the C language. The most common programmer errors which are not detected by the C compiler are:

Uninitialized variable: The C compiler does not initialize local variables by default, except when assigning a value to them or when using the static declaration. An uninitialized variable may take any random, current value from the RAM storage. An unpredictable result, segmentation violation, or bus error may occur.

Uninitialized pointer: is a special case of an uninitialized variable. In most cases, a segmentation violation or bus error will occur when accessing memory via that uninitialized pointer. Assigning a value to an uninitialized pointer will override a random part of the current RAM storage (e.g. valid FlagShip variables) which may result in an error (or segmentation) occurring later, e.g. being returned to the regular FlagShip application.

String and array handling: In most cases strings in C are terminated by zero byte `\0`. When declaring a variable, do not forget to add one byte to the required string length. Also note that the string (or array) index starts with 0 and runs to `strlen()-1`. Overwriting the declared string (or array) by only one byte more than declared will often overwrite another variable, will cause a segmentation or will destroy the heap handler, when declared by `malloc`.

Pointer handling: Using the original FlagShip values passed by value to your UDF is valid only during the execution of that UDF. Assigning this pointer to a static variable of the UDF may subsequently crash your application. The safe way is to copy the passed variable contents to the local (or static) variables using `strcpy()`, `strncpy()` etc.

Returning pointer to a local C variable: since the contents of a local variable is undefined when returning from a UDF, such a pointer will subsequently point to some random RAM area, unless the Extend System is used (see `_retc()` function).

Using malloc and free: Always check whether allocating heap was successful. Do not use a `malloc` pointer after freeing it.

The number and type of the parameters passed to and retrieved in function must always match, except when using the dynamic parameter passing of the FlagShip Extend System.

Wrong usage of system functions: some of the UNIX system functions (like `putenv()` etc.) require the parameter to be passed by reference (by pointer). Be careful when using the original parameters passed by value from FlagShip to your UDF, since this temporary copy will be destroyed on returning from the UDF. In such a case it is safer way to use a static declaration.

Assignment not comparison, as "if (x = 5)" instead of `==` or `&` instead of `&&` etc. This is generally not fatal but hard to localize.

Used directives and pragmas (like `#include`, `#ifdef` etc.) must start at the first column with most C compilers. This error is usually detected by the C compiler, which reports undeclared variables etc.

Check and use the explicit prototyping, instead of allowing the C compiler to do the default type conversion. There are differences between DOS and UNIX parameter and function types with some standard C functions. Observe warnings displayed by the C compiler (e.g. `-Wc,-W2` switch in SCO for the warning level 2).

2. Extend C API System

The FlagShip Extend System allows the user to use his specific C program as a common UDF, with access to FlagShip variables for in/output. Using the Extend System is the safest way to include C programs into the whole application, because of the default checking of passed and returned parameters.

FlagShip Extend C programs are common C source files with the .c extension. The programs may be pre-tested on the C level, e.g. using a symbolic debugger of the UNIX system. The ready-to-run C programs will be compiled by cc or FlagShip and simply linked together with the rest of the application.

The Extend C function is called from the .prg in the same way as any usual user-defined function UDF written in the FlagShip language. Access to the FlagShip system from the C program is done by parameter passing and by using the prepared exchange routines and macros included in the <FlagShip_dir>/include/FSextend.h file.

2.1 Structure of Extend Programs

Compared to a stand-alone C program, very few rules or modifications are necessary:

1. At the beginning of the C program file, include the FlagShip's extend file **FSextend.h**, which also automatically includes the FlagShip.h file. IN addition include files may be used, depending on the application context.
2. Use the macro **FSudfname(fn_name)** instead of the common function declaration in C by means of **fn_name()**. For parameter passing, see paragraph 4 to 7.
3. After internal variable definitions, but before the first executable statement, call the **FSinit()** function to receive the parameters from your FlagShip application and initialize the return stack.

4. Check the incoming parameters for type and validity using

Function/Macro	Returns	Description
PCOUNT	(int)	count of parameters passed
_parinfo (nr)	(int)	type of the parameter <nr> = or-ed constant CHARACTER, MEMO, NUMERIC, LOGICAL, DATE, SCREEN, OBJECT, ARRAY, UNDEF, EXT_ERROR
_parinfo (nr, e [...])	(int)	type of array element <e>, see _parinfo() above
_parclen (nr [,e,...])	(int)	current length of string variable or of array element <e>
_parcsiz (nr [,e,...])	(int)	total allocated length of string variable or of array element <e>
ALENGTH (nr)	(int)	determines the array length
ISARRAY (nr)	(int)	is the argument of type "A" ?
ISBYREF (nr)	(int)	is the argument passed by reference?
ISCHAR (nr)	(int)	is the argument of type "C" ?
ISDATE (nr)	(int)	is the argument of type "D" ?

ISLOG (nr)	(int)	is the argument of type "L" ?
ISMEMO (nr)	(int)	is the argument memo field ?
ISNUM (nr)	(int)	is the argument of type "N" ?
ISOBJECT (nr)	(int)	is the argument of type "O" ?
ISSCREEN (nr)	(int)	is the argument of type "S" ?

The parameter number <nr> and array index <e> (or <e1>, <e2>, <e3> for a three-dimensional array etc.) are of type int.

- Receive the parameters passed from the .prg into the C UDF using the Extend System functions

Function	Returns	Passed FlagShip value
_parc (nr [,e,...])	(char*)	chars, strings, memo
_parl (nr [,e,...])	(int)	logical (0 or 1)
_pards (nr [,e,...])	(char*)	date to "YYYYMMDD" format
_parnl (nr [,e,...])	(long)	number
_parni (nr [,e,...])	(int)	number
_parnd (nr [,e,...])	(double)	number
_parscw(nr [,e,...])	(WINDOW*)	screen

The parameter number <nr> and array index <e> (or <e1>, <e2>, <e3> for a three-dimensional array etc.) are of type int.

- Return your result from the C function to the FlagShip application using

Function	Resulting FlagShip value
_ret ()	NIL
_retc (char*)	chars, string
_retclen (char*, int)	string of specif. length
_retl (int)	logical
_retds (char*)	date from "YYYYMMDD" format
_retnl (long)	number
_retni (int)	number
_retnd (double)	number
_retscw (WINDOW*)	screen

- To store values directly into FlagShip variables which are passed by reference when the UDF is invoked, use

Function	Passed FlagShip value
_storc (char*, int [,int,...])	chars, strings
_storclen(char*, int [,int,...])	string of specif. length
_storl (int, int [,int,...])	logical
_stords (char*, int [,int,...])	date
_stornl (long, int [,int,...])	number
_storni (int, int [,int,...])	number
_stornd (double, int [,int,...])	number
_storscw (WINDOW*, int [,int,...])	screen

- For dynamic allocation of string buffers on the heap you may use the C routine **malloc(size)**, but you must copy the heap contents using "_retxxx()" before de-allocation of the space via **free(ptr)**. Instead of malloc() and free() you may prefer to use the

equivalent FlagShip functions **exmgrab(size)** and **exmback(ptr)**, which also perform heap checking.

9. Use "**FSreturn;**" instead of "return;" or instead of the implicit function end with "}" only. The macro transfers the return code back to FlagShip and closes your C routine using the standard "return".
10. For compatibility to DOS (Clipper) C programs, you may use #if... or #ifdef... definitions. See example below.
11. You may use the Open-C API (see EXT.4) within functions declared and initialized by the Extend-C API. But because of the different parameter passing it is not valid reverse, i.e. you may **not** use Extend C API in functions declared by Open-C or by the .prg syntax (i.e. not in Inline-C APIs according to EXT.3).

In the following C sample program, only the lines not enclosed in [] are significant. All others are optional.

```
/** file test.c */

#include <FSextend.h> /* FlagShip header include file */
[#include <other C headers>] /* other include files */
FSudfname (udf_name1) /* puts UDF name in FlagShip list */
{
    [int ...] /* declarati on of C vari ables */
    FSinit(); /* init parameters */
    [var = _par... (); ] /* gets FlagShip parameters into C */
    [statements; ] /* C program body */
    _ret... (); /* puts C vari ables into FlagShip */
    [statements; ] /* other C statements */
    [_ret... (); ] /* transfers other vars into FS */
    [statements; ] /* other C statements */
    [_ret... (); ] /* transfers other vars into FS */
    FSreturn; /* returns into FlagShip program */
}

FSudfname (udf_name2) /* puts UDF name in FlagShip
list */
{
    [int ...] /* declarati on of C vari ables */
    FSinit(); /* init parameters */
    [var = _par... (); ] /* gets FlagShip parameters into C */
    [statements; ] /* C program body */
    _ret... (); /* puts C vari ables into FlagShip */
    FSreturn; /* returns into FlagShip program */
}
/** eof test.c */
```

Example of an UDF to rotate a string left to right:

```
/** File strtrot.c **/ Syntax: new = str_rotat (old) **/
#include <FSextend.h> /* see rule 1 */
#include <malloc.h>
#include <string.h>

FSudfname (str_rotat) /* see rule 2 */
{
    int nn, aa, lng;
    unsigned char *oldstr, *newstr;

    FSinit (); /* see rule 3 */

    if (PCOUNT != 1 ||
        _parinfo (1) != CHARACTER) { /* see rule 4 */
        _retc (""); /* see rule 6 */
        FSreturn; /* see rule 9 */
    }
    oldstr = _parc(1); /* see rule 5 */
    lng = strlen (oldstr);

    if (lng == 0) {
        _retc ("");
        FSreturn;
    }
    newstr = malloc (lng +1); /* see rule 8 */
    if (newstr == NULL) {
        _retl (0); /* see rule 6 */
        FSreturn; /* see rule 9 */
    }
    for (nn = 0, aa = lng -1; aa >= 0; nn++, aa--)
        newstr [nn] = oldstr [aa];
    newstr [nn] = '\0';

    _retc (newstr); /* see rule 6 */
    free (newstr); /* see rule 8 */
    FSreturn; /* see rule 9 */
}
```

This user function "str_rotat" will be called from the FlagShip (or Clipper) application using a standard UDF call:

```
*** test.prg
LOCAL x, y
x = "This is my string"
y = str_rotat (x)
? y                && "gni rts ym si si hT"
```

You may compile both programs at once with:

```
$ FlagShip test.prg strtrot.c -otest
$ test
```

For reverse compatibility to C programs for Clipper's Extend system, you may use definitions (pragmas) such as:

```
#define FLAGSHIP /* see rule 10 */

#include <malloc.h>
#include <string.h>
#ifdef FLAGSHIP /* see rule 10 */
#include <FSextend.h>
FSudfname (str_rotat) /* see rule 2 */
#else
#include <EXTEND.H> /* or: EXTEND.API */
#define FSreturn /* compatib. to FS */
#define FSinit() /* compatib. to FS */
CLIPPER_STR_ROTAT()
#endif
{
  int nn, aa, lng; /* etc, unchanged FlagShip's API */
  :
}
```

Note also the slight differences between the C on DOS and UNIX, especially the int declaration (on UNIX equivalent to long) and the pointer arithmetic.

2.2 Naming Convention

In FlagShip, all UDF names given in the .prg source are translated to lowercase, shortened to 10 significant characters and prefixed with "**_bb_**". Therefore, use only up to 10 characters for C names, given in lower case. The **_bb_** prefix is added to the name by the FSudfname macro. See 2.1.2 and the description in chapter 2.9.

2.3 Access to FlagShip Variables and Arrays

In the Extend System API, all the .prg variables are accessed by parameter passing. The variables may be passed by value (the default) or by reference (the default on arrays, or if the variable is preceded by the @ operator).

Passing values by reference will speed up the execution especially with long strings, since no additional copy need be created. Variables passed by reference may be directly modified in the C function; but be very careful that they are not destroyed. It is safer to change parameter values using **_storxx** functions, as summarized in chapter 2.1.7, since these functions take care of the proper value translation and checking.

Similar to an UDF in FlagShip, the C function may return any valid value by issuing the **_retxx** function. See chapter 2.1.6. When posting more than one return value, only the last one before an FSreturn is effective.

In the .prg file, the UDF is normally invoked, like **my_udf()** or **my_udf(var1, var2)** or **my_udf(@var1, , @var3)**. The Extend System API provides a set of interface functions that allow the C program to access FlagShip parameter values. These functions begin with the

characters `_par` and are summarized in chapter 2.1.5. To check the receiving parameter, additional functions and macros, defined in `FSextend.h` are available. See chapter 2.1.4.

When calling the `_parxx` parameter or `_storxx` interface functions, you **must** ensure the inclusion of the function corresponding to the parameter type received from the `.prg` file. See also example below.

If the `FlagShip` argument is an **array**, additional parameters specify the required element and dimension. For example:

```
LOCAL array1 := {1, 2, 3}
LOCAL array2 := {{4, 5}, {6, 7}, {8, 9}}
? my_udf1 (array1, array2)
? my_udf1 (3, {NIL, NIL, {8, 9}})
? my_udf2 (array1[2], array2[3])
? my_udf2 (2, {8, 9})
```

requires the following simplified C program to receive the correct parameters:

```
FSudfname (my_udf1) {
    int1 := _parni (1, 3); /* access to element array1[3] = 3 */
    int2 := _parni (2, 3, 1); /* access to element array2[3,1] = 8 */
    int3 := _parni (2, 3, 2); /* access to element array2[3,2] = 9 */

    FSudfname (my_udf2) {
        int1 := _parni (1); /* =2, passed = array1 */
        int2 := _parni (2, 1); /* =8, passed = array2[3] */
        int3 := _parni (2, 2); /* =9, passed = array2[3] */
    }
}
```

The `FlagShip's` Extend API system supports multi-dimensional and nested arrays up to ten dimensions or nesting level 10. When invoking the `_parxx` or `_storxx` function for an array argument, all the dimensions (noted as `dimn` in the reference) available **must** be specified, since all dimensions are evaluated until a non-array (or non-object) element is found. You may always terminate the dimension evaluation by specifying 0 or -1 after the last required dimension to avoid an error or a segmentation violation. If more dimensions are specified than are available, the rest are ignored, e.g.:

```
myudf3 ({ {1, 2}, {3, 4}}) // one param as a two-di mens. array
:
FSudfname (my_udf3) {
    int ii, len;
    FSinit();
    len = _parinfa (1); /* error or segmentation */
    len = _parinfa (1, 0); /* 2 (two-di mens. array) */
    ii = _parni (1, 0); /* 0 (error) */
    ii = _parni (1, 2, 2); /* 4 (value) all dims given */
    ii = _parni (1, 2, 2, 0); /* 4 (value) terminated safely */
    ii = _parni (1, 2, 2, 1); /* 4 (value) rest is ignored */
}
```

The `Clipper` compatible Extend-C API does not support creating new arrays in the C function. If you need to do this, use the `Open-C` API, see `EXT.4.5 VAR_NEW_ARRAY()`

In a professional application, you may determine the unknown parameter type by using the `_parinfo()` or `_parinfoa()` function or the associated macros, and the parameter count with the `PCOUNT` macro. To determine e.g. the second parameter of `my_udf(NIL,xx)` where `xx` is a single value or the element [2,3] of passed array, use:

```
#include <FSextend.h>
FSudfname (my_udf) {
    long myvar;
    FSinit();
    if (PCOUNT < 2) {                /* check the count of parameters */
        _retl (0);                  /* return .F. on wrong par count */
        FSreturn;
    }
    if (ISNUM (2))                   /* argument = single value ? */
        myvar = _parnl (2);         /* yes */
    else {
        if (ISARRAY(2) &&           /* argument = 2-dimens- */
            _parinfoa(2, 2, 3) == NUMERIC) /* array and numeric ? */
            myvar = _parnl (2, 2, 3);
        else {
            _retl (-1);             /* return -1 if wrong param type */
            FSreturn;
        }
    }
    _retl (myvar++);                /* params ok, perform UDF action */
    FSreturn;                       /* eg. return argument increment */
}
```

The parameters passed may also represent a valid `FlagShip` object. To access it, use the array access `_parxx` functions and `#define`'s from `<FlagShip_dir>/include/FSextend.h`; for example:

```
*** file test.prg
LOCAL char := "my text ", num := 25
@ 10,20 GET char VALID myCudf(GETACTIVE())
@ 11,20 GET num PICTURE "9999" VALID myCudf(GETACTIVE())
READ
*** eof test.prg

/** file mycudf.c */
#include <FSextend.h>
FSudfname (mycudf)
{
    unsigned char *buff_ptr;
    int has_focus, changed = 0;
    FSinit();
    if (PCOUNT < 1 || !ISOBJECT(1)) { /* on wrong parameters, */
        _retl (0);                  /* return .F. */
        FSreturn;
    }
    buff_ptr = _parc (1, GETOBJ_CARGO); /* see FSextend.h */
    has_focus = _parl (1, GETOBJ_HASFOCUS);

    if (has_focus) {                /* get object active ? */
        if (_parl (1, GETOBJ_CHANGED) {
```

```

        _storc ("changed", 1, GETOBJ_CARGO);
        changed++;
    } else
        _storc ("must be changed", 1, GETOBJ_CARGO);
    }
    _retl (changed != 0);
    FSreturn;
}
/** eof mycudf.c **/

```

Compile: \$ FlagShip test.prg mycudf.c

2.4 Accessing and Modifying String Values

The `_parc()` function receives the pointer of the byte sequences representing the internal string storage on the heap. When passing the string variable by value (the default), FlagShip will create an internal copy of the variable and pass this copy to the UDF. Modifying the incoming parameter by means of `_storc()` function, will therefore affect the variable copy only, not the original. On the other hand, passing the variable by reference will modify the original value using `_storc()` function.

In FlagShip memo variables are the same as the character (string) values.

It is a C convention to end a string with a zero byte. When the `FS_SET("zero")` is activated, FlagShip supports the storage of zero bytes within the string and will pass the whole string length to your UDF. To determine the true string length, use `_parclen()` or `_parcsiz()` instead of the C function `strlen()`, which only determines the length up to the first zero byte. To return an embedded zero byte within the string, `_retclen()` or `_storclen()` should be used instead of `_retc()` or `_storc()`.

When `FS_SET("zero")` is not activated, FlagShip handles strings according to the C convention and will pass (or receive) only the string part up to the first included zero byte, regardless of whether the `_parcxx()`, `_retcxx()` or `_storcxx()` function is used.

In any case, **never** extend the string length, redefine the string pointer value or override the trailing zero byte on string parameters received, but play safe and return the new value by means of the `_retcxx()` function. On the other hand, one can shorten a string by replacing a valid string value with the zero byte.

Keep in mind the different indexing conventions on strings, starting with zero in C, with one in the FlagShip language. In C the last valid string byte (prior the zero byte terminator) is the value of `strlen(x) - 1`, but in a FlagShip program it is `LEN(x)`.

2.5 Access to UNIX/Windows Runtime Libraries

Using the C API, you have unlimited access to C and other UNIX or Windows libraries corresponding to your C and FlagShip version. Make sure to #include the accompanying definition file, as stated in the UNIX man pages or the library documentation. To detect additional warnings the use of the -Wc,-W3 FlagShip compiler switch (OS dependent) is recommended. Example:

```
/** file sinus.c */
/* access by UDF call: value = sinus(degree) */
#include "FSextend.h"
FSudfname (sinus) {
    double degree, retval = 0.0;
    FSi nit();
    if (PCCOUNT == 1 && ISNUM(1)) {
        degree = _parnd (1);
        retval = sin (3.1415926535 * degree / 180.0);
    }
    _retnd (retval);
    FSreturn;
}
/** eof sinus.c */
```

2.6 Using C Input/Output Functions

All file i/o in FlagShip are managed by the standard UNIX i/o functions, such as open(), fopen() etc. Therefore, there are no special restrictions in the use of such i/o or streams i/o within your C API program. Do not use databases (and associated memos and index files) which are already open, since they are internally buffered in FlagShip.

The screen i/o is managed in FlagShip by the curses library in Terminal i/o mode, or by native GUI interface in GUI mode. Therefore, the standard printf() function may work in some cases, but will by-pass the curses and is therefore not managed by it. Use the similar printf() curses function instead, correcting the external int variables _row and _column, if necessary. In GUI mode, this output is sent to console window. The best way is to use FlagShip high level output functions QOUT(), SETPOS(), DISPOUT() etc. For more information refer to chapter 2.7 and section EXT.4.

2.7 Using the Open C Interface

When using the Open C Interface, you may also access any function of the FlagShip library within the Extend C API System. For more information, refer to the section EXT.4.

2.8 Compiling the C programs

Since FlagShip supports the direct use of C files, you may preferably compile the C API program together with all other .prg files, e.g.:

```
$ FlagShip addr*.prg mycudf*.c rest*.o -Maddress
```

Of course, the C program may be compiled separately using the cc compiler and then linked with other files using e.g.

```
$ cc -c mycudf*.c  
$ FlagShip addr*.prg mycudf*.o rest*.o -Maddress
```

2.9 The Extend API System Reference

The following reference of standard interface functions and macros is listed in alphabetical order. For a summary in a logical order, refer to chapter 2.1.

Since this is standard C, all function names, variables and macros used are case-sensitive.

`_parc ()`

Syntax:

```
strPtr = _parc (order [, dim1 [, dim2, ...]] );
```

Purpose:

Returns a pointer to a FlagShip variable or an array element of type "C" received as an argument in a C UDF.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_parc(1,2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<strPtr> is an (unsigned char *) pointer pointing to the first byte of a character variable or string, or "" when an error occurs.

Description:

`_parc ()` function is used for passing the contents of a FlagShip character variable as an argument to the C function. When the parameter is passed by reference, you may modify the original value by using `_storc()`. Passing the FlagShip value by reference will also speed up execution and reduce memory requirements.

Warning: `_parc ()` does **not** make a copy of the contents. Rather it returns a pointer to the character memvar storage. If the parameter is passed by reference, any changes in the string at the address returned from `_parc ()` will directly affect the originating FlagShip variable. Extending the string will cause a fatal error. Refer to the chapters 2.3 and 2.4 for more information.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability, which is specific to FlagShip.

Example 1:

Replace a character variable xyz with another string at a specific position (similar to expanded STRPOKE or STUFF). The UDF is called by `x=my_udf(xyz,2,"aaa")` or `x=my_udf(@xyz,2,"aaa")`. The changes are returned in the UDF return value, and, if the variable was passed by reference, in the variable contents itself as well:

```
#include "FSextend.h"
#define MAXSTR 100
FSudfname (my_udf) {
    unsigned char *par1, *par3, mystr[MAXSTR+1];
    int par2, ii;
    FSinit();
    mystr [0] = 0; /* init null string */
    if (PCOUNT == 3 &&
        (ISCHAR(1) || ISMEMO(1)) &&
        ISNUM (2) && ISCHAR(3)) {
        par1 = _parc(1); /* ptr to target str */
        par2 = _parni(2) -1; /* C like str index */
        if (par2 < strlen(par1) && par2 >= 0) {
            strncpy (mystr, par1, MAXSTR);
            mystr [MAXSTR] = 0; /* if p1 is longer */
            par3 = _parc(3); /* ptr to replacement*/
            for (ii=0; ii < strlen(par2) &&
                (ii+par2) < MAXSTR; ii++)
                mystr[ii+par2] = par2 [ii];
            if (ISBYREF(1) && !ISMEMO(1))
                _storc (mystr, 1); /* replac original */
        }
    }
    _retc (mystr);
    FSreturn;
}
```

Example 2:

Access to the array element xyz[5] or abc[2,3,5] of type "C", called by `x=my_udf(xyz)` or `x=my_udf(@xyz)` or `x=my_udf ({1,2,3,4,"abc",6,7})` or `x=my_udf (abc[2,3])`:

```
#include <FSextend.h>
FSudfname (my_udf) {
    unsigned char mystr [101];
    FSinit();
    mystr [0] = 0; /* init null string */
    if (PCOUNT > 0 && _parinfo(1,5) == CHARACTER) {
        strncpy (mystr, _parc(1,5), 100);
        mystr [100] = 0; /* last valid byte */
        mystr [1] = 'X'; /* 2nd byte */
    }
    _retc (mystr);
    FSreturn;
}
```

Related:

`_parclen()`, `parcsiz()`, `_parinfo()`, `_parinfo()`, `_retc()`, `_storc()`, `ISCHAR()`

`_parclen ()`

`_parcsiz ()`

Syntax:

```
len1 = _parclen (order [, dim1 [, dim2, ...]] );  
len2 = _parcsiz (order [, dim1 [, dim2, ...]] );
```

Purpose:

Returns the length of a FlagShip character string.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimN> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_parc(1,2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<len1> is an (int) value containing the actual length of the data, excluding the trailing zero-byte delimiter or 0 when an error occurs.

<len2> is an (int) value containing the actual length of the data, including the trailing zero-byte delimiter or 0 when an error occurs.

Description:

`_parclen()` and `_parcsiz()` return the length of a character value received as a parameter from FlagShip in a C function.

`_parclen()` does not include the zero-byte terminator in the logical length. Embedded zero bytes are supported according to the state of `FS_SET("zero")`, as the `LEN()` function.

The `_parcsiz()` return value always includes all embedded and trailing zero bytes.

Include:

<FlagShip_dir>/include/FSextend.h

Example:

```
char string [20];  
FSinit ();  
if (PCOUNT > 0 && ISCHAR(1) && _parclen(1) < 20)  
    strcpy (string, _parc(1));  
else { ...
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability specific to FlagShip.

Related:

`_parc()`, `_parinfo()`, `_parinfo()`, `FS_SET()`

`_pards ()`

Syntax:

```
strPtr = _pards (order [, dim1 [, dim2, ...]] );
```

Purpose:

Returns a pointer to the string representation of a FlagShip type "D" (date) variable received as a string.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_pards(1,2,3, 4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<strPtr> is an (unsigned char *) pointer to the first byte of the string representing the date value, or "" when an error occurs.

Description:

`_pards ()` converts an internal FlagShip date value into a string of the form "YYYYMMDD". `_pards ()` returns a pointer to an internal static char array, so the next call will overwrite the old string. Therefore, when several date parameters are passed, you need to make a copy of each of the strings.

You may use `_parnl()` for validation of date; a valid date is > 0L.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
char date[9]; /* call: myUDF(i, date()) */
FSinit();
if (_parinfo(2) == DATE) { /* if argument correct: */
    if(_parnl(2) <= 0L) /* date validation */
        date[0] = 0; /* = invalid date -> "" */
    else
        strcpy (date, _pards(2)); /* get it from FS program */
} else
    strcpy (date, "19000101"); /* else my default value */
date [1] = '8'; /* makes date 19th century*/
_retds (date); /* and return it as */
FSreturn; /* FlagShip "date" variab. */
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability specific to FlagShip.

Related:

`_parinfa()`, `_parinfo()`, `_retds()`, `_stords()`

`_parinfo ()`

Syntax:

```
type = _parinfo (order, dim1 [, dim2, ...] );
```

Purpose:

Returns the type of an array element

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_parinfo(1,2, 3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Supplying -1 for <dim> will stop further array evaluation, a 0 will deliver the size of the relevant dimension. Up to 10 dimensions are supported by the Extend API.

Returns:

<type> is a (int) value containing the FlagShip type of the required parameter:

Constant in FSextend.h	Description
ARRAY	the argument element is an array
BLOCK	the argument element is a code block
CHARACTER	the argument element is a character
MEMO	the argument elem. is a memo field
DATE	the argument element is a date type
LOGICAL	the argument element is a logical
NUMERIC	the argument element is a number
OBJECT	the argument element is an object
SCREEN	the argument elem. is a screen var
UNDEF	the argument element is NIL
ERROR	illegal parameters (UNIX only, for backward compat.)
EXT_ERROR	illegal parameters, same as ERROR (Windows and UNIX)

Description:

`_parinfo ()` returns an integer code for the type of an array element. The codes are defined in `FSextend.h`. `_parinfo (order, 0)` returns the size of the array (first dimension).

Include:

<FlagShip_dir>/include/FSextend.h

Example:

```
myudf (NIL, {{1, 2}, {3, 4, 5}}) // two parameters
:
FSudfname(myudf) {
  int code, input;
  FSinit();
  code = _parinfo(2    ); /* error or segmentation */
  code = _parinfo(2, -1 ); /* ARRAY          */
  code = _parinfo(2, 0  ); /* 2 (length of 1st dim) */
  code = _parinfo(2, 2  ); /* error or segmentation */
  code = _parinfo(2, 2, -1); /* ARRAY          */
  code = _parinfo(2, 2, 0); /* 3 (length of 2nd dim) */
  code = _parinfo(2, 2, 3); /* NUMERIC       */
  code = _parinfo(2, 2, 3, 4); /* NUMERIC (4 is ignored) */
  code = _parinfo(2, 2, 5, 4); /* EXT_ERROR (5 is illegal) */

  if (code != NUMERIC) { /* awaiting: [1, 2] =number */
    _retl(0); FSreturn; } /* if not, returns FALSE */
  input = _parinfo(2, 1, 2); /* otherwise get it into C */
}
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability specific to FlagShip.

Related:

`_parinfo()`, `ALENGTH()`

`_parinfo ()`

Syntax:

```
type = _parinfo (order);
```

Purpose:

Returns the number of arguments or the type of a from FlagShip parameter passed.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

When <order> is zero, the position of last parameter passed from FlagShip (starting with 1, if any) is returned. This is equivalent to the PCOUNT macro.

Returns:

<type> is an (int) value containing the type of the required parameter, when <order> is greater than zero. Note that the return value may contain a combination of the code MPTR and e.g. NUMERIC:

Constant in FSextend.h	Description
ARRAY	the argument is an array
BLOCK	the argument element is a code block
CHARACTER	the argument is a character
DATE	the argument is a date type
LOGICAL	the argument is a logical
MEMO	the argument is a memo field
MPTR	the argument is passed by reference
NUMERIC	the argument is a number
OBJECT	the argument element is an object
UNDEF	the argument is NIL
SCREEN	the argument is a screen var
ERROR	illegal parameters (UNIX only, for backward compat.)
EXT_ERROR	illegal parameters, same as ERROR (Windows and UNIX)

Description:

Depending on the invocation, `_parinfo(0)` returns the number of parameters passed, or `_parinfo(n)` returns an integer code representing the argument type. The codes are defined in `FSextend.h`.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
#include "FExtend.h"
FSudfname (myudf) {
    int code, input;          /* call: myUDF ([num[,x]]) */
    FSinit ();               /* init parameters from FS */
    input = -1;              /* my default here */
    if (_parinfo (0) >= 1) {
        code = _parinfo(1);  /* if 1st parameter given */
        if (code == NUMERIC) /* and its type is ok: */
            input = _parni (1); /* get the 1st argument */
    }
    _ret ();                 /* return NIL */
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability specific to FlagShip.

Related:

_parclen(), parcsiz(), _parinfo(), PCOUNT, ISARRAY(), ISCHAR(), ISDATE(), ISLOG(), ISMEMO(), ISNUM(), ISSCREEN()

_parl ()

Syntax:

```
val = _parl (order [, dim1 [, dim2, ...]] );
```

Purpose:

Retrieves a logical parameter as an (int) value.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. _parl(1,2,3,4) to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<val> is an (int) value containing zero for the FlagShip FALSE argument or one for the FlagShip TRUE argument.

Description:

_parl () is used for retrieving the value from a FlagShip logical variable and is equivalent to the macro ISLOG() for single variables.

Include:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include "FSextend.h"
FSudfname (myudf) {
    int logval;
    FSinit ();
    logval = _parl (3, 12);
    :
    _ret ();
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability, which is specific to FlagShip.

Related:

_ret(), _retl(), _storl(), ISLOG()

`_parnd ()`

Syntax:

```
val = _parnd (order [, dim1 [, dim2, ...]] );
```

Purpose:

Retrieves a floating point (double) value from a FlagShip numeric variable.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_parnd(1,2,3, 4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<val> is a (double) value containing the FlagShip numeric parameter.

Description:

`_parnd ()` is used for retrieving the floating point value from a FlagShip numeric variable.

Include:

<FlagShip_dir>/include/FSextend.h (includes also math.h)

Example:

```
#include "FSextend.h"
FSudfname (myudf) {
    double numvar;           /* variable arguments allow*/
    FSinit();               /* myUDF (x, num) */
    if (ISNUM (2))          /* or myUDF (x, n[10]) */
        numvar = _parnd (2); /* or myUDF (x, n[10,10]) */
    else
        if (ISARRAY (2)) {
            if (_parinfo(2, 4) == NUMERIC)
                numvar = _parnd (2, 4); /* get: n[4] */
            else
                numvar = _parnd (2, 3, 5); /* get: n[3,5] */
        }
    :
}
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability, which is specific to FlagShip.

Related:

`_parinfo()`, `_parni()`, `_parnl()`, `_retnd()`, `_stornd()`, `ISNUM()`

`_parni ()` **`_parnl ()`**

Syntax:

```
val = _parni (order [, dim1 [, dim2, ...]] );  
val = _parnl (order [, dim1 [, dim2, ...]] );
```

Purpose:

Retrieves a (long) integer value from a FlagShip numeric variable.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_parni(1,2,3, 4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<val> is an (int) or a (long) value containing the FlagShip numeric parameter.

Description:

`_parni()` and `_parnl()` are used for retrieving the integer value from a FlagShip numeric variable. Note: on UNIX, the int is equivalent to long int. Therefore, the behavior of `_parni()` is equivalent to that of the `_parnl()` function.

In addition to numeric values, `_parnl()` also support date values. It can be used for validation of date; a valid date is > 0L.

Include: <FlagShip_dir>/include/FSextend.h

Example:

```
#include "FSextend.h"  
FSudfname (myudf) {  
    int  num1;          /* call : x[12, 17] = 123 */  
    long num2;          /* e.g. : myUDF (x, 25.51) */  
    FSinit();  
    num1 = _parni (1, 12, 27); /* get : x[12, 27] = 123 */  
    num2 = _parnl (2);         /* get : param. 2 = 25 */  
    _retni (num1 * num2);     /* return the product of */  
    FSreturn;  
}
```

Compatibility:

Compatible to C5 API and C87 Ext.System, except for the extended array access ability, which is specific to FlagShip.

Related: `parinfo()`, `_parnd()`, `_retni()`, `_retnl()`, `_storni()`, `ISNUM()`

`_parscw ()`

Syntax:

```
winPtr = _parscw (order [, dim1 [, dim2, ...]] );
```

Purpose:

Returns a copy of the curses WINDOW structure from a FlagShip "screen" variable.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To access a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_parscw(1,2, 3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<winPtr> is a WINDOW * pointer from a FlagShip "screen" variable containing the screen content, created by e.g. SAVE SCREEN TO or SAVESCREEN().

Description:

To manipulate or create a new FlagShip screen variable. You may convert the screen type into a character string and vice versa by using the FlagShip SCREEN2CHR() and CHR2SCREEN() functions.

When the parameter is passed by reference, you may modify the original screen value.

Include: <FlagShip_dir>/include/FSextend.h, includes also curses.h

Example:

see also a large example in `_retscw()` and in the section EXT.4

```
#i ncl ude <FSextend. h>
FSudfname (myudf) {          /* call: scr2 = myUDF(scr1) */
    WI NDOW * screen;
    FSi ni t ();
    screen = _parscw (1);    /* get 1st parameter      */
    :                       /* modi fy screen contents */
    _retscw (screen);       /* return changed screen  */
    FSreturn;               /* return to FlagShi p    */
}
```

Compatibility

Not available in Clipper. Because the WINDOW structure in curses.h is often UNIX version dependent, it is recommended that only the defined, portable curses functions are used.

Related:

`_parinfo()`, `_retscw()`, `ISSCREEN()`

`_ret ()`

Syntax:

`_ret ();`

Purpose:

Puts a NIL into the FlagShip return value.

Arguments:

none

Returns:

FlagShip NIL value is posted onto the return stack. The function itself does not return a value.

Description:

`_ret ()` is used to return a NIL value to FlagShip.

Include:

`<FlagShip_dir>/include/FSextend.h`

Example:

```
#include <FSextend.h>
FSudfname (myudf)                /* call: myUDF (num) */
{
    int par1;
    FSinit();
    _ret();                        /* return NIL */
    if (PCOUNT > 0) {             /* if wrong parameters */
        par1 = _parni (1);        /* else: compute ... */
        _retni (par1);           /* and return number */
    }
    FSreturn;                     /* back to FlagShip */
}
```

Compatibility

Compatible to C5 API and C87 Extend System. In FS3 and C87, the function returns a FALSE value to the caller.

Related:

`_retl()`, `FSreturn`, `_parinfo()`

`_retc ()` **`_retclen ()`**

Syntax:

```
_retc (strPtr);  
_retclen (strPtr, length);
```

Purpose:

Copies the contents of a string into the FlagShip UDF "character" return value.

Arguments:

<strPtr> is a (char *) or an (unsigned char *) pointer to the zero-byte terminated text which is to be returned by `_retc()` or to any string returned by `_retclen()`.

<length> is an (int) length to be allocated for the character variable contents, which includes the `\0` delimiter. If there is no zero terminator, it will be added.

Returns:

A FlagShip "character" variable is posted onto the return stack.

Description:

Since `strlen(strPtr)` will be returned by `_retc()`, unpredictable results may occur if the string is not zero-byte terminated. In such cases or when embedded zero bytes should be included and `FS_SET("zero")` is active the use of `_retclen()` is recommended.

Include:

<FlagShip_dir>/include/FSextend.h and /usr/include/string.h

Example:

```
FSudfname (myudf) { /* call: str = myUDF() */  
  char string[200]; *strheap;  
  FSinit();  
  strcpy (string, "This is a long string");  
  _retc (string); /* ? LEN(myUDF()) = 21 */  
  strcpy (string, "1234567890");  
  _retc (string); /* ? myUDF() "1234567890" */  
  _retclen (string, 7); /* ? myUDF() "1234567" */  
  string[0] = '-';  
  _retc (string); /* ? myUDF() "-234567890" */  
  strheap = malloc (50); /* 1. allocate UNIX heap */  
  strcpy (strheap, "my string on heap");  
  _retc (strheap); /* 2. copy string to FlagShip */  
  free (strheap); /* 3. free heap before return */  
  FSreturn; /* return to FlagShip */  
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

`_parinfo()`, `_parclen()`, `_storc()`, `FSreturn`, `FS_SET("zero")`

`_retds ()`

Syntax:

```
_retds (dateStrPtr);
```

Purpose:

Copies the contents of a date string into the FlagShip UDF "date" return value.

Arguments:

<dateStrPtr> is a (char *) pointer pointing to the text holding the date equivalent.

Returns:

A FlagShip "date" variable is posted onto the return stack. The function itself does not return a value.

Description:

`_retds()` converts the string supplied into a FlagShip date value. The zero-byte terminated string must be formed as "YYYYMMDD". Otherwise a null-date is returned.

Include:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include "FSextend.h"
FSudfname (myudf) {
    char string[9];
    FSinit();
    string = "19940530";
    _retds(string);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

`_pards()`, `_stords()`, `FSreturn`

`_retl ()`

Syntax:

```
_retl (bool);
```

Purpose:

Puts an "int" value into the FlagShip UDF "logical" return value.

Arguments:

<bool> is an (int) value representing the boolean value. A zero is interpreted as FALSE, any other value as TRUE.

Returns:

A FlagShip "logical" variable is posted onto the return stack. The function itself does not return a value.

Description:

`_retl()` converts the integer value supplied into a FlagShip logical value.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
#include "FSextend.h"
FSudfname (myudf) {
    FSinit();
    if (PCOUNT > 0 && ISNUM(1))
        _retl (1);
    else
        _retl (0);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

```
_par(), _storl(), FSreturn
```

`_retn d ()`

Syntax:

```
_retn d ( num );
```

Purpose:

Puts a "double" value into the FlagShip UDF "numeric" return value.

Arguments:

<num> is a (double) float value to be returned into FlagShip.

Returns:

A FlagShip "numeric" variable is posted onto the return stack. The function itself does not return a value.

Description:

`_retn d ()` returns a correct "numeric" value to FlagShip, using a C floating point value.

Include:

<FlagShip_dir>/include/FSextend.h (includes also math.h)

Example:

```
#include "FSextend.h"
FSudfname (pi) {
    FSinit();
    _retn d(M_PI);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

`_parnd()`, `_retn i()`, `_retn l()`, `_stornd()`, `FSreturn`

`_retni ()`

`_retnl ()`

Syntax:

```
_retni (num);  
_retnl (num);
```

Purpose:

Puts an "int" or "long" value into the FlagShip UDF "numeric" return value.

Arguments:

<num> is an (int) or (long) integer containing a numerical value.

Returns:

A FlagShip "numeric" variable is posted onto the return stack. The functions themselves do not return a value.

Description:

`_retni ()` is used to return a correct value to FlagShip. Note: on UNIX, the int is equivalent to long int. Therefore, the behavior of `_retni()` is equivalent to `_retnl()`.

Include:

<FlagShip_dir>/include/FSextend.h (includes also math.h)

Example:

```
#include "FSextend.h"  
FSudfname (myudf) { /* call: myUDF() */  
    FSinit();  
    _retni (456);  
    FSreturn;  
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

`_parni()`, `_storni()`, `_retnl()`, `FSreturn`

`_retscw ()`

Syntax:

`_retscw (screen);`

Purpose:

Copies a WINDOW structure into the FlagShip UDF "screen" return value.

Arguments:

<screen> is a WINDOW * pointer to curses structure.

Returns:

A FlagShip "screen" variable is posted onto the return stack. The function itself does not return a value.

Description:

`_retscw()` is used to manipulate or create a new FlagShip screen variable. You may convert the screen type into a character string and vice versa by using the FlagShip `SCREEN2CHR()` and `CHR2SCREEN()` functions.

Include:

<FlagShip_dir>/include/FSextend.h (also includes curses.h)

Example 1:

Due to relatively complicated handling with curses, we include here a small and simple program to manipulate screen attributes and colors (applicable in terminal i/o mode):

```
**** File TEST.PRG
*
* change lower to upper cases and blink them
* -----
SETCOLOR ("GR+/B")
c = 47
FOR y = 0 TO 23                && fill screen with
  FOR x = 0 TO 79              && '0'...'A'...'z'
    c = IF (c > 122, 48, c+1)
    @ y, x SAY CHR(c)
  NEXT
NEXT
@ 9,9 TO 21,71
scr1 = SAVESCREEN (10,10, 20,70) && save window
scr2 = myUDF1 (scr1)           && change it

RESTSCREEN (10,10, 20,70, scr1) && display orig
SETCOLOR ("R+/B")
@ 24,0 SAY "restored the original scr1, press any key..."
INKEY(0)

@ 10,10 CLEAR TO 20,70
RESTSCREEN (10,10, 20,70, scr2) && display changed
@ 24,0 SAY "restored the CHANGED scr2, press any key..."
INKEY(0)
```

```

* change colors in line 2+3, attributes in line 8
* -----
SETCOLOR ("w+/B")
CLEAR
c = 23
FOR y = 0 TO 23                                && fill screen with
  FOR x = 0 TO 79                              && chr(24)...chr(255)
    c = IF (c > 255, 24, c+1)
    @ y,x SAY chr(c)
  NEXT
NEXT

@ 9,9 TO 21,71 DOUBLE
scr1 = SAVESCREEN (10,10, 20,70)              && save screen
@ 10,10 CLEAR TO 20,70                        && and create
scr2 = scr1                                    && new screen

count1 = 45
count2 = 8
FOR jj = 2 TO 3                                && change lines 2+3
  FOR ii = 0 TO 60                             && with diverse colors
    bcolor = INT(count1 / 9) % 9
    fcolor = (count2 + 1) % 9
    scr2 = myUDF2 (ii, jj, scr2, fcolor, bcolor, 0)
    count1 = count1 + 1
    count2 = count2 + 1
  NEXT
  RESTSCREEN (10,10, 20,70, scr2)             && output it
NEXT
FOR ii = 0 TO 60                              && change attributes
  attr = ii % 5
  scr2 = myUDF2 (ii, 8, scr2, 3, 5, attr)
NEXT

RESTSCREEN (10,10, 20,70, scr2)               && output creen 2
SETCOLOR ("R+/B")
@ 24,0 SAY "restored the CHANGED scr2, press any key..."
INKEY(0)
RESTSCREEN (10,10, 20,70, scr1)              && output creen 1
@ 24,0 SAY "restored the original scr1, press any key..."
INKEY(0)
QUIT
**** eof TEST. PRG

Compile the above and following programs with:
$ FlagShip test.prg testc.c -otest

```

Example 2:

MYUDF1: Manipulates the contents of the standard input screen in terminal i/o mode: all lowercase letters should be translated into upper case. The change occurs only if character is in the normal character set range, not for alternate or protect modes. Reason: depending on the FSchrmap.def and TERM it may be that e.g. chr(218) is displayed as "c" in the alternate mode .

```

/**** file testc.c ****/
#include <FStxtend.h>
/* #include <curses.h> */

FSudfname (myudf1) {          /* FS: scr2= myUDF1(scr1) */
    WINDOW * screen;
    int x, y, ch;
    chtype chwin, attrib;
    FSinit();
    screen = _parscw (1);      /* get 1st parameter */
    for (y = 0; y < LINES; y++) {
        for (x = 0; x < COLS; x++) {
            chwin = mwinch (screen, y, x); /* char + attr*/
            attrib = chwin & A_ATTRIBUTES; /* attribute */
            ch = chwin & A_CHARTEXT; /* char only */
            if (((attrib & A_ALTCHARSET) == 0L) &&
                ((attrib & A_PROTECT) == 0L) &&
                (ch >= 'a') && (ch <= 'z')) {
                ch = ch - 32; /* upper case */
                attrib = attrib | A_BLINK | A_REVERSE;
                chwin = attrib | ch; /* add attrib */
                waddch (screen, chwin); /* move to win*/
            }
        }
    }
    _retscw (screen); /* put screen to ret var */
    FSreturn; /* return to FlagShip */
}

```

Example 3:

MYUDF2: Manipulates colors of the standard input screen on user defined coordinates (relative to the saved screen). Info: the color pairs are already initialized by the FlagShip startup routine.

```

/* -----
call: str2 = myUDF2 (x, y, scr1, f, b, attr)
args: x,y coordinates (0..n), relative to scr1
      scr1 saved "screen" variable
      f,b color for foreground and background:
          0: unchanged, 1: black, 2: red, 3: green,
          4: yellow, 5: blue, 6: magenta, 7: cyan, 8: white
      attr attributes for foreground, may be added
          together using ((attr1 * 10) + attr2) * 10 +
          attr3 ...
          0: normal, 1: intensive, 2: underline,
          3: reverse, 4: blink, 5: invisible
----- */
#include "FStxtend.h"

FSudfname (myudf2) {
#define MAXCOLORS 8
    WINDOW * screen;
    int x, y, fcol, bcol, attr, a, ch, pair;
    chtype chwin, attrib;
    int f, b;
    FSinit();
}

```

```

x      = _parni (1);          /* get 1. parameter */
y      = _parni (2);          /* get 2. parameter */
screen = _parscw (3);         /* get 3. parameter */
fcol   = _parni (4) - 1;     /* get 4. parameter */
bcol   = _parni (5) - 1;     /* get 5. parameter */
attr   = _parni (6);         /* get 6. parameter */
chwin  = mvwinch (screen, y, x); /* char + attributes */
attrib = chwin & A_ATTRIBUTES; /* attributes only */
attrib = attrib & ~A_COLOR;    /* attrib w/o color */
ch     = chwin & A_CHARTEXT;  /* char only */
if (has_colors()) {
    pair = PAIR_NUMBER(chwin) & 0x3f;
    f = pair / 8;
    b = pair % 8;
    if (fcol < 0 || fcol >= MAXCOLORS) fcol = (int) f;
    if (bcol < 0 || bcol >= MAXCOLORS) bcol = (int) b;
    pair = COLOR_PAIR(fcol * MAXCOLORS + bcol);
    if (attr > 0) attrib = attrib & ~A_BOLD
        & ~A_UNDERLINE & ~A_REVERSE
        & ~A_BLINK & ~A_INVIS; /* disable attrib */
    while (attr > 0) {
        a = attr % 10;
        switch (a) {
            case 1: attrib |= A_BOLD; break;
            case 2: attrib |= A_UNDERLINE; break;
            case 3: attrib |= A_REVERSE; break;
            case 4: attrib |= A_BLINK; break;
            case 5: attrib |= A_INVIS; break;
        }
        attr = attr / 10;
    }
    chwin = attrib | pair | ch;
    waddch (screen, chwin);
}
_retval (screen);          /* put screen to ret var */
FSreturn;                  /* return to FlagShip */
}
/* eof */

```

Compatibility

Not available in Clipper. Because the WINDOW structure in curses.h (terminal i/o) is often UNIX version dependent, it is recommended that only the defined, portable curses functions are used.

Related:

`_parscw()`, `FSreturn`, curses documentation, `printscreen` in EXT.4.8

`_storc ()` **`_storclen ()`**

Syntax:

```
ok = _storc (strPtr, order [, dim1 [, dim2, ...]]);  
ok = _storclen (strPtr, len, order [, dimn]);
```

Purpose:

Assigns a character value to a variable passed by reference using a null terminated string or by using a string with a specific length.

Arguments:

<strPtr> is a (char *) or an (unsigned char *) pointer pointing to the zero-byte terminated text which is to be returned by `_retc()`, or to any string returned by `_retclen()`.

<len> is an (int) length to be allocated for the character variable contents, including the `\0` delimiter. If the zero terminator is not present, it will be added.

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To replace an element of a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_storc(1, 2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<ok> is an (int) value, one (1) if the function is successful. Zero signals an error.

Description:

`_storc()` and `_storclen()` store a character value to a FlagShip variable passed by reference as a parameter. If the parameter specified by <order> is not a variable passed by reference, `_storxxx()` ignores the call and returns a zero.

`_storc()` determines the logical length of the character value by scanning the string supplied for a null terminator byte.

`_storclen()` allows you to specify an explicit logical length for the character value. The string you supply does not need to be null terminated and may include embedded null bytes, if `FS_SET("zero")` is activated.

Both `_storxx()` functions automatically allocate memory and make a copy of the string supplied.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
#include "FSextend.h"
FSudfname(myudf) {          /* call: myvar = "any string" */
    char *heapPtr;         /*      myUDF (@myvar)      */
    int ok = 0;
    FSinit();
    if (PCOUNT > 0 && ISCHAR(1)) {
        heapPtr = malloc (_parclen(1) + strlen("xyz") + 1);
        if (heapPtr != NULL) {
            strcpy (heapPtr, _parc (1));
            strcat (heapPtr, "xyz"); /* add "xyz" to par */
            ok = _storc (heapPtr, 1); /* replace 1st param.*/
            free (heapPtr);
        }
    }
    _retl (ok);              /* .T. on success */
    FSreturn;
}
```

Compatibility:

Compatible to C5 API System, except for the extended array access ability, which is specific to FlagShip.

Related:

`_parc()`, `_parclen()`, `_retc()`, `_retclen()`

`_stords()`

Syntax:

```
ok = _stords (strPtr, order [, dim1 [, dim2...]]);
```

Purpose:

Assigns a date value to a variable passed by reference using a date string.

Arguments:

<strPtr> is a (char *) pointer to the zero-byte terminated text in the format "YYYYMMDD" representing the date value.

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To replace an element of a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_stords(1, 2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<ok> is an (int) value, one (1) if the function is successful. Zero signals an error.

Description:

`_stords()` stores a date value to a FlagShip variable passed by reference as a parameter. If the parameter specified by <order> is not a variable passed by reference, `_stords()` ignores the call and returns a zero.

`_stords()` converts the string supplied into a FlagShip date value. The zero-byte terminated string must be formatted to "YYYYMMDD". Otherwise, a null date is returned.

Include: <FlagShip_dir>/include/FSextend.h

Example:

```
#include "FSextend.h"
FSudfname(myudf) {
    int ok = 0;
    FSinit();
    if (PCOUNT > 0 && ! ISARRAY(1))
        ok = _stords ("19940131", 1);
    _retl (ok);
    FSreturn;
}
```

Compatibility:

Compatible to the C5 API System, except for the extended array access ability, which is specific to FlagShip.

Related:

`_pards()`, `_retds()`, `ISDATE()`, `_parinfo()`

`_storl()`

Syntax:

```
ok = _storl (bool, order [, dim1 [, dim2...]]);
```

Purpose:

Assigns a logical value to a variable passed by reference.

Arguments:

<bool> is an (int) value representing the boolean value. A zero is interpreted as FALSE; any other value as TRUE.

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To replace an element of a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_storl(1, 2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<ok> is an (int) value, one (1) if the function is successful. Zero signals an error.

Description:

`_storl()` converts an integer value and stores it to a FlagShip logical variable passed by reference as a parameter. If the parameter specified by <order> is not a variable passed by reference, `_storl()` ignores the call and returns a zero.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
FSudfname(myudf) {
    int ok = 0;
    FSinit();
    if (PCOUNT > 0 && ! ISARRAY(1))
        ok = _storl (0, 1);          /* 1st param becomes .F. */
    _retl (ok);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API System, except for the extended array access ability, which is specific to FlagShip.

Related:

```
_parl(), _retl(), _parinfo()
```

`_stornd ()`

Syntax:

```
ok = _stornd (num, order [, dim1 [, dim2...]]);
```

Purpose:

Assigns a numeric value to a variable passed by reference using a double value.

Arguments:

<num> is a (double) float value to be returned in a FlagShip variable passed by reference.

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To replace an element of a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_stornd(1, 2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<ok> is an (int) value, one (1) if the function is successful. Zero signals an error.

Description:

`_stornd()` stores a floating point value to a FlagShip numeric variable passed by reference as a parameter. If the parameter specified by <order> is not a variable passed by reference, `_stornd()` ignores the call and returns a zero.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
FSudfname(myudf) {
    int ok = 0;
    FSinit();
    if (PCOUNT > 0 && ! ISARRAY(1))
        ok = _stornd (1.234, 1); /* 1st param is now 1.234 */
    _retl (ok);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API System, except for the extended array access ability, specific to FlagShip.

Related:

```
_parnd(), _retd(), _parinfo(), _storni()
```

`_storni()`

`_stornl()`

Syntax:

```
ok = _storni (num, order [, dim1 [, dim2...]]);  
ok = _stornl (num, order [, dim1 [, dim2...]]);
```

Purpose:

Assigns a numeric value to a variable passed by reference using an integer value.

Arguments:

<num> is an (int) or an (long) integer containing a numerical value.

<order> is an (int) value representing the position of the argument in the argument list starting with one.

Options:

<dimn> is an (int) value selecting the required element in the n-th dimension of the array. To replace an element of a multidimensional or nested array, enter the <dim> values for all the required dimensions, e.g. `_storni(1, 2,3,4)` to access the element [2,3,4] of a three-dimensional array passed as the first parameter. Up to 10 dimensions are supported by the Extend API.

Returns:

<ok> is an (int) value, one (1) if the function is successful. Zero signals an error.

Description:

`_storni()` and `_stornl()` store an integer value to a FlagShip numeric variable passed by reference as a parameter. If the parameter specified by <order> is not a variable passed by reference, they ignore the call and returns a zero.

Note: on UNIX, the int is equivalent to long int. Therefore, `_storni()` is equivalent to `_stornl()`.

Include:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
ok = _storni (1234567890, 1);
```

Compatibility:

Compatible to C5 API System, except for the extended array access ability, which is specific to FlagShip.

Related:

```
_parni(), _parnl(), _retni(), _retnl(), _parinfo()
```

`_xalloc ()` **`_xgrab ()`**

Syntax:

```
anyPtr = _xalloc (len);  
anyPtr = _xgrab (len);
```

Purpose:

`_xalloc()` allocates memory and returns a NULL if not successful.

`_xgrab` allocates memory and generates a run-time error if not successful.

Arguments:

<len> is the (unsigned long) number of bytes to allocate. When using character strings, do not forget to allocate one additional byte for the zero-byte terminator.

Returns:

<anyPtr> is a (void) pointer to the allocated memory, or NULL if the memory requested could not be allocated.

Description:

These functions are similar to `malloc()`, allocating memory on the UNIX heap, but they perform additional checking for a heap which may be destroyed. Memory allocated with `_xalloc()` or by `_xgrab()` must be freed after use with `_xfree()`, not `free()`, and cannot be reallocated by `realloc()`.

Include:

<FlagShip_dir>/include/FSextend.h (including also `malloc.h`)

Example:

```
FSudfname(myudf) {  
    char *strPtr;  
    FSi ni t();  
    _ret();  
    strPtr = _xalloc (strlen("my string") +1);  
    if (strPtr != NULL) {  
        strcpy (strPtr, "my string");  
        _retc (strPtr);  
        _xfree (strPtr);  
    }  
    FSreturn;  
}
```

Compatibility:

Compatible to C5 API System. The function `_xgrab()` is equivalent to `_exmgrab()` of C87.

Related:

`_xfree()`, `_xgrab()`, UNIX: `malloc()`, `free()`, `realloc()`

`_xfree ()`

Syntax:

`_xfree (anyPtr);`

Purpose:

Frees memory allocated by means of `_xalloc()`.

Arguments:

<anyPtr> is a pointer of any type allocated with `_xalloc()` or `_xgrab()`.

Returns:

nothing.

Description:

`_xfree()` releases memory allocated by `_xalloc()` or `_xgrab()`. Note that the same pointer returned by `_xalloc()` or `_xgrab()` must be passed as the argument to `_xfree()`.

Do not use `_xfree()` for memory allocated by `malloc()` or `realloc()`. Use `free()` instead.

Include:

<FlagShip_dir>/include/FSextend.h

Example:

see `_xalloc()` example

Compatibility:

Compatible to C5 API System. The function is equivalent to `_exmback()` of C87.

Related:

`_xalloc()`, `_xgrab()`, UNIX: `malloc()`, `realloc()`, `free()`

ALENGTH ()

Syntax:

```
len = ALENGTH (order)
```

Purpose:

A macro to evaluate the length of a FlagShip single-dimensional array parameter.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<len> is an (int) value representing the number of array elements. See _parinfo()

Description:

ALENGTH(order) is equivalent to _parinfo(order,0). For a multi- dimensional or nested array, use the function _parinfo() instead of the macro.

Prototype:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
FSudfname(myudf) {
    int elements = 0;
    FSinit();
    if ((PCOUNT > 0) && (ISARRAY (1)))
        elements = ALENGTH (1);
    :
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

_parinfo(), _parinfo(), ISARRAY()

ISARRAY ()

Syntax:

status = ISARRAY (order)

Purpose:

A macro to check if argument is a one or a multi-dimensional array.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting by one.

Returns:

<status> is an (int) non-zero value if the parameter is an array, zero otherwise.

Description:

ISARRAY(order) is equivalent to (_parinfo(order) & ARRAY) != 0. For a leaf of a multi-dimensional and nested array, use the _parinfo() function instead of the macro.

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname(myudf) {
    int elements = 0, value = 0;
    FSinit();
    if (ISARRAY (1)) {
        elements = ALENGTH (1);
        if (_parinfo(1, 1) == NUMERIC)
            value = _parni (1, 1);
    } else {
        _ret ();
        FSreturn;
    }
    if (_parinfo(1, 1) == ARRAY) {
        elements *= _parnfa (1, 1);
        if (_parinfo(1, 1, 1) == NUMERIC)
            value = _parni (1, 1, 1);
    }
    _retni (elements);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

_parinfa(), _parinfo(), ALENGTH(), ISCHAR(), ISDATE(), ISLOG(), ISMEMO(), ISNUM(), ISSCREEN()

ISBYREF ()

Syntax:

status = ISBYREF (order)

Purpose:

A macro to check if argument is passed by reference.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is passed from FlagShip by reference, zero otherwise.

Description:

ISBYREF(order) is equivalent to (_parinfo(order) & MPTR) != 0. For a multi-dimensional array, use the _parinfo() function instead of the macro.

Prototype: <FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname(myudf) {
    char *in_str, *temp_str;
    int by_ref = 0;
    FSinit();
    if ((PCOUNT > 0) && (ISCHAR(1)) {
        in_str = _parc(1);
        temp_str = in_str;          /* use orig. copy, but */
        if (by_ref = ISBYREF (1)) { /* don't destroy orig. */
            temp_str = malloc (strlen(in_str) +1);
            if (temp_str)
                strcpy (temp_str, in_str);
            else {
                _ret();          /* error, no memory */
                FSreturn;       /* available */
            }
        }
        if (strlen (temp_str) >= 2)
            temp_str[1] = 'X';   /* replace 2nd byte */
        _retc (temp_str);
        if (by_ref)
            free (temp_str);     /* free allocated mem */
        FSreturn;
    }
}
```

Compatibility: Compatible to C5 API and C87 Extend System

Related:

_parinfa(), _parinfo(), ALENGTH(), ISCHAR(), ISDATE(), ISLOG(), ISMEMO(), ISNUM(), ISSCREEN()

ISCHAR ()

Syntax:

status = ISCHAR (order)

Purpose:

A macro to check if argument is of type "character"

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is of the type "C", zero otherwise.

Description:

ISCHAR(order) is equivalent to (_parinfo(order) & CHARACTER) != 0. For a multi-dimensional array, use the _parinfo() function instead of the macro.

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
#define MYLEN 200
FSudfname(myudf) {
    char input[MYLEN +1];           /* add termin. */
    FSinit();
    if (PCOUNT > 0 && ISCHAR(1) &&
        _parcsiz(1) <= MYLEN)      /* may copy ? */
        strcpy(input, _parc(1));   /* ok, process */
    else {
        input[MYLEN] = 0;          /* terminator */
        if (_parinfo(1) == ARRAY &&
            _parinfo(1,1) == CHARACTER) /* param[1]==C */
            strncpy(input, _parc(1,1), MYLEN); /* cut len */
        else {
            _retl(0);              /* input is */
            FSreturn;              /* unknown or */
            /* too long */
        }
    }
    :                               /* process input */
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

_parinfo(), _parc(), _parclen(), _parcsiz(), _retc(), _storc(), ISARRAY(), ISMEMO(), ISSCREEN()

ISDATE ()

Syntax:

status = ISDATE (order)

Purpose:

A macro to check if argument is of type "date"

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is of the type "D", zero otherwise.

Description:

ISCHAR(order) is equivalent to (_parinfo(order) & DATE) != 0. For a multi-dimensional array, use the _parinfo() function instead of the macro.

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname(myudf) {
    char datum [9];          /* YYYYMMDD + term. */
    FSinit();
    if ((PCOUNT > 0) && (ISDATE (1)))
        strcpy (datum, _pards(1)); /* preserve input */
    else {
        _retl (0);          /* error, no date */
        FSreturn;
    }
    if (datum [3] == '9') { /* i ncrease year */
        datum [2] += 1;    /* by one */
        datum [3] = '0';
    } else                 /* add addit. check */
        datum [3] += 1;    /* for 1999 -> 2000 */
    _retds (datum);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

_parinfo(), _pards(), _retds(), _stords()

ISLOG ()

Syntax:

`status = ISLOG (order)`

Purpose:

A macro to check if argument is of type "logical"

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is of the type "L", zero otherwise.

Description:

ISLOG(order) is equivalent to `(_parinfo(order) & LOGICAL) != 0`. For a multi-dimensional array, use the `_parinfo()` function instead of the macro.

Prototype:

`<FlagShip_dir>/include/FSextend.h`

Example:

```
#include <FSextend.h>
FSudfname(myudf) {
    int true = 0;
    FSinit();
    if ((PCOUNT > 0) && (ISLOG (1))) /* param check */
        true = _parl (1); /* 1 stay for .T. */
    else {
        _retl (0); /* wrong param. */
        FSreturn;
    }
    _retl (true);
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

`_parinfo()`, `_parl()`, `_retl()`, `_storl()`

ISMEMO ()

Syntax:

status = ISMEMO (order)

Purpose:

A macro to check if argument is memo field.

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is of the type "C" and points directly to a memo field, zero otherwise.

Description:

ISMEMO(order) is equivalent to (_parinfo(order) & MEMO) != 0. For a multi-dimensional array, use the _parinfo() function instead of the macro.

Note that a database field is always passed by value to an UDF. Therefore, the equivalent argument type is in most cases CHARACTER. The memo field cannot be placed back directly using _storc(). Store it into a character variable using _retc() and then use REPLACE <memo> WITH myudf (<memo>).

Prototype: <FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname(myudf) {
    int actlen;
    char * buffer;
    FSinit();
    _retc(""); /* assuming error */
    if ((PCOUNT > 0) &&
        (ISCHAR(1) || ISMEMO(1))) {
        actlen = _parclen(1);
        buffer = _xalloc(actlen + 100);
        if (buffer != NULL) {
            strcpy(buffer, _parc(1));
            strcat(buffer, "--my addition--");
            _retc(buffer); /* return new string */
            _xfree(buffer); /* free heap */
        }
    }
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

_parinfo(), _parc(), _parclen(), _retc(), ISCHAR()

ISNUM ()

Syntax:

status = ISNUM (order)

Purpose:

A macro to check if argument is of type "numeric".

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is of the type "N", zero otherwise.

Description:

ISNUM(order) is equivalent to (_parinfo(order) & NUMERIC) != 0. For a multi-dimensional array, use the _parinfo() function instead of the macro.

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname(myudf) {
    int    intvar = -1;
    double dvar   = 0.0;
    FSinit();
    _ret();          /* put NIL on return */
    if (ISNUM (1) )
        intvar = _parni (1);      /* 1st: integer      */
    if (ISNUM (2) )
        dvar = _parnd (2);       /* 2nd: float        */
    if (intvar == -1)
        FSreturn;               /* error, return NIL */
    dvar = dvar * (double) intvar; /* process ...      */
    _retnd (dvar);              /* return float      */
    FSreturn;
}
```

Compatibility:

Compatible to C5 API and C87 Extend System.

Related:

_parinfo(), _parni(), _parnd(), _retni(), _retnd(), _storni(), _stornd()

ISSCREEN ()

Syntax:

`status = ISSCREEN (order)`

Purpose:

A macro to check if argument is of type "screen".

Arguments:

<order> is an (int) value representing the position of the argument in the argument list starting at one.

Returns:

<status> is an (int) non-zero value if the parameter is of the type "S", zero otherwise.

Description:

ISSCREEN(order) is equivalent to `(_parinfo(order) & SCREEN) != 0`. For a multi-dimensional array, use the `_parinfo()` function instead of the macro.

Refer to LNG.2.6.4, `_parscw()` and `_retscw()` for a detailed discussion on screen variables.

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

see example in `_retscw()`

Compatibility:

Available in FlagShip only. Clipper uses character variables to store screen contents. Refer to LNG.2.6.4.

Related:

`_parinfo()`, `_parscw()`, `_retscw()`

FSinit ()

Syntax:

```
FSinit ();
```

Purpose:

A macro to initialize the FlagShip parameter stack within a C UDF.

Arguments:

none.

Returns:

nothing.

Description:

After declaring local variables (if needed), use FSinit(); to set- up the passed parameters from the FlagShip application and the prototypes for return values. The type and value of the parameters may be determined by _par...() functions or IS...() macros, the return values into FlagShip by _ret...() or _stor..() functions.

If there are no parameters passed and no return values assigned, the FSinit() may be omitted. No _par...(), _ret...() and FSreturn functions may be used in such a case. Note that this is an extreme case, as not all C compilers will accept such functions, and if you try to evaluate such a UDF (in TYPE()) or assignment for example), you will get a run time error.

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname (udfname) { /* name of the UDF in lowercase*/
    [int ...;] /* declare local variables used*/
    FSinit(); /* init _parx() and _retx() */
    [xxx = _parx(); ] /* optional: get input vars */
    [statements;] /* body of the UDF */
    _retx(); /* return UDF value */
    FSreturn; /* returns back to FlagShip */
}
```

Compatibility:

Available in FlagShip only. For compatibility to C5 and C87, "#define FSinit();" may be declared, see EXT.2.1.

Related:

_par...(), _ret...(), IS...(), FSreturn

FSudfname ()

Syntax:

```
FSudfname (udfname) { ... }
```

or (old syntax):

```
FlagShip (udfname) { ... }
```

Purpose:

A macro to generate a FlagShip compatible function header.

Arguments:

<udfname> is the declared name of the user-defined function. The name must be given in lowercase, the first character should be a letter, the name may be up to 10 characters long.

Description:

This macro generates the header of the C function to be compatible to FlagShip's UDF calling convention, e.g. FSudfname(myudf) defines the function name `_bb_udfname`, see below. There is no limit to the number of UDFs in one source program.

FlagShip(udfname) is the supported, but obsolete alternative.

Some of the C preprocessors cannot evaluate the `##` template. If so, you may translate this macro using the script "FSexcpp file1.c file2.c ..." delivered with FlagShip, or declare the function header explicitly:

FSudfname(myname) will be translated to:

```
FSvar *_bb_myname ( int argc, FSvar *argv[] )
```

Prototype:

```
<FlagShip_dir>/include/FSextend.h
```

Example:

```
#include <FSextend.h>
FSudfname (test1) {          /* UDF test1 (= _bb_test1) */
    FSinit();                /* init _parx() and _retx() */
    _ret();                  /* return UDF value */
    FSreturn;                /* returns into FlagShip */
}
FSudfname (test2) {          /* UDF test2 (= _bb_test2) */
    FSinit();                /* init _parx() and _retx() */
    _ret();                  /* return UDF value */
    FSreturn;                /* returns into FlagShip */
}
```

Compatibility:

Available in FlagShip only. In C5 and C87, CLIPPER UDFNAME() is used instead. For compatibility, see also section EXT.2.1.

Related:

FSreturn, FSinit

FSreturn

Syntax:

FSreturn;

Purpose:

A macro to exit the C function and return to the FlagShip program.

Description:

This macro generates the usual "return(value)" C statement to exit from a C function back into the caller program. The <value> from the last _retxx() function previously called will be transferred into FlagShip as the UDF return value.

Note:

You may also use the macro "FSend;" instead. If you prefer to use FSreturn() instead of FSreturn, disable the line #define FSreturn in FSextend.h

Prototype:

<FlagShip_dir>/include/FSextend.h

Example:

```
#include <FSextend.h>
FSudfname (udftest)      /* name of the UDF: UDFTEST */
{
    FSinit();             /* init _parx() and _retx() */
    :
    _ret();                /* return UDF value      */
    :
    _retx();               /* return other UDF value */
    :
    FSreturn;             /* returns into FlagShip */
}
```

Compatibility:

Available in FlagShip only. For compatibility to C5 and C87, "#define FSreturn return" may be declared. See also EXT.2.1.

Related:

FSreturn, FSinit, FSxcpp

3. Inline C Programming

The included C code allows an experienced programmer to code program sequences directly into the .prg file and to call other C functions and libraries directly. Access to FlagShip variables and functions is possible.

The inline C programming is usually used as an alternative to the Extend API System (see chapter 2), especially for short program sequences.

3.1 Structure of Inline C Programs

Since the Inline C lines are programmed directly into the .prg source files of FlagShip language, both .prg and C programming rules apply:

1. The block of C program lines starts with the FlagShip preprocessor directive **#Cinline** posted in separate lines and starting at the first column.
2. The C program block is terminated by the FlagShip preprocessor directive **#endCinline** posted in separate lines and starting at the first column. There is no limit to the number of C program blocks within a .prg file and/or in a FlagShip PROCEDURE or FUNCTION.
3. The structure of the C program block must comply with the semantics and syntax of the C language. The FlagShip preprocessor and compiler passes the whole C block (between #Cinline and #endCinline) directly into the .c file produced.
4. All the FlagShip preprocessor directives like #define, #ifdef, #include etc. do not apply in C block and vice versa. If C directives are needed, define them within the C program block.
5. If local C variable declarations are used, the whole C program block must be included in curly brackets { ... }.
6. Static, global or external C variables and all C functions (referred to from C or by the CALL command later on) must be declared at the start of the .prg file, prior to any FlagShip language statement.
7. You may directly access and assign all TYPED FlagShip variables declared by LOCAL..AS, STATIC..AS and GLOBAL..AS. Refer to section CMD. In the C program block, use directly the TYPED variable name (the first 10 characters of the name) given in lowercase. See 3.3 and 3.4.
8. To assign any FlagShip variables other than the TYPED ones, or to invoke functions from the FlagShip library, use the Open C macros, described in section EXT.4.
9. To invoke any function from the C or Unix libraries within the C program block, use the standard C syntax, see EXT.3.3 below. To invoke FlagShip standard functions, use the Open C API calling convention. See section EXT.4.

10. Because of the different parameter passing, you may not use Extend C API within functions declared by Open-C or by the .prg syntax, i.e. not with Inline-C API. The use of Open-C API (see EXT.4) is fully supported.

The rest is standard C programming. See also the "most frequent errors" in chapter 1.

3.2 Compiling the Program

Since the inline C code is included in the standard .prg source file, no special compilation is necessary. The program modules are compiled according to section FSC, e.g.

```
$ FlagShip test.prg
$ FlagShip myprog*.prg -m -Mmyprog -otest
```

or by using the make utility etc.

Note: If the compiler switch -Dname is used, the definition is passed to both the .prg and the C part. To pass a definition or its value to the C part only, use -Wc,-Dname instead.

3.3 Accessing Unix/Windows Libraries

The most common use of Inline C is to access a UNIX, C or Windows library function, not available directly in FlagShip. For example, to calculate a random number in the range 0 to 2³², use:

```
** file test.prg
#Cinline
    long lrand48();          /* prototypes for C */
    void srand48(long);
#endCinline
initrandom()              // init random
? random()                // calculate random number
? random()                // other random number
QUIT

FUNCTION initrandom
*****                  // call it at least once
LOCAL_LONG linit_val
linit_val := SECONDS() + DAY( DATE() ) + SECONDSCPU()
#Cinline
    srand48( linit_val );  /* note lowercase of var name */
#endCinline
RETURN NIL

FUNCTION random
*****                  // subsequent call is possible
LOCAL_LONG RandomNumber
#Cinline
    randomnumb = lrand48(); /* lowercase & 10 chars of var name */
#endCinline
RETURN randomNumber
```

```
** eof test.prg
```

For more examples, refer to section LNG.8.

3.4 Using an Inline C Function

Using the FlagShip CALL command, you may execute a C inline function directly from the FlagShip language. According to rule 3.1.6, the function has to be declared at the beginning of the .prg file. Example of a function similar to STRPOKE():

```
** file test.prg

#Cinline
#include <string.h>
#ifndef uchar
# define uchar unsigned char
#endif

void ReplaceOneChar (i noutStr, posi t, value) /* callable from
----- C and by CALL */
uchar *i noutStr;
int posi t, value;
{
    if (posi t <= strlen(i noutStr) && posi t > 0 &&
        value > 0 && value <= 255)
        i noutStr[posi t -1] = (uchar) value;
}
#endCinline
STATIC any_var
LOCAL str_var

* FUNCTION test() // remove the comment if above vars have
// a file-wide scoping and compile then
// using the -na switch.

str_var = "mystring"
CALL ReplaceOneChar WITH str_var, WORD(5), WORD(ASC("X"))
? str_var // mystXi ng
#Cinline
{ /* note rule 3.1.5 */
    unsigned char *other_str = "other text";
    ReplaceOneChar (other_str, 6, (int) '-'); /* call it from C */
    printw ("\n%s\n", other_str); /* see EXT.2.6 */
}
#endCinline

?
str_var = "Other Invoking ...."
CALL ReplaceOneChar WITH str_var, WORD(7), WORD(ASC("i"))
CALL ReplaceOneChar WITH str_var, WORD(1), WORD(111)

? str_var // other i nvoking ....
RETURN NIL

*** eof test.prg
```

For more examples, refer to the CALL command.

3.5 Prototyping, Casting

When inline C is used, be very careful to use the correct #include files, especially for functions and parameters other than (int), e.g.:

```
** test.prg
? "I am now in directory", CURDIR()
#Cinline
{
# include "prototypes.h"          /* for chrdir() --- OS dependent */
# include "string.h"            /* for strxxx() */
char *str = "/usr", full[200];
strcpy(full, str);
strcat(full, "/mydir");
chdir (full);
}
#endCinline
? "and now in directory", CURDIR()
```

When using the Open C API System within the inline C, refer to section 4 for more information, e.g.:

```
** test.prg
LOCAL abc := "any text", num
#Cinline
{
# include "FSopenc.h"
# include "string.h"
extern variable *_bb_qout();
variable *varPtr, *args[1];
unsigned char tmp[200];
VAR_NEW_ARGS (args, 1);
strcpy (tmp, "new text from inline C");
SET_VAR_CHR (args[0], tmp);
_bb_qout (1, args);
SET_VAR_COPY (VAR_NAME_LOCAL (abc), args[0]);
SET_VAR_NUM (VAR_NAME_LOCAL (num), (double) 5);
}
#endCinline
?? "abc=" + abc, "num=", num
```

Normally, when compiling with the `-Wc, -W2` (or `-u"cc -W2"`) option, there should be no no warnings.

3.6 Using own main() function

In special cases, you may supply your own main() C function and start the FlagShip application from there. The default main() module, available in the FlagShip library, is simple:

```
extern int FlagShip_start(int argc, char *argv[]);

int main (int argc, char *argv[])
{
    return FlagShip_start(argc, argv);
}
```

Using your own main() module is possible. The main() may process additional stuff and then call the FlagShip application via

```
int = FlagShip_start(int, char *[]);
```

Note that you need to provide both arguments, and at least one element in argv[] as character string containing the name of the executable, same or similar to argv[0] of your C based main() function.

Example: doing some stuff (here displaying the command-line parameter and sleep for 2 seconds), passing one (or the 1st optional) argument, plus the executable name to FlagShip_start():

```
/* file mymain.c */
#include "stdio.h"
#include "string.h"
extern int FlagShip_start(int argc, char *argv[]); // prototype!

int main (int argc, char *argv[]) // your own main
{
    int ii, ok, myArgc = 1;
    char myArgv[2][1025];

    for(ii=0; ii < argc; ii++)
        printf("command-line parameter %d in main(): '%s' \n",
               ii, argv[ii]);
    printf("doing my stuff... \n"); sleep(2);

    argv[0][1024] = 0;
    strncpy(myArgv[0], argv[0], 1024); // current executable
    if(argc > 1) {
        myArgc++;
        argv[1][1024] = 0;
        strncpy(myArgv[1], "Opt. param: ", 1024);
        strcat(myArgv[1], argv[1], 1024 - strlen(argv[1]));
    }
    printf("starting FlagShip application... \n");

    ok = FlagShip_start(myArgc, myArgv); // start FlagShip's main

    printf("The application returned %d\n", ok);
}
```

```
printf("Finishing my stuff...\n"); sleep(2);  
return 0; // or: return ok;  
}  
/* eof mymain.c */
```

and preferably compile/link:

```
FlagShip my*.prg mymain.c
```

or compile your mymain.c by C compiler to object and link this object with your application, see details in section FSC, e.g.

```
cc -c mymain.c  
FlagShip my*.prg mymain.o
```

or in MS-Windows

```
cl -c mymain.c  
FlagShip my*.prg mymain.obj
```

4. Open C System

Since the FlagShip compiler produces .c source files from the .prg's, an experienced C programmer may modify these files to optimize the code in some circumstances. Also when using standard C programs, it is possible to execute functions from the FlagShip library.

4.1 Structure of the Executable Program

Any .prg file compiled by FlagShip is translated into a .c file of the same name. With fully automatic compilation by using the main program name only and omitting the -m switch, separate .c files equivalent to the .prg and .fmt ones are produced.

When binding the final executable, i.e. if the compiler switch -c is not used, a start-up module named mainprog_m.c is also automatically produced. Refer also to FSC.1.1.4. This start-up module initializes the curses library, the FlagShip run-time system, the dynamic scoped variable system and then invokes the PROCEDURE or FUNCTION named mainprog. The name is specified at FlagShip compile-time by the -Mmainprog option or defined by the first file given name. Note: it is not possible to use FlagShip library functions without executing the start-up module.

The .c modules are usually compiled by starting FlagShip, which then invokes cc to produce the .o object files. Where required, invoking cc directly is also possible, while using the default switches (determined by the FlagShip -v option, see section FSC).

In the final phase, FlagShip (or cc) invokes the ld linker, to link all the specified .o modules and used C and UNIX library functions into an UNIX executable file, named a.out by default.

For detailed information, refer to sections LNG and FSC.

4.2 Structure of the C program

The following example of a program, written in the FlagShip (xBASE) language

```
1: *** test.prg
2: mycolor = "W+/B, N/W"
3: SET COLOR TO (mycolor)
4: xx = myudf (mycolor, 55)
5: USE mydata
6: QUI T
7:
8: FUNCTION myudf (p1, p2)
9: LOCAL xyz := 1
10: ? p2, p2 + xyz
11: RETURN NIL
```

will first be preprocessed into test.bp, according to the file std.fh, included by default. Note that the commands are now translated to the equivalent FlagShip functions (compiled with -nd -nl switches):

```

#line "test.prg" 1
#line "std.fh" 1
#line "set.fh" 1
#line "std.fh" 25

#line "test.prg" 1
* * * test.prg // 1
mycolor ="W+/B, N/W" // 2
SetColor(mycolor) // 3
xx = myudf(mycolor, 55) // 4
dbUseArea(.f., , 'mydata' , , if(.f. .or. .f., ! .f., NIL), .f.) // 5
__Quit() // 6

__FUNCTION myudf (p1, p2) // 8
__LOCAL xyz :=1 // 9
Qout(p2, p2 +xyz) // 10
__RETURN NIL // 11

#line "test.prg" 13

```

This .bp file is now compiled by the FlagShip compiler into a C file named test.c. The C code produced is relatively complex, since all the significant language differences between the high level FlagShip (xBASE) and the low level C have to be considered.

The most significant differences are the run-time system, macros, dynamic variable scoping, non-fixed variable types, dynamic string length, code blocks, objects, overlaid names for variables and database fields and their different visibility etc. (Refer also to the LNG section). Therefore, an additional overhead is necessary, which is usually performed by special FlagShip functions.

This overhead may be reduced (and the application speed increased) by using local and TYPED, not the dynamic variables (PRIVATE, PUBLIC).

Note: the following listing of the translation into C is given for your orientation only and may be modified and optimized without further notice. To access FlagShip language elements (variables, functions) from a user C program, use the Open C API functions and macros described in chapters 4.4 to 4.8.

```

# include "FlagShip.h" // * Note 1 */
static char *fgs_file_name = "test"; // * Note 2 */

extern variable *_bb_test(); // * Note 3 */
extern variable *_bb_setcolor();
extern variable *_bb_myudf();
extern variable *_bb_if();
extern variable *_bb_dbusearea();
extern variable *_bb__quit();
extern variable *_bb_qout();
static int _bbvar_mycolor; // * Note 4 */
static int _bbvar_xx;
static int _bbvar_nil;

/* * * * test.prg */ // * Note 5 */

```

```

variable *_bb_test(parno, parptr) /* Note 6 */
int parno;
variable *parptr[];
{
    struct fn_stack fn_stack; /* Note 7 */
    char *__who_me = fgs_fn_start(&fn_stack, /* Note 7 */
                                fgs_file_name, "test", 2, parno); /* Note 7 */
    variable *par0[7], *par1[3], *par2[1]; /* Note 8 */
#line 2 "test.prg" /* Note 9 */
    par0[0] = set_cvar (mv_names[_bbvar_mycol or].v,
                      "W+/B, N/W"); /* Note 10 */
#line 3 "test.prg" /* Note 9 */
    par0[0] = mk_var_cp(mv_names[_bbvar_mycol or].f);
    par0[1] = _bb_setcol or(1, &par0[0]);
#line 4 "test.prg"
    par0[0] = mk_var_cp(mv_names[_bbvar_mycol or].f); /* Note 11 */
    par0[1] = set_nv(55.0, 0);
    par0[2] = _bb_myudf(2, &par0[0]);
    par0[3] = cpy_var(mv_names[_bbvar_xx].v, par0[2]);
#line 5 "test.prg"
    par0[0] = mk_var_cp(FALSE_VAR);
    par0[1] = cre_tmpvar();
    par0[2] = set_cv("mydata");
    par0[3] = cre_tmpvar();
    lbor1;
    if(0) { /* Note 12 */
        par1[1] = TRUE_VAR;
        par1[0] = par1[1];
    } else {
        par1[2] = mv_names[_bbvar_nil].f;
        par1[0] = par1[2];
    }
    par0[4] = mk_var_cp(par1[0]);
    par0[5] = mk_var_cp(FALSE_VAR);
    par0[6] = _bb_dbusearea(6, &par0[0]); /* Note 13 */
#line 6 "test.prg"
    par0[0] = _bb__quit(0, 0);
    lbl0:
        return fgs_fn_end(&fn_stack); /* Note 14 */
}

variable *_bb_myudf(parno, parptr) /* Note 6 */
int parno;
variable *parptr[];
{
    struct fn_stack fn_stack; /* Note 7 */
    char *__who_me = fgs_fn_start(&fn_stack,
                                fgs_file_name, "myudf", 8, parno);
    variable *par0[7], *par1[3], *par2[1];
    variable *_fgspvar_p1; /* Note 15 */
    variable *_fgspvar_p2;
    variable *_fgslvar_xyz = fgs_l ocal_param(0, 0, 0);
    _fgspvar_p1 = fgs_l ocal_param(parno, 0, parptr);
    _fgspvar_p2 = fgs_l ocal_param(parno, 1, parptr);

#line 9 "test.prg"

```

```

cpy_var(_fgsl var_xyz, set_nv(1.0, 0));
#l i n e 10 "test.prg"
par0[0] = mk_var_cp(_fgspvar_p2);
par1[0] = u_add_u(_fgspvar_p2, _fgsl var_xyz, 0);
par0[1] = mk_var_cp(par1[0]);
par0[2] = _bb_qout(2, &par0[0]); /* Note 16 */
#l i n e 11 "test.prg"
fn_stack.rval ptr=mk_var_cp(mv_names[_bbvar_ni l].f); /* Note 17 */
goto l b l 2;
l b r e t u r n 3;
fn_stack.l i n _ n o = 13; i f (_sstep) _d e b u g g e r (); /* Note 18 */
#l i n e 13 "test.prg"
l b l 2:
return fgs_fn_end(&fn_stack); /* Note 14 */
}

static struct f_sym_tab l o c _ t a b [] = { /* Note 19 */
{ "test", _bb_test },
{ "setcol or", _bb_setcol or },
{ "myudf", _bb_myudf },
{ "i f", _bb_i f },
{ "dbusearea", _bb_dbusearea },
{ "__qui t", _bb__qui t },
{ "qout", _bb_qout },
{ "", 0 };
static struct g_sym_tab g_l o c _ t a b = { "FS...", l o c _ t a b };
static void refer_to_func() {
struct g_sym_tab *ss = &g_l o c _ t a b;
ss = &ss[1];
}

static struct f_var_tab var_tab[] = { /* Note 20 */
{ "mycol or", &_bbvar_mycol or },
{ "xx", &_bbvar_xx },
{ "ni l", &_bbvar_ni l },
{ "", &fgs_fs403 }};
static struct g_var_tab g_var_tab = { "FS...", var_tab };
static void refer_to() {
struct g_var_tab *vv = &g_var_tab;
vv = &vv[1];
}

```

Explanatory notes on the above C code:

- 1: The <FlagShip_dir>/include/FlagShip.h file contains all the required #define and structures (e.g. the variable* one) used. Additional Unix files, such as math.h, curses.h and ctype.h are included.
- 2: Name of the .prg file, required for the RTE and error system.
- 3: A block of external FlagShip function prototypes.
- 4: Block specifying FlagShip dynamic scoped PRIVATE and PUBLIC variables used. Note: all variable names are prefixed with _bbvar_, shortened to 10 characters and in lower case.

- 5: If the FlagShip preprocessor directive `#nocomment` is not used, all `*` and `NOTE` comments are passed to C for your orientation.
- 6: FlagShip `PROCEDURE` or `FUNCTION`. The UDF name is used in lowercase, shortened to 10 characters and prefixed with `_bb_` to omit conflicts with other UNIX or user supplied functions. The formal parameters are always passed in the parameter array `parptr` of variable length.
- 7: Internal FlagShip information for the run-time system. Do not change it.
- 8: Dynamic arrays storing temporary variables, arguments, results etc.
- 9: When the compiler switch `-nL` is not used, the `#line` directive specifies the corresponding line number of the `.prg` file. Used by the `cc` compiler to display errors and warnings according to the original source file.
- 10: Access to the dynamic scoped variable "mycolor", copying a string constant "W+/B,N/W" into it.
- 11: Copy the contents of the field or dynamic variable "mycolor" into the first element of a local array `par0`, the numeric constant 55 into the second element. Invoke the `PROCEDURE` or `FUNCTION` "myudf", using two elements of the array `par0`. The function result is stored into the third array element and then copied into the dynamic FlagShip variable "xx".
- 12: The `IF()` or `IIF()` function is translated directly into C code.
- 13: The standard FlagShip function "dbusearea" is invoked using six parameters.
- 14: Termination of a `PROCEDURE` or `FUNCTION` with correction of variable scoping and clean-up of temporary variables.
- 15: Declaration and initialization of formal `LOCAL` parameters "p1", "p2" and `LOCAL` variable "xyz".
- 16: The standard FlagShip function "qout" is invoked using 2 parameters.
- 17: Preparing the return value of the UDF.
- 18: If the debugger and line numbers of the `.prg` are not disabled (by the `-nd -nl` switch), additional info is created for each `.prg` statement. In this overview, only this one line is listed.
- 19: Table of UDFs used for the run-time system.
- 20: Table of dynamic scoped variables.

4.3 Modifying the C Source

An experienced C programmer may modify the .c source file according to chapter 4.2 by inserting additional C statements, variables, structures and so on.

Be very careful not to destroy the internal tables needed for the FlagShip run-time and variable system. System crashes and segmentations will occur in such a case. Do not use or modify undocumented, internal FlagShip functions, since they may be changed without further notice. Instead, use the documented, portable functions and C macros of the Open C system, described in chapters 4.4 to 4.8

4.4 The FlagShip Open C API System

The modular, open architecture of FlagShip allows you to access nearly the whole system, including variables and standard functions through any C program. By using such a C interface, access from other programming languages, such as Fortran, Cobol etc. is also possible. The Open C System is very simple for an experienced C programmer (compared to the produced C code by FlagShip), since the programmer himself controls the variable usage:

1. The main program module must be any FlagShip PROCEDURE, FUNCTION or a simple .prg program. This main module can contain only one statement, the CALL, #Cinline or Extend C invocation of other C programs. Alternatively, you may provide your own main() C function, see EXT.3.7
2. To access FlagShip standard functions, like the database or i/o system, use the syntax described in section FUN and the calling sequence described in chapter 4.6. When calling a FlagShip function, the calling program is responsible for making a copy of constants and variables, which are not passed by reference to ensure their original contents.
3. For your convenience, most of the FlagShip commands are translated by the preprocessor to equivalent functions, so that they are also directly accessible by the C program. Although the translation is given in the CMD section, this information is approximate. Refer to the /include/std.fh file for the current valid syntax of the translation or study the preprocessed .bp file, created by the -a compiler switch.
4. For portability, use the documented functions and macros in chapters 4.5 and 4.8 only. Do not use the functions generated by the FlagShip compiler, since these may be modified and optimized without further notice.
5. To include language programs other than FlagShip or C (like Cobol or Fortran), compile these modules into .o files and compile/link them by FlagShip together with other .prg or .c programs. Supply the additional language dependent libraries using the -l compiler switch. Refer to section FSC.
6. You even may replace FlagShip functions by your own. When you replace library module by an object file linked in manually, the object module (based on .prg or .c file) must contain all public functions available in the library's module, otherwise linker error displays.
7. You may use the Open-C API within functions declared and initialized by the Extend-C API. But because of the different parameter passing, you may **not** use Extend C API within functions declared by Open-C or by the .prg syntax, i.e. not in Inline-C APIs.

4.5 Using FlagShip Variables

To access, create or modify FlagShip's PRIVATE, PUBLIC, LOCAL and STATIC variables from the C code, several functions and C macros are available in the `<FlagShip_dir>/include/FSopenc.h` #include file. Overview of the most important functions/macros managing the variable system:

a. Creating and maintaining new FlagShip variables

Function / Macro	Returns	Description
VAR_NEW	(FSvar *)	Creates a new temp variable
VAR_NEW_ARGS(...)	none	Creates an array of variables
VAR_NEW_ARRAY(...)	(FSvar *)	Creates an FlagShip array
VAR_NEW_COPY(...)	(FSvar *)	Creates new var copying old one
VAR_NEW_STATIC(...)	(FSvar *)	Creates new static variable
VAR_DELETE	none	Deletes all temp variables

b. Access by name to variables already created by FlagShip

Function / Macro	Returns	FlagShip Variable Type
VAR_NAME_LOCAL(name)	(FSvar *)	LOCAL
VAR_NAME_LOCPAR(name)	(FSvar *)	LOCAL parameter name
VAR_NAME_STATIC(name)	(FSvar *)	STATIC
VAR_NAME_MEMVAR(name)	(FSvar *)	PRIVATE, PUBLIC, PARAMETERS
VAR_NAME_FIELD(name)	(FSvar *)	FIELD (or PRIVATE)
VAR_NAME_MEMVAR_BYNAME_OR_NEW(cName)		PRIVATE, PUBLIC
VAR_NAME_MEMVAR_BYNAME_OR_NULL(cName)		PRIVATE, PUBLIC
VAR_NAME_FIELD_BYNAME_OR_ERR(cName)		FIELD
VAR_NAME_FIELD_BYNAME_OR_NULL(cName)		FIELD
name (in lowercase)	(int)	LOCAL_INT, STATIC_INT
name (in lowercase)	(long)	LOCAL_LONG, STATIC_LONG
name (in lowercase)	(double)	LOCAL_DOUBLE, STATIC_DOUBLE

c. Determining the variable type

Function / Macro	Returns	Description
VAR_ISTYPE(var)	(char)	Var type C,D,M,N,L,S,O,B,U...
VAR_ISMODE(var)	(char)	Var mode F,L,I,N,n,C,T,t,V,v...
VAR_ISLEN(var)	(int)	allocated string length
VAR_ISDECI(var)	(int)	deci count displayed on numvar
VAR_ISDIM(var)	(int)	array dimension size
IS_VAR_ARR(var)	(int) 0,1	is variable an array ?
IS_VAR_BLOCK(var)	(int) 0,1	is variable a code block ?
IS_VAR_CHR(var)	(int) 0,1	is variable a string ?
IS_VAR_DATE(var)	(int) 0,1	is variable a date ?
IS_VAR_LOG(var)	(int) 0,1	is variable a boolean ?
IS_VAR_NIL(var)	(int) 0,1	is variable undefined, NIL ?
IS_VAR_NUM(var)	(int) 0,1	is variable numeric ?
IS_VAR_INT(var)	(int) 0,1	is variable integer numeric ?
IS_VAR_FP(var)	(int) 0,1	is variable float numeric ?
IS_VAR_OBJ(var)	(int) 0,1	is variable an object ?
IS_VAR_SCR(var)	(int) 0,1	is variable a screen ?
IS_VAR_TRUE(var)	(int) 0,1	is log.variable .T. ?
IS_VAR_FALSE(var)	(int) 0,1	is log.variable .F. ?
IS_VAR_TRUE_BLK(var)	(int) 0,1	is eval. code block .T. ?
IS_VAR_EMPTY(var)	(int) 0,1	is variable empty or NIL ?
IS_VAR_SPECIAL(var)	(int) 0,1	is it a special variable ?
IS_VAR_STD(var)	(int) 0,1	is it a std.variable C,D,M,N,L?
IS_VAR_FIELD(var)	(int) 0,1	is variable a .dbf field ?
IS_VAR_BYREF(var)	(int) 0,1	is variable passed by refer ?

d. Compare or manipulate two variables

Function / Macro	Returns	Description
IS_VAR_EQ(var1,var2)	(int) 0,1	is var1 = var2 ?
IS_VAR_EE(var1,var2)	(int) 0,1	is var1 == var2 ?
IS_VAR_NE(var1,var2)	(int) 0,1	is var1 != var2 ?
IS_VAR_GT(var1,var2)	(int) 0,1	is var1 > var2 ?
IS_VAR_GE(var1,var2)	(int) 0,1	is var1 >= var2 ?
IS_VAR_LT(var1,var2)	(int) 0,1	is var1 < var2 ?
IS_VAR_LE(var1,var2)	(int) 0,1	is var1 <= var2 ?
SET_VAR_ADD(var1,var2)	var	add var1 + var2

e. Access to the variable contents

Function / Macro	Returns	FlagShip Variable Type
VAR_ARRELEM(var)	(FSvar *)	access to an array element
VAR_BLOCK(var)	(*fn())	code BLOCK
VAR_CHR(var)	(uchar *)	CHARACTERs of VAR_ISLEN()
VAR_DATE(var)	(long)	DATE
VAR_LOG(var)	(int) 0,1	LOGICAL, 1 represents .T.
VAR_NUM(var)	(double)	NUMERIC
VAR_INT(var)	(long)	NUMERIC, INTVAR
VAR_OBJ(var,elem)	(FSvar *)	access to object element
VAR_SCR(var)	(window *)	SCREEN
VAR_SPECIAL(var)	(any *)	access user def. pointer
VAR_ISFLDPOS(var)	(ushort)	ordinal field position
VAR_ISFLDWA(var)	(ushort)	associated WA to field
name	(int)	LOCAL_INT, STATIC_INT
name	(long)	LOCAL_LONG, STATIC_LONG
name	(double)	LOCAL_DOUBLE, STATIC_DOUBLE

f. Modifying the variable contents

Function / Macro	Input	FlagShip Variable Type
SET_VAR_BLOCK(...)	(ptrCfun)	BLOCK
SET_VAR_CHR(...)	(char *)	CHARACTER
SET_VAR_CHRLEN(...)	(char *)	CHARACTER, may include \0
SET_VAR_COPY(...)	(FSvar *)	any type C,D,M,N,L,S
SET_VAR_DATE(...)	(long)	DATE
SET_VAR_LOG(...)	(int)	LOGICAL
SET_VAR_NIL(...)	none	NIL, undefined
SET_VAR_INT(...)	(long)	NUMERIC, INTVAR
SET_VAR_NUM(...)	(double)	NUMERIC
SET_VAR_NUMDECI(...)	(double)	NUMERIC
SET_VAR_SCR(...)	(window *)	SCREEN
SET_VAR_SPECIAL(...)	(any *)	n.a.
SET_VAR_LOWER(...) (FSvar *)	CHARACTER
SET_VAR_UPPER(...) (FSvar *)	CHARACTER
SET_VAR_MACRO(...) (char *)	result of the macro eval.
name	(int)	LOCAL_INT, STATIC_INT
name	(long)	LOCAL_LONG, STATIC_LONG
name	(double)	LOCAL_DOUBLE, STATIC_DOUBLE

g. UDF declaration or invocation

Function / Macro	Description
UDF_DECL(name)	declare UDF header
UDF_PROT(name)	prototype external UDF
UDF_EXEC(name)(argv)	invoke any UDF

h. Object/class access

Function / Macro	Description
OBJ_DECL_METH(clas, meth)	declare method function
OBJ_DECL_METHACCESS(clas, meth)	declare access function
OBJ_DECL_METHASSIGN(clas, meth)	declare assign function
OBJ_METHEXEC(...)	invoke a method
OBJ_ACCEXEC(...)	access instance or access method
OBJ_ASSEXEC(...)	assign instance or assign method

i. Predefined constant variables. Note: **Do not** change their value in any way. Use them for comparison and by copying to regular variables only!

Name	Var Type	FlagShip Variable Type
NIL_VAR	(FSvar *)	NIL
NULLSTR_VAR	(FSvar *)	CHARACTER, null string ""
NULLDATE_VAR	(FSvar *)	DATE value { / / }
TRUE_VAR	(FSvar *)	LOGICAL value .T.
FALSE_VAR	(FSvar *)	LOGICAL value .F.
ZERO_VAR	(FSvar *)	NUMERIC value 0.0
IZERO_VAR	(FSvar *)	INTVAR value 0
NILLCB_VAR	(FSvar *)	empty code BLOCK
ERROR_VAR	(FSvar *)	ERROR value

The rules to access, create or modify FlagShip variables are quite simple and comply with the usage of standard C structures:

1. FlagShip variables are managed in a "variable pool". An internal garbage collection on variables is done regularly, but at least at the RETURN from a FlagShip PROCEDURE or FUNCTION, at the end of FOR or WHILE loops, and when issuing the VAR_DELETE macro. Note: Additional garbage collection is invoked when generated in the .c code from the FlagShip compiler using the del_tmpvar(...) or fgs_cleanup (...) call.

Therefore, the newly-created FlagShip variables have a lifetime at least as long as the entire C program and the return into the .prg program, unless they are explicitly deleted previously. For the more information on the lifetime and scoping of variables specified in the .prg part, refer to section LNG.2.6. To avoid variable deletion, use VAR_NEW_STATIC().

2. FlagShip variables may have any valid type. This can be changed during the variable lifetime. The TYPED variables declared in FlagShip have the same type (int, long, double) as in standard C, which cannot be changed.
3. A new FlagShip variable name is declared in the C source as (FSvar *) or in the older, equivalent syntax as (variable *), the structure of which is defined in the <FlagShip_dir>/include/FlagShip.h file. Note: when returning to the .prg program, this variable's name will be invisible, but its contents are still valid. Refer to paragraph 1 above. Hint: it is good programming practice to assign 0 to the variable pointer at its declaration, since the variable system will detect such uninitialized variables.

Before using the variable (pointer) for the first time, allocate the memory space for it via `VAR_NEW` or `VAR_NEW_xxx()`. Do not use `SET_VAR_xxx()` or `SET_VAR_COPY()` before doing the initialization.

4. To access a variable declared in the `.prg` source by name in the `#Cinline` part, use `VAR_NAME_xxx()` functions/macros.
5. When the Open C API is used in `#Cinline` code, include the `FSopenc.h` file at the beginning of the `.prg` source, or within the `{..}` part of the inline C code.

Refer to the chapter 4.8 for a full description of functions and macros using the FlagShip variable system.

4.6 Using FlagShip Standard Functions

You may invoke any FlagShip standard function, described in the section FUN (and CMD), directly from the C source. The only assumption is that the variable and run-time system is already initialized according to chapter EXT.4.4.1.

To avoid conflicts with other Unix or user-defined functions, any standard FlagShip function is prefixed with "_bb_" (underscore, bb, underscore), specified in lowercase and abbreviated to 10 characters, e.g. the standard function DBCREATEINDEX() is declared in the library as _bb_dbcreatein(). Note: since this prefix may be changed in the future releases, you should prefer the usage of UDF_DECL(), UDF_PROT() and UDF_EXEC() described below.

The parameters are passed by the standard argc,argv[] calling convention. Any standard FlagShip function (e.g. "anyname") is declared as

```
UDF_DECL(anyname)
{
    ...
    return <Fsvar *>
}
```

which is expanded by the UDF_DECL() macro to

```
FSvar *_bb_anyname (int argc, FSvar *argv[] )
{
    ...
    return <Fsvar *>
}
```

Should no parameters be required, the function may be invoked from the C program, e.g.

```
FSvar *retvar;
UDF_PROT(anyname);
retvar = UDF_EXEC(anyname)(0, 0);
```

or the same by the older syntax:

```
variable *retvar;
extern variable *_bb_anyname();
retvar = _bb_anyname (0, 0); /* if not used yet */
```

When one or more parameters are required, an array of variables must be created and the array elements filled with the arguments to pass. Additionally, a number of elements to use (argc) is passed to the function:

```
FSvar *param[4], *retval = 0;
VAR_NEW_ARGS (param, 4); /* fill all elements by NIL_VAR */
SET_VAR_NUM (param[0], 1.0); /* 1st param = numeric 1 */
SET_VAR_CHR (param[3], "string"); /* 4th param = character value */
retval = UDF_EXEC(anyname)(4, param);
```

which is expanded to

```
variable *param[4], *retval = 0;
VAR_NEW_ARGS (param, 4);          /* fill all elements by NIL_VAR */
SET_VAR_NUM (param[0], 1.0);     /* 1st param = numeric 1 */
SET_VAR_CHR (param[3], "string"); /* 4th param = character value */
retval = _bb_anyname (4, param);
```

The above example also demonstrates skipped parameters, where the skipped array elements must be filled with NIL_VAR. It is equivalent to FlagShip programming by

```
retval = AnyName (1, , , "string")
```

The standard function returns its function value by using a new FlagShip pool variable (refer to EXT.4.5.1). You may use this variable directly or copy it to another variable, for example:

```
FSvar *retvar1 = 0, *retvar2 = 0;
FSvar *othervar;          /* if required */
UDF_PROT(anyname); /* prototype as extern variable *_bb_anyname() */
retvar1 = UDF_EXEC(anyname) (0, 0);
retvar2 = UDF_EXEC(anyname) (0, 0);
othervar = VAR_NEW_COPY (retvar1); /* if required */
VAR_DELETE; /* if not more needed */
```

When calling a FlagShip function, the **invoked** program is responsible for making a copy of constants and variables which are not passed by reference in order to ensure their original contents.

For your convenience, most of the FlagShip commands are translated by the preprocessor to equivalent functions, so that they are also directly accessible by the C program. Although the translation is given in the CMD section, it is approximate information. Refer to *<FlagShip_dir>/include/std.fh* file for the syntax currently valid for the translation or study the preprocessed .bp file, created by the -a compiler switch.

4.7 Screen and File Input/Output

Since the **screen input/output** is already initialized by means of UNIX curses routines in the start-up module (see chapter EXT.4.4.1 and section LNG.5), all standard curses functions can be used to manage input and output.

For your convenience, all FlagShip standard i/o functions may be used, as described in Chapter 4.6. These will automatically manage the PC-8 character set and the i/o mapping used. Refer also to section LNG.5.

Should the curses and FlagShip screen/terminal i/o not be required in a **special case**, you may deactivate and activate the curses facility.

- a. Using the `endwin()` curses function for de-activating, and the FlagShip `FS_SET` ("setenv", "TERM", `NIL`, `.T.`) function for reactivating the curses.
- b. Using the `def_prog_mode()` and `endwin()` curses functions for de-activating, and the `reset_prog_mode()` curses function for reactivating the curses.
- c. Disabling the Curses start-up initialization in the `cursini.prg`, according to chapter SYS.2.7.
- d. Best is to compile with `-io=b` FlagShip switch which then use standard C i/o only.

Warning: when the curses facility is not activated, do not use FlagShip terminal and screen functions, since unpredictable results may occur.

All **file i/o** in FlagShip are managed by standard UNIX or Windows i/o functions, such as `open()`, `fopen()` etc. There are therefore no special restrictions in using such i/o or streams i/o within your C API program. Do not open databases (and associated memos and index files) which are already open by FlagShip, since they are internally buffered in the FlagShip standard functions.

4.8 Using the Open C API

When using the Open C system, the example of chapter 4.2 may be also coded directly in C. Note, that this simplified code will not support macros, debugger, PROCNAME(), PROCLINE() etc., as opposed to the full FlagShip translation given chapter 4.2:

```
*** file main.prg
CALL my_C_udf
*** eof

/** file myCudf.c */
#include "FSopenc.h"
void my_C_udf ()
{
    FSvar *mycolor = 0, *xx = 0, *arg[6];
    int ii;
    UDF_PROT(setcolor); /* prototype it */
    UDF_PROT(dbusearea);
    VAR_NEW_ARGS (arg, 6); /* init array of NILs */
    mycolor = SET_VAR_CHR (VAR_NEW, "W+/B,N/W");
    SET_VAR_COPY (arg[0], mycolor);
    UDF_EXEC(setcolor)(1, arg); /* arg1 = color string */

    SET_VAR_COPY(arg[0], mycolor); /* arg1 = color string */
    SET_VAR_NUM (arg[1], (double) 55); /* arg2 = num.constant */
    xx = myudf (2, arg);

    SET_VAR_COPY(arg[0], FALSE_VAR); /* arg1 = .F. */
    SET_VAR_COPY(arg[1], NIL_VAR); /* arg2 = NIL */
    SET_VAR_CHR (arg[2], "mydata"); /* arg3 = dbf name */
    SET_VAR_COPY(arg[5], FALSE_VAR); /* arg6 = .F. */
    UDF_EXEC(dbusearea)(6, arg); /* arg4, arg5 are NILs */
    FS_QUIT_COMMAND;
}

FSvar *myudf (fnargc, fnargv)
int fnargc;
FSvar *fnargv[];
{
    FSvar *xyz = VAR_NEW, *p1 = 0, *p2 = 0, *arg[2];
    if (argc < 2)
        return (VAR_NEW_COPY(NIL_VAR));
    VAR_NEW_ARGS (arg, 2);
    p1 = fnargv[0];
    p2 = fnargv[1];
    xyz = SET_VAR_NUM (VAR_NEW, 1.0);
    SET_VAR_COPY(arg[0], p1);
    SET_VAR_NUM (arg[1], VAR_NUM(xyz) + VAR_NUM(p2));
    UDF_EXEC(qout)(2, arg); /* QOUT (p1, p2+xyz) */
    VAR_DELETE;
    return (VAR_NEW_COPY(NIL_VAR)); /* or: return (NIL_VAR); */
}
/** eof **/
```

Here is another example of the Open C API used as an inline C program. **PrintScreen()** is a general purpose function (in terminal i/o mode) to extract characters from the current screen, similar to pressing the PRINTSCREEN key, but you can reassign the function to any other key via SET KEY. The PrintScreen() function creates a one-dimensional array (1 to MAXROW()+1) containing the plain text of every screen line, extracted from the current screen window. It is designed for Curses, i.e. for use in the Terminal i/o mode. In GUI mode, you may use the Menu->Windows->PrintScreen, see also initiomenu.prg

```
#define ALTERNATIVE1      /* enable one of these */
// #define ALTERNATIVE2  /* two alternatives */

@ 0,0 SAY "Any text"
@ 5,9 SAY "Another text"
SET KEY -9 TO PrintScreen          // activate by F10
WAIT "Press F10 now and then another key"
scrarr = PrintScreen()             // activate from prg
? "The character at coordinates 0, 0 is:", SUBSTR(scrarr[1], 1, 1)
? "The character at coordinates 5,10 is:", SUBSTR(scrarr[6], 11, 1)
SET PRINT ON
SET CONSOLE OFF
AEVAL (scrarr, {|x| QOOUT(x+CHR(10))} ) // avoid empty line

FUNCTION PrintScreen (fromSetKey, dum2, dum3)
*****
#ifdef ALTERNATIVE1
/*
 * this alternative also demonstrates use of std. FS functions in C
 */
LOCAL_INT ymax := MAXROW()          // used also in C
LOCAL      aScreen[ymax +1]         // e.g. aScreen[25]
LOCAL status_print, status_cons, status_dev, ii := 0

#Cinline
{
# include "FSopenc.h"
# include "malloc.h"

UDF_PROT(savescreen);               /* not used yet */
UDF_PROT(maxcol);                   /* not used yet */

FSvar *actscreen = VAR_NEW;
WINDOW *winPtr;
int x, y, xmax;
unsigned char *oneline, *str;

actscreen = UDF_EXEC(savescreen)(0,0); /* SAVE SCREEN */
winPtr = VAR_SCR (actscreen);
xmax = (int) VAR_INTNUM(_bb_maxcol(0,0)); // xmax=MAXCOL()
oneline = malloc((unsigned)((xmax +1) * sizeof(unsigned char)));
if (oneline != NULL) {
for (y=0; y <= ymax; y++) {
str = oneline;
for (x=0; x <= xmax; x++)
*(str++) = (unsigned char)
(mvwinch(winPtr, y, x) & A_CHARTEXT); //curses
```

```

        SET_VAR_CHRLEN (VAR_ARRELEM (VAR_NAME_LOCAL(ascreeen), y+1),
                        oneline, xmax +1);
    }
    free (oneline);
}
VAR_DELETE;
}
#endCinline
#endif
#ifdef ALTERNATIVE2
/*
 * simplified ALTERNATIVE1
 */
LOCAL nYmax      AS INT
LOCAL nXmax      AS INT
LOCAL aScreen    AS ARRAY

nYmax := MAXROW()           // used also in C
nXmax := MAXCOL()          // used also in C
aScreen := ARRAY(nYmax + 1)

#Cinline
{
    #include "FSopenc.h"
    #include "malloc.h"

    int y;
    unsigned char *oneline;
    oneline = malloc((unsigned) ((nxmax + 1) * sizeof(unsigned char)));
    if (oneline != NULL) {
        for (y=0; y <= nymax; y++) {
            mvinnstr(y, 0, oneline, nxmax);
            SET_VAR_CHRLEN(VAR_ARRELEM (VAR_NAME_LOCAL(ascreeen), y+1),
                            oneline, nxmax +1);
        }
        free (oneline);
    }
}
#endCinline
#endif

// if invoked by SET KEY, store it in the printer-spool file

IF VALTYPE(fromSetKey) == "C"           // invoked by SET KEY
status_cons = SET (_SET_CONSOLE)      // SET CONSOLE on/off ?
status_print= SET (_SET_PRINTER)      // SET PRINTER on/off ?
status_devi = SET (_SET_DEVICE)       // SET DEVICE TO ??
SET PRINTER ON
SET CONSOLE OFF
? "Print Screen in proc/fn", fromSetKey, "at", DATE(), TIME()
AEVAL (aScreen, {|x| QOUT(STR(ii++,2) + ":", x) } )
? SPACE(4) + REPLICATE("-----+", 8)
?
SET DEVICE TO SCREEN
ALERT("The current screen is printed; to file " + FS_SET("print"))
SET (_SET_DEVICE, status_devi)

```

```
    SET CONSOLE (status_cons)
    SET PRINTER (status_print)
ENDIF
RETURN aScreen
```

Note: The output of some special characters, e.g. semi-graphical, may differ according to the currently used terminal and character mapping, especially when an alternate character set of the terminal is used to display them.

Note also the usage of standard functions and memory allocation in the previous examples.

4.9 Access to C++ functions and classes

The Open-C and Extend-C APIs support standard C calling convention. If you need to invoke C++ function which internally use slightly different call convention, you need to declare the by FlagShip accessible UDFs in your C/C++ header file using the `extern "C"` declaration. Here for example your `MyUdf1()` and `MyUdf2()` functions callable from the `.prg` code and `myCudf()` callable from C, C++ and via Inline-C:

```
/*----- mycppudfs.h -----*/
#i fdef __cpl uspl us
extern "C" {
#endi f

#i ncl ude "FSopenc. h"

// #i ncl ude ... for C source and C l i b s

// UDF_PROT(myudf1); // prototype your MyUdf*() functions
// UDF_PROT(myudf2); // if also called in your C/C++ code
// UDF_PROT(aadd); // and standard FS functions used in .cpp

i nt myCudf(char *); // prototype C callabl e functions

#i fdef __cpl uspl us
} /* end of extern "C" */

// your C++ declarati ons i f requi red, e. g.
// #i ncl ude ... for the C++ code and C++ l i b s
// cl ass ...

#endi f /* end of i fdef __cpl uspl us */

/-- eof mycppudfs. h --
```

and include the header file in your C++ source file(s):

```
/*----- mycppudfs.cpp -----*/
#i ncl ude "mycppudfs. h"

UDF_DECL(myudf1) // callabl e from .prg as myUdf1("str")
{
// ... standard C or C++ code, see EXT. 4.1-9, e. g.
FSvar *retVar = VAR_NEW;
pri ntf("\nEnteri ng MyUdf1()");
i f(argc > 0 && IS_VAR_CHR(argv[0]))
SET_VAR_I NT(retVar, myCudf((char *) VAR_CHR(argv[0])) );
el se
SET_VAR_I NT(retVar, -1);
return retVar;
}

UDF_DECL(myudf2) // callabl e from .prg as MyUdf2()
{
```

```

// .... standard C or C++ code, see EXT. 4.1-9, e.g.
printf("\nEntering MyUdf2()");
return SET_VAR_CHR(VAR_NEW, "Hello world");
}

int myCudf(char *str) // callable from .c, .cpp and Inline-C
{
    return strlen(str);
}
//-- eof mycppudfs.cpp --

```

An example of the .prg main program is:

```

/*----- myapp.prg -----*/
? "calling myUdf1 ..."
x := myUdf1("hello Udf1")
? "calling myUdf2 ..."
? "ret of myUdf2(=", myUdf2(), "and of myUdf1('hello Udf1')=", x

#Cinline
{
    #include "mycppudfs.h"
    int ret;
    ret = myCudf("hi");
    printf("\nmyCudf() returns: %d\n", ret); /* printed on stdout */
}
#endCinline
wait
//-- eof myapp.prg --

```

Then compile the .cpp file(s) separately by C++ compiler to object e.g.:

```
c++ -c -DFGLI NUXELF -DFGLI NUX -I. -I../i nclude mycppudfs.cpp
```

and simply add the resulting object file to your FlagShip link list, e.g.

```
FlagShip myapp*.prg mycppudfs.o -m -Mmyapp -o myapp [-i o=b]
```

The rest is equivalent to Open-C API described above. However, the C++ compiler is far more nitpicking than C, so be careful to correct and exact prototype your parameters. Note: if there is no factual constraint to use C++ use standard C instead, since it is more fault tolerant, the object files are usually smaller and faster, it is by far more backward compatible to other system/kernel sub-releases than C++ object code and the C++ libs, and whose .c files can be directly passed to the FlagShip command line.

4.10 The Open C API Reference

The following reference of the Open C API functions and macros is given in alphabetical order. For a logical order, refer to chapter 4.5.

IS_VAR_xxx ()

Syntax:

```
status = IS_VAR_ARR (varPtr)
status = IS_VAR_BLK (varPtr)
status = IS_VAR_BYREF (varPtr)
status = IS_VAR_CHR (varPtr)
status = IS_VAR_DATE (varPtr)
status = IS_VAR_EMPTY (varPtr)
status = IS_VAR_FALSE (varPtr)
status = IS_VAR_FIELD (varPtr)
status = IS_VAR_FP (varPtr)
status = IS_VAR_INT (varPtr)
status = IS_VAR_LOG (varPtr)
status = IS_VAR_NIL (varPtr)
status = IS_VAR_NUM (varPtr)
status = IS_VAR_OBJ (varPtr)
status = IS_VAR_SCR (varPtr)
status = IS_VAR_SPECIAL (varPtr)
status = IS_VAR_STD (varPtr)
status = IS_VAR_TRUE (varPtr)
status = IS_VAR_TRUE_BLK (varPtr)
```

Purpose:

Checks if the FlagShip variable is of the required type.

Arguments:

<varPtr> is a (FSvar *) variable which has been initialized, the type of which is to be checked.

Returns:

<status> is an (int) value. Zero signals an error, i.e. the variable is not of the required type, any other value signals success.

Description:

Among other flags, the variable type is also stored in the structure of a FlagShip pool variable. Assigning a value to it by means of SET_VAR_xx() or SET_VAR_COPY() will also automatically set the variable info flags. The current variable type can also be determined by the VAR_ISTYPE() function.

Function / Macro	Determines
------------------	------------

IS_VAR_ARR (varPtr)	is variable an array (A) ?
IS_VAR_FIELD (varPtr)	is variable a .dbf field ?
IS_VAR_NIL (varPtr)	is variable undefined, NIL ?
IS_VAR_BLK (varPtr)	is it a code block variable (B)?
IS_VAR_CHR (varPtr)	is it a string variable (C,M,E,V) ?
IS_VAR_DATE (varPtr)	is it a date variable (D) ?
IS_VAR_FP (varPtr)	is it a num. variable (F or N)?
IS_VAR_INT (varPtr)	is it a IntVar variable (I) ?
IS_VAR_LOG (varPtr)	is it a boolean variable (L) ?
IS_VAR_NUM (varPtr)	is it a general numeric (N,I,F) ?
IS_VAR_OBJ (varPtr)	is it an object variable (O) ?
IS_VAR_SCR (varPtr)	is it a screen variable (S) ?
IS_VAR_SPECIAL (varPtr)	is it a special user variable ?
IS_VAR_BYREF (varPtr)	is variable passed by refer ?
IS_VAR_EMPTY (varPtr)	is the variable empty or NIL ?
IS_VAR_FALSE (varPtr)	is log.variable .F. ?
IS_VAR_TRUE (varPtr)	is log.variable .T. ?
IS_VAR_STD (varPtr)	is it a std.variable (C,D,M,N,L) ?
IS_VAR_TRUE_BLK (varPtr)	does the code block return .T. ?

Example 1:

```

*** test.prg
LOCAL myvar := "any text"

#C i n l i n e
{
#i n c l u d e "FSopenc.h"
  FSvar *temp = VAR_NEW;
  i f (IS_VAR_CHR( VAR_NAME_LOCAL(myvar)))
    SET_VAR_COPY (temp, VAR_NAME_LOCAL(myvar));
  e l s e
    SET_VAR_CHR (temp, "unknown entry");
  i f (VAR_I S L E N (temp) > 2)
    VAR_CHR(temp)[2] = 'X';
  SET_VAR_COPY (VAR_NAME_LOCAL(myvar), temp);
  VAR_DELETE; /* release all temp vars */
}
#endC i n l i n e

? myvar // anX text

```

Example 2:

Assign correct values to local C variables, regardless the passed parameter type.
See also example in VAR_ISTYPE()

```

*** test.prg, compile: FlagShip test.prg -i o=b ; a.out | test.exe
? myFunct()
? myFunct(15)
? myFunct(10 / 3)
? myFunct("Hel l o worl d")
wai t

```

```

function myFuncT(par1)
local retVal
#Cinline
{
#include "FSopenc.h"
    int    myInt = 0;
    double myNum = 0.0;
    char   *myString = NULL;
    char   retStr[300];
    FSvar  *myVar;

    myVar = VAR_NAME_LOCPAR(par1);    // ptr to passed param
    if(IS_VAR_NUM(myVar)) {          // param numeric?
        myInt = VAR_INTNUM(myVar);
        myNum = VAR_FPNUM(myVar);
    } else if(IS_VAR_CHR(myVar))    // param character?
        myString = VAR_CHR(myVar);

    sprintf(retStr, "par: %c int=%d double=%f string=%.200s",
            VAR_ISTYPE(myVar), myInt, myNum, myString);
    SET_VAR_CHR(VAR_NAME_LOCAL(retval), retStr);
}
#endCinline
return retVal

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_??(), VAR_ISTYPE(), SET_VAR_xxx(), SET_VAR_COPY()

IS_VAR_EQ ()
IS_VAR_EE ()
IS_VAR_NE ()
IS_VAR_GT ()
IS_VAR_GE ()
IS_VAR_LT ()
IS_VAR_LE ()

Syntax:

```
status = IS_VAR_EQ (varPtr1, varPtr2)
status = IS_VAR_EE (varPtr1, varPtr2)
status = IS_VAR_NE (varPtr1, varPtr2)
status = IS_VAR_GT (varPtr1, varPtr2)
status = IS_VAR_GE (varPtr1, varPtr2)
status = IS_VAR_LT (varPtr1, varPtr2)
status = IS_VAR_LE (varPtr1, varPtr2)
```

Purpose:

Compares two FlagShip variables.

Arguments:

<varPtr1> and <varPtr2> are (FSvar *) FlagShip variables which have to be compared.

Returns:

<status> is an (int) value. Zero signals an unsuccessful comparison, any other value signals success.

Description:

You may compare two FlagShip variables of the same (or of comparable) type, same as in the FlagShip language. See details in chapter LNG.2.9.

Function / Macro	Determines
IS_VAR_EQ (var1, var2)	is var1 = var2 ?
IS_VAR_EE (var1, var2)	is var1 == var2 ?
IS_VAR_NE (var1, var2)	is var1 != var2 ?
IS_VAR_GT (var1, var2)	is var1 > var2 ?
IS_VAR_GE (var1, var2)	is var1 >= var2 ?
IS_VAR_LT (var1, var2)	is var1 < var2 ?
IS_VAR_LE (var1, var2)	is var1 <= var2 ?

Example:

```
FSvar *var1 = VAR_NEW, *var2 = VAR_NEW;
SET_VAR_CHR(var1, "abc");
SET_VAR_CHR(var2, "abcde");
printf ("%s) is %s == to (%s)\n",
        VAR_CHR(var1), (IS_VAR_EE(var1, var2) ? "" : "NOT"),
        VAR_CHR(var2));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_xxx(), SET_VAR_xxx(), VAR_NEW

OBJ_DECL_METH ()

OBJ_DECL_METHACCESS ()

OBJ_DECL_METHASSIGN ()

Syntax:

```
OBJ_DECL_METH      ( className , methName )
OBJ_DECL_METHACCESS ( className , methName )
OBJ_DECL_METHASSIGN ( className , methName )
```

Purpose:

Declares the function header for a method (or an access/assign method) of a class written in C.

Arguments:

<className> is the class name (up to 10 characters) in lower case.

<methName> is the name of the method, or access/ assign method given in lower case. Note, that the METHOD name is not abbreviated, but significant in the given length. The ACCESS and ASSIGN method names <methName> may contain up to 10 significant characters.

Note: it is recommended not to place white space within the brackets (...) in these macros, since some cc compilers have problems with a correct interpretation of the embedded spaces (or tabs).

Description:

OBJ_DECL_METH declares the function header for a METHOD of a class, including the parameter prototypes. The function is declared as

```
FSvar * _bb_<className>_<methName>_me (int argc, FSvar *argv[]);
```

which is equivalent to the

```
METHOD <methName> CLASS <className>
```

declaration in the FlagShip language.

OBJ_DECL_METHACCESS() declares the function header of an ACCESS method of an class, including the parameter prototypes. The function is declared as

```
FSvar * _bb_<className>_<methName>_ac (FSvar *ovSel f);
```

which is equivalent to the

```
ACCESS METHOD <methName> CLASS <className>
```

declaration in the FlagShip language.

OBJ_DECL_METHASSIGN() declares the function header of an ASSIGN method of a class, including the parameter prototypes. The function is declared as

```
FSvar * _bb_<className>_<methName>_as  
                                (FSvar *ovSel f, FSvar *assVar);
```

which is equivalent to the

```
ASSIGN METHOD <methName> CLASS <className>
```

declaration in the FlagShip language.

Example:

```
#include "FSopenc.h"
OBJ_DECL_METH (myclass, mymethod)          /* argc, argv */
{
    FSvar *ovSelf;
    if ((argc < 1) || (! IS_VAR_OBJ(argv[0])) {
        fprintf(stderr, "\nWrong param count");
        return NIL_VAR;
    }
    ovSelf = argv[0];
    printf ("\n%d parameters received", argc -1);
    return ovSelf;
}

OBJ_DECL_METHACCESS (myclass, myinst)      /* ovSelf */
{
    printf ("\nReturning contents of instance 'abc' ");
    return OBJ_ACCEXEC (ovSelf, "abc", -1);
}

OBJ_DECL_METHASSIGN (myclass, myinst)     /* ovSelf, assVar */
{
    printf("\nAssigning value of type %c to instance 'abc'",
        VAR_I STYPE(assVar));
    OBJ_ASSEXEC (ovSelf, "abc", -1, assVar);
    return OBJ_ACCEXEC (ovSelf, "abc", -1);
}
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

OBJ_xxxEXEC(), CB4rdd.tar.Z, ascirdd.tar.Z

OBJ_METHEXEC ()

OBJ_ACCEXEC ()

OBJ_ASSEXEC ()

Syntax:

```
[varPtr = ] OBJ_METHEXEC (methName, methNo, argc,  
                        varArg)
```

```
[varPtr = ] OBJ_ACCEXEC (objVar, methName, methNo)
```

```
[varPtr = ] OBJ_ASSEXEC (objVar, methName, methNo,  
                        varPtr)
```

Purpose:

Executes the specified object Method (or an access/ assign method).

Arguments:

<**objVar**> is a FlagShip pool variable (FSvar *) containing the instantiated object.

<**methName**> is a string (char *) containing the method (or access/assign method) name. Note, that the METHOD name is not abbreviated, but significant in the given length. The ACCESS and ASSIGN method names are abbreviated to 10 significant characters. The given <methName> is converted to lower case.

<**methNo**> is an (int) number corresponding to the internal low- level order in the class for direct array-like access (see corresponding #define's in produced .c code). If the internal order number is unknown, specify it -1 for a look-by-name processing. Late binding (refer to LNG.11.6) is always performed. For early binding see VAR_OBJ().

<**varPtr**> is a FlagShip pool variable (FSvar *) containing the assigned value for the ASSIGN method.

<**argc**> is an (int) number containing the number of passed arguments, i.e. the length of <varArg>. At least one argument (the object variable) has to be passed to the usual METHOD.

<**varArg**> is an array of FlagShip pool variables (FSvar *[]) containing a) the instantiated object and b) the optional parameters passed to the METHOD.

Returns:

<**varPtr**> is the value (FSvar *) returned from the METHOD or the ACCESS, ASSIGN method.

Description:

OBJ_METHEXEC() allows you to invoke the specified object method, pass optional arguments and retrieve the returned value. The corresponding .prg syntax is

```
[var :=] obj : methName([params. . . ])
```

OBJ_ACCEXEC() invokes the specified object ACCESS method, or a visible INSTANCE, retrieving the returned value. The corresponding .prg syntax is

```
val ue := obj : accName -or- val ue := obj : i nstName
```

OBJ_ASSEXEC() invokes the specified object ASSIGN method, passes the required value and optionally retrieves the return value, which is usually equivalent to the passed value. This is equivalent to assigning a value to a visible INSTANCE. The corresponding .prg syntax is

```
obj : assName := val ue -or- obj : i nstName := val ue
```

Usually, you will prefer to invoke the object by using the simple obj:method() syntax in the FlagShip language, since the FlagShip compiler converts all the required parameters for you and determines also the <methNo> for an early binding, if the class prototype is known.

Example:

```
PROTOTYPE CLASS xyz
  EXPORT      iii      AS IntVar
PROTOTYPE METHOD abc() CLASS xyz
PROTOTYPE ACCESS def      CLASS xyz
PROTOTYPE ASSIGN ghi (val) CLASS xyz
LOCAL myObj := xyz {}

// the following C code is equivalent to:
// LOCAL retVar
//   retVar := myObj : def
//   myObj : ghi := "abcdef"
//   myObj : iii := 1234
//   retVar := myObj : iii
//   myObj : abc (1.0)

#C i n l i n e
{
#i n c l u d e "FSopenc.h"
  FSvar *retVar = VAR_NEW, *argv[2];
  SET_VAR_COPY(retVar,
    OBJ_ACCEXEC(VAR_NAME_LOCAL(myobj), "def", -1) );

  OBJ_ASSEXEC(VAR_NAME_LOCAL(myobj), "ghi", -1,
    SET_VAR_CHR(VAR_NEW, "abcdef") );
  OBJ_ASSEXEC(VAR_NAME_LOCAL(myobj), "iii", -1,
    SET_VAR_INT(VAR_NEW, 1234) );

  SET_VAR_COPY(retVar,
    OBJ_ACCEXEC(VAR_NAME_LOCAL(myobj), "iii", -1) );
  VAR_NEW_ARGS (argv, 2);
  SET_VAR_COPY(argv[0], VAR_NAME_LOCAL(myobj));
  SET_VAR_NUM (argv[1], 1.0);
  OBJ_METHEXEC("abc", -1, 2, argv );
}
#e n d C i n l i n e
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

OBJ_DECL_xxx(), CB4rdd.tar.Z, ascirdd.tar.Z

SET_VAR_ADD ()

Syntax:

```
[varPtr3 =] SET_VAR_ADD (varPtr1, varPtr2);
```

Purpose:

Adds two numeric, date or string variables.

Arguments:

<varPtr1> and <varPtr2> are already initialized FlagShip pool variables (FSvar *) of the type 'N', 'F', 'I', 'D', 'C' or 'M' containing the first and second operator.

Returns:

<varPtr3> is the resulting FlagShip pool variable (FSvar *), containing the result of the addition. Its type will be set to 'N', 'I', 'D' or 'C' according to the type of the operators.

Description:

SET_VAR_ADD() performs the same operation as in `var3 := var1 + var2`. Same as in the FlagShip language, only the same or compatible types (see VAR_IS_TYPE) may be added:

Result varPtr3	varPtr1	varPtr2
N (float numeric)	N, F, (I)	N, F, I
I (integer numeric)	I	I
D (date)	D	N, F, I
C (character)	C, M	C, M

All other combinations will result in a run-time error.

Example:

```
FSvar *text1, *text2, *text3;
FSvar *num1 = VAR_NEW, *num2, *num3 = VAR_NEW;

text1 = SET_VAR_CHR (VAR_NEW, "this is ");
text2 = SET_VAR_CHR (VAR_NEW, "my string");
text3 = SET_VAR_ADD (text1, text2);
num1 = SET_VAR_NUM (num1, 1.56);
num2 = SET_VAR_INT (VAR_NEW, 20);
SET_VAR_COPY (num3, SET_VAR_ADD (num1, num2));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_xxx(), VAR_IS_TYPE()

SET_VAR_BLOCK ()

Syntax:

```
[varPtr =] SET_VAR_BLOCK (varPtr, ptrCfun);
```

Purpose:

Creates and assigns a code block variable to a C function realizing the block body.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_BLOCK() assigns type B (code block) to it, releasing its previous contents.

<ptrCfun> is a pointer (FSvar *(*)(fnName)) to a function of any valid C name, containing the body of the code block. Note: in FlagShip source, such a function is named `_bb_cb_<no-within- line>_<line-no>_<internal-no>`, e.g. declared as

```
FSvar * _bb_cb_1_125_7(int argc, FSvar *argv[])
```

Returns:

<varPtr> points to the original FlagShip variable.

Description:

SET_VAR_BLOCK() assigns any C function (which meets the FlagShip calling convention) to a usual FlagShip code block variable.

Example:

The setmycb() UDF returns a code block variable, which may be evaluated also from .prg e.g. via EVAL(setmycb(), "xyz").

```
UDF_DECL(MyCbFun) /* == FSvar * MyCbFun (argc, argv) */
                  /* int argc; FSvar *argv[]; */
                  /* name is unreachable from .prg ! */
{
    printf ("\n%d parameters passed", argc);
    if ((argc > 0) && IS_VAR_CHR(argv[0]))
        printf ("\n1st parameter is %s", VAR_CHR(argv[0]));
    return NIL_VAR;
}

UDF_DECL(setmycb)
{
    UDF_PROT(MyCbFun); /* extern FSvar *MyCbFun() */
    return SET_VAR_BLOCK (VAR_NEW, (FSvar *(*)( ))MyCbFun);
}
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_BLOCK(), VAR_BLOCK_COMPILE(), VAR_BLOCK_EVAL()

SET_VAR_CHR () SET_VAR_CHRLEN () SET_VAR_LOWER () SET_VAR_UPPER ()

Syntax:

```
[varPtr =] SET_VAR_CHR      (varPtr, strPtr);  
[varPtr =] SET_VAR_CHRLEN (varPtr, strPtr, len);  
[varPtr =] SET_VAR_LOWER  (varPtr);  
[varPtr =] SET_VAR_UPPER  (varPtr);
```

Purpose:

Copies a string into a FlagShip pool variable or converts a variable to lower/upper case.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_CHRxx assigns type C (character) to it, releasing its previous contents.

<strPtr> is an (unsigned char *) string or constant to be copied into the FlagShip variable.

<len> is the (unsigned int) number of bytes to be copied from <strPtr>, including zero-bytes. The <strPtr> must have <len> or more characters; otherwise a segmentation violation may occur. When the last copied byte (i.e. strPtr[len-1]) is not a zero-byte, the function will automatically add it.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

SET_VAR_CHR() copies a zero byte terminated string into the target variable, similar to the C function strcpy().

SET_VAR_CHRLEN() copies the specified number of bytes of the string into the target variable, similar to the C function strncpy(). The string may include any number of zero bytes.

SET_VAR_LOWER() and SET_VAR_UPPER() implace changes the content of C typed variable to lower or upper case.

Example:

```
FSvar    *text1 = 0, *text2 = VAR_NEW;
unsigned char *xx1 = "any string";
unsigned char xx2[20];
int ii;

text1 = SET_VAR_CHR (VAR_NEW, xx1);

for (ii = 0; ii < 20; ii++) xx2[ii] = 'A' + ii;
xx2[5] = xx[9] = 0; /* zero byte in 6th and 10th char */
SET_VAR_CHRLEN (text2, xx2, 20);
printf ("\nstrlen(text2)=%d assigned length=%d\n",
        strlen(VAR_CHR(text2)),
        (int) VAR_ISLEN(text2)); /* 5 21 */

printf ("\nChanging 'text1' [%s] to upper case [%s]",
        VAR_CHR(text1), SET_VAR_UPPER(txt1) );
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_CHR(), IS_VAR_CHR(), SET_VAR_COPY(), VAR_NEW,
VAR_NEW_ARGS(), VAR_NEW_COPY()

SET_VAR_COPY ()

Syntax:

```
[varPtr =] SET_VAR_COPY (varPtr, varSrc);
```

Purpose:

Copies the contents of one FlagShip pool variable into another. Only variable types C, D, M, N, L, S, U can be copied.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. The variable becomes the same type as the source variable <varSrc>, and its previous contents are released.

<varSrc> is a (FSvar *) FlagShip pool variable of type C, D, M, N, L, S, U which had already been initialized, the contents of which are to be copied into the target variable <varPtr>.

Returns:

<varPtr> points to the target FlagShip variable.

Description:

The function copies the contents of an already initialized variable into the target FlagShip pool variable.

Example:

```
FSvar *oldvar = VAR_NEW, *newvar = VAR_NEW;
FSvar *args[1];
VAR_NEW_ARGS (args, 1);           /* init var array */
SET_VAR_CHR (oldvar, "any text"); /* create char var */
SET_VAR_COPY (newvar, oldvar);

SET_VAR_COPY (args[0], newvar);
_bb_qout (1, args);              /* print string */
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_XXX(), VAR_ISTYPE(), VAR_NEW, VAR_NEW_ARGS(),
VAR_NEW_COPY()

SET_VAR_DATE ()

Syntax:

```
[varPtr =] SET_VAR_DATE (varPtr, date);
```

Purpose:

Copies a long value representing the date into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_DATE assigns type D (date) to it, releasing its previous contents.

<date> is an (unsigned long) value representing the date as a number of days since January 1, 0001 AD.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

The function copies the date value into the target FlagShip pool variable and marks it as date.

Example:

```
*** test.prg
#C i n l i n e
#i n c l u d e "FSopenc.h"
#e n d C i n l i n e

LOCAL otherdate, numval
mydate = DATE()
#C i n l i n e
{
    unsigned long date = 727929;          /* January 1, 1994 */
    FSvar *varPtr = VAR_NAME_MEMVAR (mydate);
    FSvar *other = VAR_NAME_LOCAL (otherdate);
    if (IS_VAR_DATE(varPtr))
        date = VAR_DATE(varPtr);
    date += 10;
    SET_VAR_DATE (varPtr, date);          /* change .prg var */
    SET_VAR_DATE (other, date);          /* change .prg var */
    SET_VAR_NUM (VAR_NAME_LOCAL(numval), (double) date);
}
#e n d C i n l i n e
? date(), otherdate, mydate, numval
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility: Available in FlagShip only.

Related:

VAR_DATE(), IS_VAR_DATE(), SET_VAR_COPY(), VAR_NEW, VAR_NAME_xx()

SET_VAR_LOG ()

Syntax:

```
[varPtr =] SET_VAR_LOG (varPtr, bool);
```

Purpose:

Copies an integer value representing the boolean state into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which had already been initialized. SET_VAR_LOG assigns type L (logical) to it, releasing its previous contents.

<bool> is an (int) value representing the boolean state, where zero is interpreted as FALSE, all other values as TRUE.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

The function copies the integer boolean value into the target FlagShip pool variable and marks it as logical.

Example:

```
*** test.prg
LOCAL open_ok := .F.

#C i n l i n e
{
# i n c l u d e "FSopenc.h"
# i n c l u d e "fcntl.h"
i n t h a n d l e ;
FSvar *varPtr = VAR_NAME_LOCAL (open_ok);
h a n d l e = o p e n ("myfi le", O_RDONLY);
i f (h a n d l e > 0) {
    S E T _ V A R _ L O G (varPtr, h a n d l e);          /* success ? */
    c l o s e (h a n d l e);
}
}
#e n d C i n l i n e
? "open of myfi le was " + i f (open_ok, "", "not ") + ;
"successful"
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_LOG(), IS_VAR_LOG(), SET_VAR_COPY(), VAR_NEW

SET_VAR_MACRO ()

Syntax:

```
[varPtr =] SET_VAR_MACRO (varPtr, strPtr);
```

Purpose:

Evaluates a string containing a macro and copies the result into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_MACRO() assigns the type resulting from the macro evaluation to it, releasing its previous contents.

<strPtr> is an (unsigned char *) string or constant containing the macro to be evaluated.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

SET_VAR_MACRO() performs the same functionality as the macro evaluation &(strVar) in the .prg language.

Note, that the C "external" statement used in UDF_PROT() is not equivalent to the FlagShip EXTERNAL command; it does **not** advise the linker to automatically include the specified object file into the application. Therefore, use the EXTERNAL command for standard functions invoked by macro only.

Example:

```
FSvar *retVar = VAR_NEW;
long recno;
UDF_PROT(recno); /* is not equiv. to EXTERN RECNO */
SET_VAR_MACRO(retVar, "{|| RecNo() }");
recno = VAR_INT(VAR_EVAL_BLOCK(retVar));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_CHR(), IS_VAR_CHR(), SET_VAR_COPY(), VAR_NEW,
VAR_NEW_ARGS(), VAR_NEW_COPY()

SET_VAR_NIL ()

Syntax:

```
[varPtr =] SET_VAR_NIL (varPtr);
```

Purpose:

Copies the constant NIL_VAR into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_NIL assigns type U (undefined, NIL) to it, releasing its previous contents.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

This function resets the FlagShip variable to an "undefined" state, just as when initializing it using VAR_NEW or assigning the FlagShip constant NIL in the .prg part.

Example:

Using FlagShip standard functions, set the color to white on blue, clear screen and print NIL, "text" and "other text" starting at the 11th column of the 3rd row.

```
FSvar *args[3];
VAR_NEW_ARGS (args, 3);
SET_VAR_CHR (args[0], "W+/B");
_bb_setcolor (1, args);           /* SET COLOR TO. . */
SET_VAR_NIL (args[0]);
_bb_scroll (0, args);           /* CLEAR SCREEN */
SET_VAR_NUM (args[0], 2.0);
SET_VAR_NUM (args[1], 10.0);
_bb_setpos (2, args);           /* SETPOS(2,10) */
SET_VAR_NIL (args[0]);
SET_VAR_CHR (args[1], "text");
SET_VAR_CHR (args[2], "other text");
_bb_qqout (3, args);           /* ?? NIL, "text".. */
VAR_DELETE;                     /* release all */
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_NIL(), VAR_NEW, VAR_NEW_ARGS()

SET_VAR_NUM ()

SET_VAR_NUMDECI ()

SET_VAR_INT ()

Syntax:

```
[varPtr =] SET_VAR_NUM      (varPtr, doubleNum);
[varPtr =] SET_VAR_NUMDECI (varPtr, doubleNum,
                             deciOut);
[varPtr =] SET_VAR_INT      (varPtr, longNum);
```

Purpose:

Copies a double (or long int) value into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_NUMxx() assigns type "N" (float numeric) to it, releasing its previous contents, whilst SET_VAR_INT() the type "I".

<doubleNum> is a (double) value representing the numeric value.

<deciOut> is an (int) representing the number of decimals to be displayed by the standard FlagShip i/o functions. The valid range is 0 to 16. This setting applies to the output only and does not influences the stored value. Using -1 or the SET_VAR_NUM() function will set the decimal output according to the current SET DECIMALS value, or the default value of two.

<longNum> is a (long) value representing the numeric value.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

The functions copies a double (or long int) value into the target FlagShip pool variable and marks it as floating (or integer) numeric.

Example:

```
FSvar *mynvar = VAR_NEW;
FSvar *myivar = VAR_NEW;
double mypi = M_PI;          /* in math.h = 3.141592... */
int myint = 12345;
SET_VAR_NUM (mynvar, mypi);
SET_VAR_NUMDECI (mynvar, mypi, 8);
SET_VAR_NUM (myivar, (double) myint);
SET_VAR_NUMDECI (mynvar, (double) (myint +2), 0);
SET_VAR_INT (myivar, (long) (myint +2));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NUM(), VAR_INT(), VAR_INTNUM(), VAR_FPNUM(), IS_VAR_NUM(),
IS_VAR_INT(), VAR_NEW, SET_VAR_COPY()

SET_VAR_SCR ()

Syntax:

```
[varPtr =] SET_VAR_SCR (varPtr, winPtr);
```

Purpose:

Copies the contents of a curses WINDOW into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized. SET_VAR_SCR assigns type S (screen) to it, releasing its previous contents.

<winPtr> is a (WINDOW *) pointer to the curses WINDOW structure.

Returns:

<varPtr> points to the original FlagShip variable.

Description:

The function copies the contents of a curses WINDOW structure into the target FlagShip pool variable and marks it as screen.

Example:

```
*** test.prg
LOCAL_INT ytop := 5, xtop := 10
LOCAL      actscr
#Cinline
{
#   include "FSopenc.h"
    UDF_PROT(savescreen);
    FSvar *varPtr = VAR_NEW, *args[4];
    WINDOW *mywind;

    VAR_NEW_ARGS (args, 4);
    SET_VAR_NUM (args[0], (double) ytop);
    SET_VAR_NUM (args[1], (double) xtop);
    varPtr = _bb_savescreen (4, args); /* SAVESCREEN () */
    mywind = VAR_SCR (varPtr);
    /* you may manipulate the window now,
       see e.g. example in EXT.2.9._retscw() */
    SET_VAR_SCR (VAR_NAME_LOCAL(actscr), mywind);
}
#endCinline

RESTSCREEN (ytop, xtop, , , actscr)
```

Include:

<FlagShip_dir>/include/FSopenc.h (#includes also curses.h)

Compatibility: Available in FlagShip only**Related:**

IS_VAR_SCR(), VAR_NEW, VAR_NEW_ARGS(), VAR_NEW_COPY()

SET_VAR_SPECIAL ()

Syntax:

```
[varPtr =] SET_VAR_SPECIAL (varPtr, ptr);
```

Purpose:

Assigns the pointer to user defined memory into a FlagShip pool variable.

Arguments:

<varPtr> is a FlagShip pool variable (FSvar *) of any type which has already been initialized by VAR_NEW.

<ptr> is any pointer, e.g. (int *) to previously initialized memory, e.g. via malloc() .

Returns:

<varPtr> points to the original FlagShip variable.

Description:

The function connects a user defined memory to a FlagShip variable. It is primarily used for special purposes, where a FlagShip variable carries a special pointer (e.g. in instances of classes).

Important: to avoid memory leaks, you have to take care to release the allocated memory before the variable <varPtr> is released by the FlagShip garbage collector (see EXT.4.5.1).

Example:

Manages a user defined class, containing an instance which stores an array of C structure. Also demonstrates, how to access the SELF: object within a method body.

```
*** test.prg
CLASS myClass
  INSTANCE nCount AS IntVAR // element count
  INSTANCE struEmp AS SPECIAL // array of C struct

* INIT() method is autom. invoked by the class creator
* xObj := MyClass [{"Name", nAge}]
*

METHOD init(cName, nAge AS IntVar) CLASS myClass
self: nCount := 0
if val type(cName) == "C"
  self: AddName(cName, nAge)
endif
return self

METHOD AddName(cName, nAge) CLASS myClass
self: nCount ++ // add new element
#Cinline
{
# include "FSopenc.h"
struct employee {
  char name [21];
  int age;
```

```

    };
    FSvar *specPtr;
    int size = VAR_INT (OBJ_ACCEXEC (
        VAR_NAME_LOCPAR(sel f), "ncount", -1 ));
    specPtr = VAR_SPECIAL (OBJ_ACCEXEC (
        VAR_NAME_LOCPAR(sel f), "struemp", -1 ));
    if (size > 1)
        specPtr = (struct specPtr *) realloc(specPtr,
            size * sizeof(empl oye));
    else
        specPtr = (struct specPtr *)
            malloc (size * sizeof(empl oye));
    strncpy (specPtr[size]. name,
        VAR_CHR(VAR_NAME_LOCPAR(cname)), 20)
    specPtr[size]. age =
        (int) VAR_INTNUM(VAR_NAME_LOCPAR(nage));
    SET_VAR_SPECIAL(OBJ_ACCEXEC (
        VAR_NAME_LOCPAR(sel f), "struemp", -1 ),
        specPtr);
}
#endCnline
return

* AXIT() method is autom. invoked when freeing the object
* You must free all previously allocated memory
* for SPECIAL variables. All usual FlagShip variables
* will be freed automatically.
*
METHOD axit() CLASS myClass
#Cnline
{
# include "FSopenc.h"
    if (VAR_INT (OBJ_ACCEXEC (
        VAR_NAME_LOCPAR(sel f), "ncount", -1 ))
        free ( VAR_SPECIAL (OBJ_ACCEXEC (
            VAR_NAME_LOCPAR(sel f), "struemp", -1 ));
        VAR_NAME_OBJINST("struemp")));
}
#endCnline
return

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_SPECIAL(), VAR_SPECIAL(), OBJ_ACCEXEC()

UDF_DECL ()

Syntax:

UDF_DECL (udfName)

Purpose:

Declares the function header for a user defined function (written in C language), callable also from the FlagShip .prg source.

Arguments:

<udfName> is the function name, case sensitive. If the UDF should be available also for the .prg source, the name has to be given in lower case and may contain up to 10 characters.

Note: it is recommended not to place white space within the brackets (...) in this macro, since some cc compilers have problems with a correct interpretation of the embedded spaces (or tabs).

Description:

UDF_DECL() declares a function header of an usual FlagShip UDF (user defined function), including the parameter prototyping. The function is declared as

```
FSvar * _bb_<udfname> (i nt argc, FSvar *argv[]);
```

which is equivalent to

```
FUNCTION <udfName> ([params...])
```

declaraton in the FlagShip language.

Example:

```
*** test.prg

// FUNCTION Di spl ArgType (par1, par2, par3)
// LOCAL retVar := ""
// retVar += if (PCOUNT() >= 1, " " + VALTYPE(par1), "")
// retVar += if (PCOUNT() >= 2, " " + VALTYPE(par2), "")
// retVar += if (PCOUNT() >= 3, " " + VALTYPE(par3), "")
// return retVar

#C i n l i n e
{
# i n c l u d e "FSopenc.h"
UDF_DECL (di spl argty) /* argc, argv */
{
i n t i i ;
c h a r * s t r = " *";
F S v a r * r e t V a r = V A R _ N E W ;
S E T _ V A R _ C H R ( r e t V a r , "" );
f o r ( i i = 1 ; i i < = a r g c ; i i + + ) {
*( s t r + 1 ) = ( c h a r ) V A R _ I S T Y P E ( a r g v [ i i - 1 ] );
S E T _ V A R _ A D D ( r e t V a r , S E T _ V A R _ C H R ( V A R _ N E W , s t r ) );
}
r e t u r n r e t V a r ;
}
```

```
    }  
  }  
#endCinline  
  
LOCAL cVar := "text", aVar := {9, 8, 7}  
? DisplayArgType (cVar, 5, aVar)           // C I A  
return
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

CMD.FUNCTION, UDF_EXEC(), EXT.4.5

UDF_EXEC ()

Syntax:

```
[varPtr = ] UDF_EXEC (udfName) (argc, argv);
```

Purpose:

Executes any standard FlagShip or user defined function (UDF), passing optional parameters to it.

Arguments:

<udfName> is the function name given in lower case. The name has to meet the FlagShip naming convention and may contain up to 10 characters.

Note: it is recommended not to place white space within the brackets (...) in this macro, since some cc compilers have problems with a correct interpretation of the embedded spaces (or tabs).

<argc> is an (int) value representing the number of passed arguments in <argv>. If you don't pass any argument, specify it 0.

<argv> is an array (FSvar *[]) or (variable *[]) of FlagShip variables containing the passed arguments. If you don't pass any argument, enter 0.

Description:

UDF_EXEC() allows you to invoke any standard or user defined function, declared in the FlagShip language as FUNCTION udfname(...) or as PROCEDURE udfname(...). The same declaration in C language appears via UDF_DECL(udfName). Prototype the external functions via UDF_PROT(udfName).

Example:

```
*** test.prg
#C i n l i n e
#i n c l u d e "FSopenc.h"
UDF_DECL (mycudf)                /* argc, argv */
{
    UDF_PROT(dbgoto);             /* prototype */
    UDF_PROT(recno);             /* prototype */
    FSvar *var = VAR_NEW;
    var = UDF_EXEC(recno)(0,0);   /* = RECNO() */
    SET_VAR_COPY (
        VAR_NAME_LOCAL(nrec1), var)); /* copy to nRec1 */
    i f ((argc > 0) && IS_VAR_NUM(argv[0]))
        UDF_EXEC(dbgoto)(1, argv); /* = GOTO 5 */
    SET_VAR_COPY (VAR_NAME_LOCAL(nrec2,
        UDF_EXEC(recno)(0,0) ); /* nRec2=recno() */
    i f (argc > 1)
        return argv[1];
    e l s e
        return NI L_VAR;
}
#endC i n l i n e
```

```
LOCAL nRec1, nRec2, tmp
use mydbf
goto bottom
? "Performing GOTO 5 by C UDF "
tmp := MyCudf(5)
?? "from record", nRec1, if (nRec2 != tmp, "not", "") + ;
"successful "
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

CMD.FUNCTION, UDF_EXEC(), EXT.4.5

UDF_PROT ()

Syntax:

```
UDF_PROT (udfName);
```

Purpose:

Prototypes an external user defined function.

Arguments:

<udfName> is the function name given in lower case. The name has to meet the FlagShip naming convention and may contain up to 10 characters.

Note: it is recommended not to place white space within the brackets (...) in this macro, since some cc compilers have problems with a correct interpretation of the embedded spaces (or tabs).

Description:

UDF_PROT prototypes an external UDF. The macro is expanded to e.g.

```
extern FSvar * _bb_udfName (int, FSvar *[]);
```

Note, that the C "external" statement is not equivalent to the FlagShip EXTERNAL command; it does **not** advise the linker to automatically include the specified object into the application.

Example:

See examples in UDF_EXEC(), SET_VAR_SCR(), VAR_BLOCK_COMPILE() etc.

Include:

```
<FlagShip_dir>/include/FSopenc.h
```

Compatibility:

Available in FlagShip only.

Related:

EXT.4.8, CMD.EXTERNAL

VAR_ARRELEM ()

Syntax:

```
elemPtr = VAR_ARRELEM (arrPtr, elem);
```

Purpose:

Accesses an element of a FlagShip single or multi-dimensional array created in the .prg part or by VAR_NEW_ARRAY().

Arguments:

<arrPtr> is a (FSvar *) pointer to a FlagShip pool variable, already created and initialized by VAR_NEW_ARRAY() or _bb_array().

<elem> is an (int) value representing the required element number in the first array dimension starting at one. The valid range is in the FlagShip language notation, i.e. 1 to VAR_ISDIM(arrPtr). A run-time error will occur otherwise. To access a multi-dimensional or nested array, use VAR_ARRELEM() for each dimension. See example.

Returns:

<elemPtr> is a (FSvar *) pointer to a FlagShip pool variable, representing the array element. This variable (array element) can contain any valid type, including the type "array" (a multi- dimensional and nested array).

Description:

Determines the pointer to the FlagShip array element. Checks the type of the element using VAR_ISTYPE(). To access an element of an any-dimensional array, see example below.

Example:

```
*** test.prg
LOCAL   aaa := {9, 8, 7}, bbb[3, 4]
DECLARE ccc[5, 10, 5, 2]
bbb [2, 3] = .T.
#C i n l i n e
{
#   i n c l u d e "FSopenc.h"
/* #d e f i n e ALTERNATIVE_METHOD */

FSvar *elemPtr = 0, *varPtr = 0;
double value = 0.0;
int type;

varPtr =VAR_NAME_LOCAL(aaa);
type   =VAR_I STYPE(varPtr);           /* == 'A' */
elemPtr=VAR_ARRELEM(varPtr, 3);       /* aaa[3] */
value  =(I S_V A R_N U M(e l e m P t r) ? V A R_N U M(e l e m P t r) : 0.0);
SET_VAR_NUM (e l e m P t r, 55.0);    /* aaa[3] = 55 */

varPtr =VAR_NAME_LOCAL(bbb);

#i f d e f ALTERNATIVE_METHOD
```

```

type =VAR_I STYPE(varPtr); /* == 'A' */
el emPtr=VAR_ARRELEM(varPtr, 2); /* bbb[2] */
type =VAR_I STYPE(el emPtr); /* == 'A' */
el emPtr=VAR_ARRELEM(el emPtr, 3); /* bbb[2, 3] */
type =VAR_I STYPE(el emPtr); /* == 'L' */
#el se
el emPtr= /* al ternat.*/
VAR_ARRELEM(VAR_ARRELEM(varPtr, 2), 3); /* bbb[2, 3] */
#endi f
SET_VAR_NUM (el emPtr, 123.45); /* bbb[2, 3] = 123.45 */
varPtr =VAR_NAME_MEMVAR(ccc);
el emPtr=VAR_ARRELEM(varPtr, 1); /* ccc[1] */
el emPtr=VAR_ARRELEM(el emPtr, 1); /* ccc[1, 1] */
el emPtr=VAR_ARRELEM(el emPtr, 3); /* ccc[1, 1, 3] */
el emPtr=VAR_ARRELEM(el emPtr, 2); /* ccc[1, 1, 3, 2] */
SET_VAR_CHR (el emPtr, "xx"); /* ccc[1, 1, 3, 2] = "xx" */
}
#endC i n l i n e
AEVAL (aaa, { |el em| QOOUT(el em) }) // 9 8 55.00
? bbb[2, 2], bbb[2, 3] // NI L 123.45
? ccc[1, 1, 3, 1], ccc[1, 1, 3, 2] // NI L xx

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_ARR(), VAR_NEW_ARRAY(), FUN.ARRAY(), LNG.2.6.4, VAR_NEW,
VAR_NEW_ARGS()

VAR_BLOCK ()

Syntax:

```
fnPtr = VAR_BLOCK (varPtr);
```

Purpose:

Retrieves the address of a function representing a code block. Not applicable for a macro-evaluated code block (refer to LNG.2.3) which is internally stored as a character string and evaluated by the run-time macro system.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already initialized as type B (code block).

Returns:

<fnPtr> is a (FSvar *fn()) pointer to the code block function.

Description:

The code block is an (internally named) function, which is usually assigned to a FlagShip variable and executed by means of EVAL(), AEVAL() or DBEVAL().

Example:

```
*** test.prg
LOCAL myblock := { |par| QOUT (par) }

#Cinline
{
# include "FSopenc.h"
FSvar *(*blkPtr)();
FSvar *args[1];
VAR_NEW_ARGS (args, 1);
blkPtr = VAR_BLOCK (VAR_NAME_LOCAL (myblock));
SET_VAR_CHR (args[0], "text from inline C part");
(*blkPtr)(1, args); /* = EVAL(myblock, "text..") */
}
#endCinline

EVAL (myblock, "text from .prg part")
```

Note: The above direct evaluation of a code block is valid only for compiled code blocks, which do not use variables other than block parameter variables. In all other cases, use `_bb_eval()` or `VAR_BLOCK_COMPILE()` and `VAR_BLOCK_EVAL()` instead.

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility: Available in FlagShip only

Related:

SET_VAR_BLOCK(), VAR_BLOCK_COMPILE(), VAR_BLOCK_EVAL(),
IS_VAR_BLOCK(), VAR_NEW, VAR_NEW_ARGS()

VAR_BLOCK_COMPILE ()

Syntax:

```
[varPtr2 =] VAR_BLOCK_COMPILE (varPtr1);
```

Purpose:

Creates a code block variable by compiling the given string variable.

Arguments:

<varPtr1> is an already initialized FlagShip pool variable (FSvar *) containing the body of the resulting code block. On any type other than C, ".F." is assumed.

Returns:

<varPtr2> is an already initialized FlagShip pool variable (FSvar *). VAR_BLOCK_COMPILE() assigns type B (code block) to it, releasing its previous contents.

Description:

VAR_BLOCK_COMPILE() creates a code block variable from the given string variable. The string may contain any valid expression of the FlagShip language. This allows you, together with VAR_BLOCK_EVAL(), a simple access to any FlagShip standard or a user defined function, alternative to invoking them directly via UDF_EXEC() (or according to chapter 4.6). It is similar to the syntax

```
Var := &("{|" + cVar + "|"}")
```

specified in the .prg source.

Example:

```
FSvar *text;
FSvar *block, *num;
text = SET_VAR_CHR (VAR_NEW, "LastRec() - RECNO()");
block = VAR_BLOCK_COMPILE (text);
num = VAR_BLOCK_EVAL (block);

printf ("%l d records remaini ng\n", VAR_INT(num));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_BLOCK_EVAL(), SET_VAR_MACRO(), UDF_EXEC(), EXT.4.6

VAR_BLOCK_EVAL ()

Syntax:

```
[varPtr2 =] VAR_BLOCK_EVAL (varPtr1);
```

Purpose:

Evaluates a code block variable, similar to the EVAL() function.

Arguments:

<varPtr1> is an already initialized FlagShip pool variable (FSvar *) containing a code block variable. You may check it via IS_VAR_BLK().

Returns:

<varPtr2> is an already initialized FlagShip pool variable (FSvar *) containing the result of the code block evaluation.

Description:

VAR_BLOCK_EVAL() is the C equivalent of the standard FlagShip EVAL() function. Together with VAR_BLOCK_COMPILE(), it allows you a simple access to any FlagShip standard or a user defined function, alternative to invoking them directly via UDF_EXEC() (or according to chapter 4.6). It is similar to the syntax

```
resVar := EVAL(bVar)
```

specified in the .prg source.

Example:

```
FSvar *num;
num = VAR_BLOCK_EVAL ( VAR_BLOCK_COMPILE (
    SET_VAR_CHR (VAR_NEW, "LastRec() - RecNo()") ) );
printf ("%ld records remaining\n", VAR_INT(num));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_BLOCK_COMPILE(), UDF_EXEC(), EXT.4.6

VAR_CHR ()

Syntax:

```
strPtr = VAR_CHR (varPtr);
```

Purpose:

Accesses the string (pointer) stored in a FlagShip character variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already initialized as type C (character).

Returns:

<strPtr> is an (unsigned char *) pointer to the first byte of a zero-byte terminated string.

Description:

Any FlagShip character variable contains a dynamically allocated string at least one byte long (the zero terminator). The currently allocated string length can be determined using VAR_ISLEN(varPtr).

Changing the contents of the string will also directly change the contents of the FlagShip variable (or the database FIELD).

You may shorten the current string by overwriting any **included** character with a zero byte. The new string length will be determined correctly by C functions such as strlen() or in FlagShip program when FS_SET("zero") is not active. **Never** expand or overwrite the stored string behind VAR_ISLEN(varPtr). Changing the pointer value <strPtr> will crash your application unconditionally.

The safest way is to copy the string starting at <strPtr> into locally allocated memory and/or assign a new string or the string changes to the FlagShip variable by using the SET_VAR_CHRxx() function.

Note: the internal FlagShip variable type returned by IS_VAR_TYPE() may be 'C', 'E', 'M', 'V'; hence a comfortable check for character type is the IS_VAR_CHR() macro. All these types are accessed by VAR_CHR() which returns pointer to it first char.

Example:

```
** test.prg
#C i n l i n e
# i n c l u d e "FSopenc.h"
# i n c l u d e "string.h"
#endC i n l i n e

LOCAL abc := "text"
LOCAL_INT allocated

#C i n l i n e
{
// #defi ne TEST_ONLY
FSvar *varPtr = VAR_NAME_LOCAL(abc); // ptr to FS vari abl e
```

```

unsigned char *strPtr, mystr[100], *newstr;
int ii;
for (ii=0; ii < 99; ii++)
    mystr[ii] = 'A' + ii;           // init mystr: "ABCDEFGG..."
mystr[99] = '\0';                 // terminate string

#ifdef TEST_ONLY
    strPtr = VAR_CHR(varPtr);      // ptr to variable-string
    ii = strlen(strPtr);           // = 4
    fprintf(stderr, "ii = %d\n", ii); // output to stderr/console
    strPtr[4] = 'X';               // crash follows (length)
    strPtr[5] = 'X';               // crash follows (length)
    strcpy(strPtr, mystr);         // crash follows (size)
    strPtr = mystr;                // crash follows (local)
#endif

    strPtr = VAR_CHR(varPtr);      // ptr to variable-string
    strPtr[2] = 'X';               // var abc: "teXt"
    strncpy(strPtr, mystr, 4);     // var abc: "ABCD"

    SET_VAR_CHR(varPtr, mystr);   // var abc: "ABCDEFGG..."

    newstr = (unsigned char *) malloc(
        (unsigned int)(VAR_ISLEN(varPtr) +50) );
    if (newstr != NULL) {         // Using malloc is
        strcpy(newstr, VAR_CHR(varPtr)); // possible. Copy
string,
        strcat(newstr, " additional text"); // and append your text.
        SET_VAR_CHR(varPtr, newstr);     // But: first assign,
        free(newstr);                   // than free it!
    }
    allocated = VAR_ISLEN(varPtr);     // string size of var abc
}
#endif

? "len(abc)=" , LEN(abc), "alloc.length=", allocated
? "abc= '" + abc + "'"
wait

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_CHR(), SET_VAR_CHR(), VAR_NEW, VAR_NEW_ARGS(),
 SET_VAR_COPY()

VAR_DATE ()

Syntax:

```
value = VAR_DATE (varPtr);
```

Purpose:

Accesses the long value representing a FlagShip date variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already initialized as type D (date).

Returns:

<value> is a (long) value representing the FlagShip date equivalent.

Description:

Any FlagShip date variable is stored as a long value representing the date as the number of days since January 1, 0001 AD.

To change the stored value, use the SET_VAR_DATE() function.

Example:

```
FSvar *varPtr = VAR_NEW, *args[4];
long  datValue;
varPtr = _bb_date (0,0);           /* std. funct. DATE() */
datValue = VAR_DATE (varPtr);
VAR_NEW_ARGS (args, 4);
SET_VAR_CHR (args[0], "Current date is");
SET_VAR_DATE (args[1], varPtr);
SET_VAR_CHR (args[2], "and the sum of days is");
SET_VAR_NUM (args[3], (double) datValue);
_bb_qout (4, args);               /* ? command */
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_DATE(), SET_VAR_DATE(), VAR_NEW, VAR_NEW_ARGS(),
SET_VAR_COPY()

VAR_DELETE

VAR_DEL_ONEVAR()

Syntax:

```
VAR_DELETE;  
VAR_DEL_ONEVAR (varPtr);
```

Purpose:

VAR_DELETE frees the allocated memory for all temporary FlagShip pool variables or an array of variables.

VAR_DEL_ONEVAR() frees the allocated memory for specified pool variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable which should be freed.

Returns:

nothing.

Description:

Deletes one or all newly allocated FlagShip temporary variables created by VAR_NEW, VAR_NEW_COPY(), VAR_NEW_ARGS() or VAR_NEW_ARRAY() and releases their assigned contents. Variables created by VAR_NEW_STATIC() are not released.

After invoking VAR_DELETE, new variables can be created and initialized again by VAR_NEW or VAR_NEW_xxx(), e.g. to use them as a return value.

Note the scope and visibility of the FlagShip pool variables, described in chapter 4.5.1. Variables not explicitly deleted will be released by the internal garbage collection.

Example:

```
FSvar *myvar = VAR_NEW, *args[2];  
char *str = "any string";  
SET_VAR_CHR (myvar, str);  
VAR_NEW_ARGS (args, 2);           /* initialize args[] */  
SET_VAR_COPY (args[0], myvar);  
SET_VAR_NUM (args[1], 1.234);  
_bb_qout (2, args);  
  
:  
VAR_DELETE;           /* release contents of myvar and args[] */  
  
myvar = VAR_NEW;      /* initialize variable again */  
SET_VAR_CHR (myvar, str); /* and copy string into */  
:  
VAR_DELETE (myvar);   /* release contents of myvar */
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, VAR_NEW_ARGS(), VAR_NEW_COPY(), SET_VAR_COPY(),
VAR_DEL_SINCE_MARK()

VAR_DEL_MARK() VAR_DEL_SINCE_MARK()

Syntax:

```
value = VAR_DEL_MARK ( );  
VAR_DEL_SINCE_MARK (value);
```

Purpose:

VAR_DEL_MARK() specifies the point from which all subsequently allocated FlagShip pool variables should be deleted with VAR_DEL_SINCE_MARK().

Argument, Return:

<value> is a (int) value specifying the start of the memory pool freeing.

Description:

As opposite to VER_DELETE, which frees all locally allocated FlagShip pool variables, VAR_DEL_SINCE_MARK() allows you to free the allocated memory for a defined set of FlagShip variables, similarly to multiple invocation of VAR_DEL_ONEVAR(). You may set different deletion levels by multiple invocation of VAR_DEL_MARK().

Note the scope and visibility of the FlagShip pool variables, described in chapter 4.5.1. Variables not explicitly deleted will be released by the internal garbage collection.

Example:

```
FSvar *myvar = VAR_NEW  
char *str = "any string";  
int mymark = VAR_DEL_MARK(); /* mark start */  
FSvar *tmp = VAR_NEW, *args[2];  
SET_VAR_CHR (myvar, str);  
VAR_NEW_ARGS (args, 2); /* initialize args[] */  
SET_VAR_COPY (args[0], myvar);  
SET_VAR_NUM (args[1], 1.234);  
fsFnName(qout)(2, args); /* call QOUT(...) */  
VAR_DELETE_SINCE_MARK(mymark); /* release tmp, args[] */  
/* myvar is still valid*/
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_DELETE, VAR_DEL_ONEVAR(), VAR_NEW_ARGS(), VAR_NEW,
VAR_NEW_COPY(), SET_VAR_COPY()

VAR_ISDECI ()

Syntax:

```
value = VAR_ISDECI (varPtr);
```

Purpose:

Retrieves the number of decimal digits to be displayed when using the specified FlagShip numeric variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already initialized as type N (numeric).

Returns:

<value> is an (int) value representing the number of decimal digits to be displayed when SET FIXED is ON. The <value> range may be from 0...16 decimal digits or determined dynamically with -1, which will display decimal numbers according to the current SET DECIMALS and SET FIXED values.

Description:

Any FlagShip numeric variable is stored as a double precision floating point (usually 64 bits). The number of decimal digits set by SET_VAR_NUMDECI() does not influence the stored value, but is used for display purposes (?, ??, QOUT(), QQOUT() etc.) only. To change the stored value, use SET_VAR_NUM() or SET_VAR_NUMDECI() functions.

Example:

```
*** test.prg
abc := 1234.5432
? abc                                     // 1234.54

#C i n l i n e
{
#   i n c l u d e "FSopenc.h"
    FSvar *var = VAR_NAME_MEMVAR(abc);
    i f (IS_VAR_NUM(var) && VAR_ISDECI(var) < 5)
        SET_VAR_NUMDECI (var, var, 5); /* SET DECIM TO 5 */
}
#endC i n l i n e

SET FIXED ON
? abc                                     // 1234.54320
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_NUM(), SET_VAR_NUM(), SET_VAR_NUMDECI(), VAR_INT()

VAR_ISDIM ()

Syntax:

```
value = VAR_ISDIM (arrPtr);
```

Purpose:

Retrieves the length of a FlagShip array.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable which has already been initialized by VAR_NEW_ARRAY() or _bb_array() as type A (array).

Returns:

<value> is an (int) value representing the length of the array <varPtr> or the sub-array of a multi-dimensional or nested array.

Description:

Determines the dimension and size of a FlagShip variable of type "array" created in the .prg part or by using the VAR_NEW_ARRAY() function.

Example:

```
*** test.prg
#C i n l i n e
# i n c l u d e "FSopenc.h"
#e n d C i n l i n e
LOCAL_I N T arrl n g
xxx = ARRAY (2,3,4)           // creates PRIVATE xxx[2,3,4]

#C i n l i n e
arrl n g = VAR_I S D I M (VAR_NAME_MEMVAR(xxx));    /* == 2 */
arrl n g = VAR_I S D I M (
    VAR_ARRELEM(VAR_NAME_MEMVAR(xxx), 1));        /* == 3 */
arrl n g = VAR_I S D I M (
    VAR_ARRELEM( VAR_ARRELEM(
        VAR_NAME_MEMVAR(xxx), 1), 1 ));          /* == 4 */
#e n d C i n l i n e

? arrl n g                               // 4
? LEN(xxx), LEN(xxx[1]), LEN(xxx[1,1])       // 2 3 4
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, VAR_NEW_ARRAY(), FUN.ARRAY(), LNG.2.6.4

VAR_ISFLDPOS ()

VAR_ISFLDWA ()

Syntax:

```
value = VAR_ISFLDPOS (varPtr);
```

```
value = VAR_ISFLDWA (varPtr);
```

Purpose:

Determines the ordinal field position or the working area of a specified field variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip field variable, already initialized by VAR_NEW and assigned to a field by VAR_NAME_FIELD().

Returns:

<value> is a (unsigned short) length of the field position (starting by 1) or the corresponding working area (starting by 1).

Description:

A field variable is in FlagShip handled very similar to a usual memory pool variables. The significant difference is, that the variable points indirectly to the internal buffer of the database record. You should therefore not write or manipulate these variables directly, but by the SET_VAR_xxx() macros only. Of course, you also cannot change the type of the field variable.

Example:

```
*** test.prg
USE mydbf index myindex
if !used() ; quit ; endif
if type("name") == "U"
    ? "Sorry, field 'NAME' does not exist"
else
    ? 'Field "Name" is the ' + ;
    ltrim(str(fieldpos("Name"))) ".th field in wa " + ;
    ltrim(str(select())) + ", is of the type", ;
    type("name"), "and is", fieldlen("name"), "long"
    if val type(name) == "C"
        ? "Replaci ng '" + name + "' wi th 'any-string' now"
        replac name wi th "any-string"
    endif
endif
? "--- and now the same in C ---"
?
#Cinline
{
# include "FSopenc.h"
FSvar *field1 = VAR_NEW;
field1 = VAR_NAME_FIELD ("name");
if (IS_VAR_NIL(field1))
    printf ("Sorry, field 'NAME' does not exist\n");
else {
```

```

printf("Field \"NAME\" is the %d.th field in wa %d",
      VAR_I SFLDPOS(fi el d1), VAR_I SFLDWA(fi el d1) );
printf(", is of type %c and is %d long\n",
      VAR_I STYPE(fi el d1), VAR_I SLEN(fi el d1));
if (IS_VAR_CHR(fi el d1)) {
    printf ("Repl acing '%s' by 'foo' now\n");
    VAR_CHR(fi el d1), SET_VAR_CHR(fi el d1, "foo");
}
}
printf ("Press any key...");
}
#endC i n l i n e
i nkey(0)
REFRESH

? "Check: fi el d Name = '" + name + "'"

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, VAR_NAME_FIELD(), SET_VAR_xxx()

VAR_ISLEN ()

Syntax:

```
value = VAR_ISLEN (varPtr);
```

Purpose:

Determines the size of allocated memory for the contents of a character variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable of type 'C', already initialized by VAR_NEW and filled by SET_VAR_CHR().

Returns:

<value> is a (long) length of the allocated string, including trailing (and embedded) zero-bytes.

Description:

Any FlagShip variable of the type "character" contains at least one, and up to 2,147,483,647 (two gigabytes), including the trailing zero-byte.

Note: The allocated space for the string is automatically deleted when assigning the variable a new contents by using SET_VAR_xxx(), by deleting the whole variable using the VAR_DELETE macro, or by the internal garbage collection described in chapter 4.4.1.

Example:

```
*** test.prg
PRIVATE myvar := SPACE(1000)
LOCAL_LONG len1, len2
#C i n l i n e
{
#   i n c l u d e "FSopenc.h"
    FSvar *intern = VAR_NEW;
    SET_VAR_CHR (intern, "1234567890");
    len1 = VAR_ISLEN( VAR_NAME_MEMVAR (myvar));
    len2 = VAR_ISLEN( intern);
}
#endC i n l i n e
? len1, len2 // 1001 11
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, SET_VAR_CHR(), SET_VAR_CHRLEN()

VAR_ISMODE ()

Syntax:

```
value = VAR_ISMODE (varPtr);
```

Purpose:

Retrieves the mode of a FlagShip variable or the array element.

Arguments:

<varPtr> is a (FSvar *) pointer to an already initialized FlagShip pool variable.

Returns:

<value> is a (char) value representing the variable mode, which is an additional piece of information to the variable type:

'T' or 't'	temporary pool-variable
'N' or 'n'	standard FlagShip variable
'F'	database field or memo
'V' or 'v'	parameter passed by value
'S' or 's'	instance variable of any type
'M' or 'm'	not assignable instance variable
'P' or 'p'	instance variable M+S
'L' or 'l'	soft code-block
'C'	constant, will not be destroyed by SET_VAR_xx()
'U'	unknown autoPRIVATE variable

Description:

Determines the additional info of a FlagShip pool variable. The function returns a single character as described above.

Example:

```
LOCAL aa := DATE(), result := "t/m, t/m"
#Cinline
{
# include "FSopenc.h"
variable *myvar = VAR_NEW;
char *strPtr;
SET_VAR_CHR (myvar, "");
strPtr = VAR_CHR( VAR_NAME_LOCAL(result));
strPtr[0] = VAR_I STYPE (VAR_NAME_LOCAL(aa));
strPtr[2] = VAR_I SMODE (VAR_NAME_LOCAL(aa));
strPtr[5] = VAR_I STYPE (myvar);
strPtr[7] = VAR_I SMODE (myvar);
}
#endCinline
? "type/mode of aa, myvar:", result // ...: D/N, C/T
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, VAR_NEW_ARGS(), VAR_NEW_COPY()

VAR_ISTYPE ()

Syntax:

```
value = VAR_ISTYPE (varPtr);
```

Purpose:

Retrieves the type of a FlagShip variable or an array element.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable which has already been initialized.

Returns:

<value> is a (char) value representing the variable type: 'C' (character constant or long string), 'E' (short character string), 'M' (dbt content), 'V' (dbv content), 'N' or 'F' (numeric), 'I' or 'i' (integer), 'L' (logical), 'D' (date), 'S' (screen), 'A' (array), 'O' (object), 'B' or 'b' (code block), 'U' (NIL variable).

Description:

Determines the (internal) type of a FlagShip variable. The function returns a single character, similar to the standard TYPE() or VALTYPE() functions.

There are comfortable macros for detection of standard types, e.g. IS_VAR_CHR() automatically reports the type C,E,M,V as of character type, all of them are accessed by VAR_CHR() to get char *.

Similarly, IS_VAR_NUM() checks for 'I','i','N' and 'F', the general access is either by VAR_INTNUM() to get integer, or by VAR_FPNUM() to get double, regardless the VAR_ISTYPE() of I,i,N,F.

Example 1:

```
*** test.prg
LOCAL aa, bb := 10
xx = "test"
#C i n l i n e
{
#i n c l u d e "FSopenc.h"
  FSvar *varPtr, *myvar = VAR_NEW, *par[2];
  char *result = " , , , ";

  SET_VAR_LOG (myvar, 1);
  varPtr = VAR_NAME_LOCAL(aa);
  result[0] = VAR_ISTYPE(varPtr);
  result[2] = VAR_ISTYPE(VAR_NAME_LOCAL(bb));
  result[4] = VAR_ISTYPE(VAR_NAME_MEMVAR(xx));
  result[6] = VAR_ISTYPE(myvar);
  VAR_NEW_ARGS (par, 2);
  SET_VAR_CHR(par[0], "Var type of aa, bb, xx, i n l i n e:");
  SET_VAR_CHR(par[1], result);
  _bb_qout (2, par);
}
#endC i n l i n e
```


VAR_LOG ()

Syntax:

```
value = VAR_LOG (varPtr);
```

Purpose:

Returns an integer value representing the state of a FlagShip logical (boolean) variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already initialized as type L (logical).

Returns:

<value> is an (int) zero value for FlagShip FALSE value, or one for TRUE.

Description:

Any FlagShip logical variable is stored as one character byte. The LOG_VAR() function translates this character to an integer value, which is often used in C for boolean comparisons.

To change the stored value, use the SET_VAR_LOG() function.

Example:

```
** test.prg
LOCAL status := .T.
PUBLIC abc
#C i n l i n e
{
#i n c l u d e "FSopenc.h"
    FSvar *varPtr = VAR_NEW, *args[4];
    int bool;
    VAR_NEW_ARGS (args, 4);
    varPtr = VAR_NAME_LOCAL (status);
    bool = VAR_LOG (varPtr);
    SET_VAR_CHR (args[0], "status --> bool is");
    SET_VAR_NUM (args[1], (double) bool);
    _bb_qout (2, args);
    SET_VAR_LOG (VAR_NAME_MEMVAR (abc), 1);
}
#endC i n l i n e
? status, abc // .T. .T.
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_LOG(), SET_VAR_LOG()

VAR_NAME_FIELD ()

VAR_NAME_MEMVAR ()

Syntax:

```
varPtr = VAR_NAME_FIELD (name);
```

```
varPtr = VAR_NAME_MEMVAR (name);
```

Purpose:

Determines the full name of a database field or a FlagShip memory variable.

Arguments:

<name> is a literal specifying the variable name used in the .prg program. The literal <name> must be specified in lowercase and abbreviated to 10 significant characters.

Returns:

<varPtr> is a (FSvar *) pointer to the FlagShip FIELD or dynamic variable declared in the .prg part.

Description:

VAR_NAME_FIELD() determines the full variable name of a database field or a PRIVATE, PUBLIC variable specified in a .prg program, where the FIELD is preferred. If the field <name> is not found in the current working area, the function scans the list of dynamically scoped variables. To check whether the variable is a database field, use IS_VAR_FIELD(varPtr) or IS_VAR_FIELD (VAR_NAME_FIELD (name)) function as described below. Assigning a value to a FIELD variable by means of SET_VAR_xxx() replaces the database field contents.

VAR_NAME_MEMVAR() determines the full variable name of a PRIVATE, PUBLIC, or PARAMETERS (dynamic scoped) variable specified in a .prg program.

Note: some older C compilers (preprocessors) do not support this macro. If you get a C compiler error, specify the variable name directly as given in the FSopenc.h file, e.g.

```
mv_names[_bbvar_name].f instead of VAR_NAME_FIELD (name), or  
mv_names[_bbvar_name].v instead of VAR_NAME_MEMVAR (name).
```

If the memory variable is not declared in the same .prg file, use VAR_NAME_MEMVAR_BYNAME_OR_NEW("varName") or VAR_NAME_MEMVAR_BYNAME_OR_NULL("varName") instead, which searches in the PRIVATE or PUBLIC variables pool instead of context of current .prg file. Similarly the VAR_NAME_FIELD_BYNAME_OR_ERR("fieldName") or VAR_NAME_FIELD_BYNAME_OR_NULL("fieldName") for database fields.

Example:

```
PRIVATE abc := 1  
PUBLIC def := NIL  
FIELD firstname  
  
USE address // fields FIRSTNAME and CITY
```

```

#C i n l i n e
#i n c l u d e " F S o p e n c . h "
{
    char *first = 0, *cityPtr = 0;
    if (VAR_I STYPE(VAR_NAME_F I E L D(firstname)) == 'C' ) {
        first = VAR_CHR(VAR_NAME_F I E L D(firstname));
        if (VAR_I S L E N(VAR_NAME_F I E L D(firstname)) > 5)
            first[5] = 'X';          /* replace F I E L D contents */
    }
    if (VAR_I STYPE(VAR_NAME_F I E L D(c i t y)) == 'C' &&
        I S_V A R_F I E L D (VAR_NAME_F I E L D(c i t y)) &&
        VAR_I S L E N (VAR_NAME_F I E L D(c i t y)) > 0) {
        c i t y P t r = VAR_CHR(VAR_NAME_F I E L D(c i t y));
        c i t y P t r [ 0 ] = (char) toupper(c i t y P t r [ 0 ]);
        SET_VAR_CHR(VAR_NAME_F I E L D(c i t y), c i t y P t r);
    }
}
#e n d C i n l i n e
? f i r s t n a m e , c i t y                // both are changed

xyz = "text"
#C i n l i n e
{
    FSvar *var;
    var = VAR_NAME_M E M V A R (abc);
    if (VAR_I STYPE(var) == 'N' )
        SET_VAR_NUM (var, VAR_NUM(var) + 10.0);
    var = VAR_NAME_M E M V A R (xyz);
    if (VAR_I STYPE(var) == 'C' && VAR_I S L E N (var) > 2)
        VAR_CHR(var)[2] = 'X';
    SET_VAR_LOG(VAR_NAME_M E M V A R (def), 1);
}
#e n d C i n l i n e
? abc, def, xyz                // 11 .T. teXt

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NAME_LOCAL(), VAR_NAME_STATIC(), SET_VAR_xx(),
VAR_NAME_MEMVAR_BYNAME_OR_NEW(),
VAR_NAME_MEMVAR_BYNAME_OR_NULL(),
VAR_NAME_FIELD_BYNAME_OR_ERR(),
VAR_NAME_FIELD_BYNAME_OR_NULL()

VAR_NAME_MEMVAR_BYNAME_OR_NEW() VAR_NAME_MEMVAR_BYNAME_OR_NULL() VAR_NAME_FIELD_BYNAME_OR_ERR(cName) VAR_NAME_FIELD_BYNAME_OR_NULL(cName)

Syntax:

```
varPtr = VAR_NAME_MEMVAR_BYNAME_OR_NEW (cName);  
varPtr = VAR_NAME_MEMVAR_BYNAME_OR_NULL (cName);  
varPtr = VAR_NAME_FIELD_BYNAME_OR_ERR (cName);  
varPtr = VAR_NAME_FIELD_BYNAME_OR_NULL (cName);
```

Purpose:

Determines the FlagShip memory variable by searching current PRIVATE and PUBLIC pool, or the full name of a database field.

Arguments:

<cName> is character string specifying the memory variable or field name given in lower case. If the requested variable name is longer than 10 characters, abbreviate the <cName> to 10 characters.

Returns:

<varPtr> is a (FSvar *) pointer to the FlagShip FIELD or dynamic variable declared in some .prg source, or NIL var or NULL pointer.

Description:

VAR_NAME_MEMVAR_BYNAME_OR*() is similar to VAR_NAME_MEMVAR() except that it searches at run-time in current PRIVATE and PUBLIC variable pool. If the search fails, the <varPtr> contains either new variable containing NIL value, or is set NULL (the pointer returns 0), in dependence on the macro used, see example below.

VAR_NAME_FIELD_BYNAME_OR*() is similar to VAR_NAME_FIELD() except that it searches at run-time for the field name in currently open and selected database.

Example:

```
/* mysource.c  
* compile: FlagShip -c mysource.c  
* and add the object to your link list  
*/  
#include "FSopenc.h"  
UDF_DECL(myfunct)  
{  
    FSvar *myPubVar, *myPrivateVar;  
  
    // search for FlagShip PUBLIC variable named myPublicVariable  
    myPubVar = VAR_NAME_MEMVAR_BYNAME_OR_NEW("mypublicva");  
  
    if(IS_VAR_NIL(myPubVar)) {
```

```

    fprintf(stderr,
            "variable myPublicVariable not found or is NIL\n");
    return FALSE_VAR;
}
// process myPubVar...

// search for FlagShip PRIVATE variable named myPrivateVariable
myPrivateVar = VAR_NAME_MEMVAR_BYNAME_OR_NULL("myprivate");
if(!myPrivateVar) {
    fprintf(stderr, "variable myPrivateVariable not found\n");
    return FALSE_VAR;
}
// process myPrivateVar...

return TRUE_VAR;
}

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NAME_LOCAL(), VAR_NAME_STATIC(), SET_VAR_xx(),
 VAR_NAME_MEMVAR(), VAR_NAME_FIELD()

VAR_NAME_LOCAL () VAR_NAME_LOCPAR ()

Syntax:

```
varPtr = VAR_NAME_LOCAL (varname);
```

```
varPtr = VAR_NAME_LOCPAR (parname);
```

Purpose:

Determines the full variable name as specified in the .prg program part.

Arguments:

<varname> is a literal specifying the variable name declared in the .prg program as LOCAL.

<parname> is a literal specifying the variable name declared in the .prg program as a formal (parenthesized) parameter of a user defined PROCEDURE or FUNCTION.

The literal <varname> or <parname> must be specified in lowercase and abbreviated to 10 significant characters.

Returns:

<varPtr> is a (FSvar *) pointer to the FlagShip variable declared in the .prg part.

Description:

When a LOCAL variable <name> is declared in the .prg source, it is prefixed with _fgslvar_name to avoid conflicts with user defined variables. For the same reason formal (local) parameters are prefixed with _fgspvar_name .

Note: some older C compilers (preprocessors) do not support this macro. If you get a C compiler error, specify the variable name directly as given in the FSopenc.h file, e.g. _fgslvar_name instead of VAR_NAME_LOCAL(name) or _fgspvar_name instead of VAR_NAME_LOCPAR(name).

Example:

```
*** test.prg
xx = "string"
? myfun1 (1, @xx) // 6
? xx // String
QUIT
FUNCTION myfun1 (numPar, StrPar)
LOCAL_INT retval := 0
#Cinline
{
# include "FSopenc.h"
int pos;
if (VAR_ISTYPE(VAR_NAME_LOCPAR(numpar)) == 'N' &&
VAR_ISTYPE(VAR_NAME_LOCPAR(strpar)) == 'C') {
pos = (int) VAR_NUM(VAR_NAME_LOCPAR(numpar));
pos--; /* C string index */
retval = (int) (VAR_I SLEN(
VAR_NAME_LOCPAR(strpar)) -1);
```

```

        if (retval >= posi t)
            VAR_CHR(VAR_NAME_LOCPAR(strpar))[posi t] = (char)
                toupper(VAR_CHR(VAR_NAME_LOCPAR(strpar))[posi t]);
    }
}
#endC i n l i n e
RETURN retval

FUNCTION myfun2 (numPar, StrPar)    // the same as myudf1()
LOCAL_INT retval := 0
IF VALTYPE(numPar) == "N" .and. VALTYPE(StrPar) == "C"
    retval = LEN (StrPar)
    IF numPar <= retval
        STUFF (StrPar, numPar, 1, ;
            UPPER(SUBSTR(StrPar, numPar, 1)) )
    ENDI F
ENDI F
RETURN retval

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NAME_LOCAL(), VAR_NAME_STATIC(), SET_VAR_xx()

VAR_NAME_STATIC ()

Syntax:

```
varPtr = VAR_NAME_STATIC (varname);
```

Purpose:

Determines the full name of a STATIC variable specified in the .prg program part.

Arguments:

<varname> is a literal specifying the variable name declared in the .prg program as LOCAL.

The literal <varname> must be specified in lowercase and abbreviated to 10 significant characters.

Returns:

<varPtr> is a (FSvar *) pointer to the FlagShip STATIC variable declared in the .prg part.

Description:

When a STATIC variable <name> is declared in the .prg source, it is prefixed with _fgssvar_name to avoid conflicts with user defined variables.

Note: some older C compilers (preprocessors) do not support this macro. If you get a C compiler error, specify the variable name directly as given in the FSopenc.h file, e.g. _fgssvar_name instead of VAR_NAME_STATIC (name)

Example:

```
*** test.prg
LOCAL      abc      := 10
STATIC     xyz      := 50
STATIC_LONG typed := 5

#C i n l i n e
# i n c l u d e "FSopenc.h"
# d e f i n e D I F F E R E N T _ U S A G E 1

#i f (D I F F E R E N T _ U S A G E == 1)
    typed = typed + (l o n g) V A R _ N U M (V A R _ N A M E _ S T A T I C (x y z)) +
              (l o n g) V A R _ N U M (V A R _ N A M E _ L O C A L (a b c));

#e l i f (D I F F E R E N T _ U S A G E == 2)
    typed = typed + (l o n g) V A R _ N U M (_f g s s v a r _ x y z) +
              (l o n g) V A R _ N U M (_f g s l v a r _ a b c);

#e l i f (D I F F E R E N T _ U S A G E == 3)
    {
        FSvar *var1, *var2;
        var1 = V A R _ N A M E _ S T A T I C (x y z);
        var2 = V A R _ N A M E _ L O C A L (a b c);
        typed= typed + (l o n g) (V A R _ N U M (v a r 1) + V A R _ N U M (v a r 2));
    }
#e n d i f
```

```
    SET_VAR_NUM( VAR_NAME_STATIC(xyz), (double)(typed+1));  
#endCinline  
? abc, xyz, typed // 10.00 66.00 65.00
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NAME_LOCAL(), VAR_NAME_STATIC(), SET_VAR_xx()

VAR_NEW VAR_NEW_STATIC VAR_NEW_ARGS ()

Syntax:

```
varPtr = VAR_NEW;  
varPtr = VAR_NEW_STATIC;  
VAR_NEW_ARGS (argPtr, size);
```

Purpose:

Allocates memory for a FlagShip variable or an array of variables.

Arguments:

<argPtr> is a (FSvar **) pointer storing the address of the variable structure array.

<size> is an (int) counter specifying the declared size of the array of variables.

Returns:

<varPtr> is the assigned pointer to the variable structure using VAR_NEW.
VAR_NEW_ARGS() returns nothing.

Description:

VAR_NEW creates and initializes a FlagShip pool variable (declared as variable *varPtr) containing a NIL value and assigns its pointer to <varPtr>.

VAR_NEW_STATIC is similar to VAR_NEW, except that the contents of variable initialized by VAR_NEW_STATIC will **not** be released by VAR_DELETE or the FlagShip garbage collection. Use VAR_NEW_STATIC as an executable statement only.

VAR_NEW_ARGS() is similar to VAR_NEW, but creates and initializes an array of variables (declared as variable *varPtr[size]), e.g. to use it for argument passing to a FlagShip standard function, or to store temporary variables.

Note that the array of variables created by VAR_NEW_ARGS() is **not** equivalent to the FlagShip type "A" (array), created by VAR_NEW_ARRAY(); it instead initializes the array of FSvar's to NIL. Note also the scope and visibility of the FlagShip pool variables, described in chapter 4.5.1.

Example 1:

```
FSvar *myvar = 0;           // single variable, yet uninitialized  
FSvar *myarg[5];          // array of 5 FSvar ptrs, yet  
uninitialized  
myvar = VAR_NEW;          // init FSvar to NIL variable  
VAR_NEW_ARGS (myarg, 5); // myarg[0]=VAR_NEW; ... myarg[4]=VAR_NEW;
```

Example 2:

```
DO WHILE Inkey() != 27
  @ 5,5 say counter()
ENDDO

FUNCTION counter()
LOCAL ret
#Cinline
{
#include "FSopenc.h"
  static FSvar *count = 0;      /* note the C static */
  if (!count) {                /* if not yet initialized, */
    count = VAR_NEW_STATIC;    /* init it now */
    SET_VAR_NUM(count, 0.0);   /* with a zero value */
  }
  VAR_NUM(count)++;           /* increase count content */
  SET_VAR_COPY (VAR_NAME_LOCAL(ret), count);
  // VAR_DELETE;             /* count is not deleted */
}
#endCinline
RETURN ret
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_DELETE, VAR_NEW_ARRAY(), VAR_NEW_COPY(), SET_VAR_xx(),
SET_VAR_COPY()

VAR_NEW_ARRAY ()

VAR_NEW_STATIC_ARRAY ()

Syntax:

```
[varPtr =] VAR_NEW_ARRAY (varPtr, dim);  
[varPtr =] VAR_NEW_STATIC_ARRAY (varPtr, dim);
```

Purpose:

Allocates memory for a FlagShip local or static array variable and initializes all its elements with NIL_VAR.

Arguments:

<varPtr> is an uninitialized (FSvar *) pointer in which the newly created FlagShip array pointer will be stored.

<dim> is an (int) counter specifying the required array size. The valid range is 1 to 65535.

Returns:

<varPtr> is a (FSvar *) pointing to the newly created FlagShip array.

Description:

Similar to VAR_NEW, VAR_NEW_STATIC, the VAR_NEW_ARRAY() function creates and initializes a set of new variables resulting in a FlagShip array type (refer also to LNG.2.6.4). All the array elements have type NIL.

To create a multi-dimensional array, use the standard ARRAY() function. The same applies for a nested array. See example and section LNG.2.6.4.

Note that the FlagShip type "A" (array) is **not** equivalent to the C array of variables used for parameter passing and created by VAR_NEW_ARGS(). Note also the scope and visibility of the FlagShip pool variables, described in chapter 4.5.1.

Example 1:

Create and assign a single-dimensional and two-dimensional arrays, similar to the .prg statement DECLARE varSingle[5], varMulti [10, 4]:

```
FSvar *varSingle, *varMulti, *args[2];  
VAR_NEW_ARRAY (varSingle, 5);          /* create varSingle[5] */  
  
VAR_NEW_ARGS (args, 2);  
SET_VAR_INT (args[0], 10);  
SET_VAR_INT (args[1], 4);  
varMulti = UDF_EXEC(array)(2, args);   /* create varMulti [10, 4] */  
// varMulti = _bb_array (2, args);    /* ditto, altern. syntax */
```

Example 2:

Create and assign a nested array, similar to the .prg statement

```
varNested := {NIL, {{11, 12, 13, 14}, NIL}, 99}

LOCAL xx := NIL
#Cinline
#include "FSopenc.h"
{
    FSvar *varNested, *varPtr, *dummy;
    int ii;

    varNested = VAR_NEW_ARRAY (dummy, 3);          /* 1st dimens. */
    SET_VAR_COPY(VAR_ARRELEM (varNested, 2),
                VAR_NEW_ARRAY(dummy, 2));         /* second */
    SET_VAR_COPY(VAR_ARRELEM(VAR_ARRELEM(varNested, 2), 1),
                VAR_NEW_ARRAY(dummy, 4));         /* third */

    SET_VAR_NUM (VAR_ARRELEM (varNested, 3), (double) 99);
    varPtr = VAR_ARRELEM (varNested, 2);
    varPtr = VAR_ARRELEM (varPtr, 1);            /* varNested[2,1] */
    for (ii = 1; ii <= VAR_ISDIM(varPtr); ii++)
        SET_VAR_NUM(VAR_ARRELEM(varPtr, ii), (double)(ii+10));
    SET_VAR_COPY (VAR_NAME_LOCAL(xx), varNested);
}
#endCinline

? xx[1], xx[2, 1, 1], xx[2, 1, 2], xx[2, 2], xx[3]
// NIL, 11, 12, NIL, 99
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, VAR_DELETE, FUN.ARRAY(), VAR_NEW_ARGS(),
VAR_NEW_STATIC, SET_VAR_xxx(), IS_VAR_xx()

VAR_NEW_COPY ()

VAR_NEW_STATIC_COPY ()

Syntax:

```
varPtr = VAR_NEW_COPY (varPtr2);  
varPtr = VAR_NEW_STATIC_COPY (varPtr2);
```

Purpose:

Allocates memory for a FlagShip local or static variable and copies another variable into it.

Arguments:

<varPtr2> is a FlagShip variable which has already been initialized, the copy of which is assigned to the new variable pointer. Only (standard) variables of type C, N, D, L, M and NIL can be copied.

Returns:

<varPtr> is a (FSvar *) pointer to the new variable.

Description:

Similar to VAR_NEW, the VAR_NEW_COPY() function creates and initializes a new variable <varPtr> copying the contents of the variable <varPtr2> into it. To make a copy to an existing variable, use SET_VAR_COPY() instead.

VAR_NEW_STATIC_COPY() creates and initializes a new variable similarly to VAR_NEW_STATIC.

Note the scope and visibility of the FlagShip pool variables, described in chapter 4.5.1.

Example:

```
FSvar *myvar = 0;  
myvar = VAR_NEW_COPY (VAR_NAME_LOCAL(othervar));
```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

VAR_NEW, VAR_NEW_ARGS(), VAR_NEW_STATIC, VAR_DELETE,
SET_VAR_COPY(), VAR_NAME_xx(), SET_VAR_xx(), IS_VAR_xx()

VAR_NUM ()

VAR_INT ()

VAR_INTNUM()

VAR_FPNUM ()

Syntax:

```
valueN = VAR_NUM (varPtr);
valueN = VAR_FPNUM (varPtr);

valueI = VAR_INT (varPtr);
valueI = VAR_INTNUM (varPtr);
```

Purpose:

Accesses the floating point or long integer value stored in a FlagShip numeric variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already initialized as type N (numeric) or type I (integer numeric).

Returns:

<valueN> is a (double) value representing the FlagShip numeric variable.

<valueI> is a (long) value representing the FlagShip long integer variable.

Description:

VAR_NUM() allows an access to a FlagShip numeric variable of type N, which is stored as a double precision floating point (usually 64 bits wide).

VAR_INT() allows an access to an INTVAR FlagShip variable (typed I), which stores signed long integer, usually 32 bits wide in the range +/- 2147483647.

VAR_INTNUM() and **VAR_FPNUM()** are generalized access functions to FlagShip variable of any numeric type (N,I,F).

To set and change the stored value, you should use SET_VAR_NUM(), SET_VAR_INT() or SET_VAR_NUMDECI() functions, which performs validity checks.

Example:

General purpose function to convert an integer number to hexa string.

```
*** test.prg
? "123456 (decim) is " + num2hex (123456) + " (hex)"

FUNCTION num2hex (num) AS CHAR
LOCAL retval AS CHAR
#C i n l i n e
{
# i n c l u d e "FSopenc.h"
char buffer[10]; /* max 8 + sign + \0 byte */
```

```

if (IS_VAR_NUM(VAR_NAME_LOCPAR(num)))
    sprintf (buffer, "%lX",
            (long) VAR_INTNUM( VAR_NAME_LOCPAR(num)) );
else
    buffer[0] = 0;          /* wrong var type, "" is returned */
    SET_VAR_CHR (VAR_NAME_LOCAL(retval), buffer);
}
#endCline
RETURN retval

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_NUM(), SET_VAR_NUM(), SET_VAR_NUMDECI(), VAR_NEW,
VAR_NEW_ARGS(), SET_VAR_COPY()

VAR_OBJ ()

Syntax:

```
varPtr = VAR_OBJ (objPtr, elem);
```

Purpose:

Accesses an element of a FlagShip object type, created e.g. by the class function `_bb_getnew()`, `_bb_tbrowsenew()` etc.

Arguments:

<objPtr> is a (FSvar *) pointer to a FlagShip pool variable, already created and initialized as object.

<elem> is an (int) value representing the required element number in the object array. It is highly recommended to use only the definitions from FlagShip.h. These are prefixed with `GETOBJ_`, `ERROBJ_`, `TBROBJ_` and `COLOBJ_`, e.g. `GETOBJ_CARGO` or `TBROBJ_STABLE` etc.

Returns:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, representing the object element. This variable can contain any valid type, according to the class definitions in section OBJ.

Description:

FlagShip objects are similar to arrays, but object elements contains both normal data (user-modifiable) and internal class data, which is user-protected and cannot be modified.

VAR_OBJ() determines the pointer to the FlagShip object array element. To retrieve the stored value, use VAR_CHR(), VAR_NUM(), VAR_LOG() and VAR_DATE() functions.

Example:

```
*** test.prg
LOCAL myget := GETNEW(), value := SPACE(10)
myget:CARGO := "cargo text"
myget:ROW := 10
myget:COL := 5
myget:BLOCK := {|par| IF(par==NIL, value, value:=par)}

#C i n l i n e
{
# i n c l u d e "FSopenc.h"
FSvar *getObj, *varPtr = VAR_NEW, *args[3];
VAR_NEW_ARGS (args, 3);
if (! IS_VAR_OBJ (VAR_NAME_LOCAL(myget)) )
    FS_QUIT_COMMAND;

varPtr = VAR_OBJ (VAR_NAME_LOCAL(myget), GETOBJ_CARGO);
SET_VAR_CHR (args[0], "myget:CARGO =");
if (IS_VAR_CHR (varPtr))
    SET_VAR_COPY (args[1], varPtr);
el se
```

```

    SET_VAR_CHR (args[1], "cargo undefined");
    _bb_qout (2, args); /* print myget: CARGO */
    varPtr = VAR_OBJ (VAR_NAME_LOCAL(myget), GETOBJ_ROW);
    SET_VAR_CHR (args[0], "myget: ROW =");
    if (IS_VAR_NUM (varPtr))
        SET_VAR_COPY (args[1], varPtr);
    else {
        SET_VAR_CHR (args[1], "row undefined, set to 1");
        SET_VAR_NUM (varPtr, 1.0);
    }
    _bb_qout (2, args); /* print myget: ROW */
}
#endif

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_OBJ(), VAR_ARRELEM(), VAR_NEW, VAR_NEW_STATIC,
 VAR_NEW_ARGS()

VAR_SCR ()

Syntax:

```
winPtr = VAR_SCR (varPtr);
```

Purpose:

Accesses a FlagShip screen variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already created and initialized as type "screen".

Returns:

<winPtr> is a (WINDOW *) pointer to the curses WINDOW structure.

Description:

FlagShip uses the curses WINDOW structure to store the screen contents of a SAVE SCREEN command or the SAVESCREEN() function.

Example:

A general purpose function to display the current coordinates of the saved screen portion from a SAVESCREEN() function or a SAVE SCREEN command (in terminal i/o mode):

```
*** test.prg
LOCAL scrfull, scrpart := SAVESCREEN (5, 6, 20, 30)
SAVE SCREEN TO scrfull
display_coord (scrpart, "scrpart")
display_coord (scrfull, "scrfull")
FUNCTION display_coord (scrVar, varname)
*****
LOCAL_INT row_top, row_bott, col_top, col_bott
#Cinline
{
# include "FSopenc.h"
# define USE_STANDARD_CURSES
FSvar *varPtr = VAR_NEW;
WINDOW *winPtr;

varPtr = VAR_NAME_LOCPAR(scrvar);
winPtr = VAR_SCR (varPtr);

#ifdef USE_STANDARD_CURSES
/* OS independent version, uses curses functions */

getbegyx (winPtr, row_top, col_top);
getmaxyx (winPtr, row_bott, col_bott);
row_bott += row_top - 1;
col_bott += col_top - 1;
#else
/* curses dependent version, uses its structure */

row_top = row_bott = (int)(winPtr->begy);
```

```

    row_bott += (int)(winPtr->_maxy) -1;
    col_top   = col_bott = (int)(winPtr->_begx);
    col_bott += (int)(winPtr->_maxx) -1;
#endi f

    VAR_DELETE;          /* release contents of varPtr */
}
#endC i n l i n e

? "current screen coordinates of '" + varname + ;
  "' are: rows", LTRIM(STR(row_top)) + ;
  "... " + LTRIM(STR(row_bott)), ;
  "and col ums", LTRIM(STR(col_top)) + ;
  "... " + LTRIM(STR(col_bott))
RETURN

```

Include:

<FlagShip_dir>/include/FSopenc.h (#include's also curses.h)

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_SCR(), VAR_ARRELEM(), VAR_NEW, VAR_NEW_ARGS()

VAR_SPECIAL ()

Syntax:

```
ptr = VAR_SPECIAL (varPtr);
```

Purpose:

Accesses the pointer stored in the FlagShip SPECIAL variable.

Arguments:

<varPtr> is a (FSvar *) pointer to a FlagShip pool variable, already created and initialized via SET_VAR_SPECIAL() as "special" type.

Returns:

<ptr> is a pointer of any type, e.g. (int *) corresponding to the stored contents.

Description:

VAR_SPECIAL() allows access to a user defined memory, the pointer of which is stored in a FlagShip pool variable of the "special" type.

Example:

Manages an array of (long) values

```
*** test.prg
#Cinline
{
# include "FSopenc.h"
  UDF_DECL(assignspec)
  {
    long *myLongArr;
    int ii;
    if (argc < 1) /* param mandat. */
      return FALSE_VAR;
    if (IS_VAR_SPECIAL(argv[0]) &&
        (VAR_SPECIAL(argv[0])) )
      free (VAR_SPECIAL (argv[0])); /* avoid mem leak */
    myLongArr = malloc (10* sizeof(long)); /* 10x long */
    SET_VAR_SPECIAL (argv[0], myLongArr); /* assign */
    for (ii=0; ii < 10; ii++)
      *(myLongArr++) = 0L; /* clear to 0 */
    return TRUE_VAR;
  }

  UDF_DECL(savelongar)
  {
    long *myLongArr, oldVal;
    int index;
    if ((argc < 3) || (! IS_VAR_SPECIAL(argv[0])))
      return IZERO_VAR;
    myLongArr = (long *) VAR_SPECIAL (argv[0]);
    index = VAR_INT(argv[2]) -1; /* no check here! */
    oldVal = *(myLongArr + index);
    *(myLongArr + index) =
      (long)VAR_INT(argv[1]); /* store in array */
    return SET_VAR_INT(VAR_NEW, oldVal);
  }
}
```

```

    }
    UDF_DECL(releaseSpec)
    {
        long *myLongArr;
        if ((argc < 1) || (! IS_VAR_SPECIAL(argv[0])))
            return FALSE_VAR;
        myLongArr = (long *) VAR_SPECIAL(argv[0]);
        if (myLongArr)
            free(myLongArr);
        SET_VAR_NULL(argv[0]);
        return TRUE_VAR;
    }
}
#endCinline

FUNCTION start()
LOCAL mySpecial
assignSpec (@mySpecial)
? valueType(mySpecial) // '?'
? "LongVal [3] being:", saveLongArr(mySpecial, 1234, 3), ;
  "is now replaced by: 1234"
// ...
releaseSpec (@mySpecial) // must release by yourself
return NIL

```

Include:

<FlagShip_dir>/include/FSopenc.h

Compatibility:

Available in FlagShip only.

Related:

IS_VAR_SPECIAL(), SET_VAR_SPECIAL()

5. Inter-Process communication

Usually, two FlagShip applications communicate using shared databases. You may communicate very easily between two applications using the UNIX pipes.

Example for Inter Process Communication between a running C program (sender) and a running FlagShip Application (receiver).

This communication can be used to send "queries" to a FlagShip program running in background, for example. That background program may send its answer (using a second pipe or a normal file). In this way, a simple "database server" as a "query machine" can be implemented.

```
-----Sender as C file
/* file sender.c */
#include <stdio.h>
#include <fcntl.h>
main()
{
    int fd;
    if ((fd = open("/tmp/mypipe", O_RDWR)) <= 0) {
        printf("error\n");
        return(0);
    }
    write(fd, "message 1\n", 10);
    write(fd, "message 2\n", 10);
    close(fd);
}
```

```
-----Receiver as FlagShip Program
*** file receiver.prg
IF ((fd: = FOPEN("/tmp/mypipe")) <= 0)
    ? "error"
ENDIF
buffer = SPACE(1)
message = ""
prtfile = FS_SET ("print")
process = SUBSTR (prtfile, RAT(".", prtfile) +1)
WHILE INKEY() != 27
    x1 := FREAD(fd, @buffer, 1)
    IF SUBSTR(buffer, 1, 1) >= " " .AND. x1 > 0
        message += SUBSTR(buffer, 1, 1)
    ELSE
        IF LEN(message) > 0
            ? ">got message:", message, "in process", process
            message = ""
        ENDIF
    ENDIF
ENDIF
ENDDO
FCLOSE(fd)
```

```
-----Commands to test communication
FlagShip receiver.prg -oreceiver
cc sender.c -osender

# create a named pipe
mknod /tmp/mypipe P
chmod +rw /tmp/mypipe

# switch to terminal 1
receiver
#switch to terminal 2
sender

#switch to terminal 1, you should see:
>got message: message 1 in process 9876
>got message: message 2 in process 9876
```

Note the pipe declaration using the `mknod` command, otherwise a regular file will be used. When a two-way communication is required, you may use two or more pipes. For the synchronization of more than two processes, the pipe may be locked/checked using the `FLOCKF()` function.

Additional inter-process communication using shared memory or semaphores is also possible.

Index

#	
#Cinline	EXT-60
#endCinline	EXT-60
-	
_bb_prefix	EXT-78
_parc()	EXT-16
_parclen()	EXT-18
_parcsiz()	EXT-18
_pards()	EXT-20
_parinfa()	EXT-22
_parinfo()	EXT-24
_parl()	EXT-26
_parnd()	EXT-27
_parni()	EXT-28
_parnl()	EXT-28
_parscw()	EXT-29
_ret()	EXT-30
_retc()	EXT-31
_retclen()	EXT-31
_retds()	EXT-32
_retl()	EXT-33
_retnd()	EXT-34
_retni()	EXT-35
_retnl()	EXT-35
_retscw()	EXT-36
_storc()	EXT-40
_storclen()	EXT-40
_stords()	EXT-42
_storl()	EXT-43
_stornd()	EXT-44
_storni()	EXT-45
_stornl()	EXT-45
_xalloc()	EXT-46
_xfree()	EXT-47
_xgrab()	EXT-46

A	
ALENGTH()	EXT-48

API	
- C	EXT-see C API
Array	
- access	
-- element in C	EXT-116
- element	
-- type in C	EXT-134
- initialize in C	EXT-147

C	
C API	EXT-4
- #Cinline	EXT-60
- #endCinline	EXT-60
- access to FlagShip vars	EXT-10
- common mistakes	EXT-4
- example	EXT-8
-- calculate sinus	EXT-14
-- rotate string	EXT-9
-- screen manipulation	
--- characters	EXT-37
--- colors	EXT-36
- extend C	EXT-6
- external libraries	EXT-14
- FSextend.h	EXT-12
- FSopenc.h	EXT-73
- include file	
-- FSextend.h	EXT-12
-- FSopenc.h	EXT-73
- input/output	EXT-14
- inter-process communication	EXT-158
- naming convention	EXT-10
- open C	
-- example	
--- random numbers	EXT-61
--- replace characters	EXT-62
-- inline	EXT-60
--- extern libraries	EXT-61
-- main() function	EXT-64
- open C API	EXT-66
-- overview	EXT-72
- overview	EXT-6
- parameter checking	EXT-12

- source from compiler..... EXT-66
 - modifying EXT-71
- using open C EXT-14
- C interface..... EXT-4
- C++ API EXT-85
- Code block
 - access in C..... EXT-118
 - compile in C EXT-119
 - evaluate in C EXT-120
- Constant
 - predefined in C EXT-76

D

- Database
 - work area
 - access in C EXT-129

F

- Field
 - name in C..... EXT-137, 139
 - position in C EXT-129
- FlagShip
 - C interface..... EXT-4
 - compiler
 - C output..... EXT-66
- FSextend.h file EXT-12
- FSinit() EXT-57
- FSopenc.h file EXT-73
- FSreturn EXT-59
- FSudfname() EXT-58
- Function
 - declare in C EXT-75, 111
 - invoke in C EXT-75, 113
 - method
 - access in C..... EXT-76, 94
 - declare in C EXT-92
 - prototype in C EXT-115
 - use in C EXT-78

G

- Garbage collection EXT-76, 124, 126

I

- Input/output in C..... EXT-80
- Inter-process communication..... EXT-158
- IS_VAR_ARR() EXT-87
- IS_VAR_BLK() EXT-87
- IS_VAR_BYREF() EXT-87
- IS_VAR_CHR() EXT-87
- IS_VAR_DATE() EXT-87
- IS_VAR_EE() EXT-90
- IS_VAR_EMPTY() EXT-87
- IS_VAR_EQ() EXT-90
- IS_VAR_FALSE() EXT-87
- IS_VAR_FIELD() EXT-87
- IS_VAR_FP() EXT-87
- IS_VAR_GE() EXT-90
- IS_VAR_GT() EXT-90
- IS_VAR_INT() EXT-87
- IS_VAR_LE() EXT-90
- IS_VAR_LOG() EXT-87
- IS_VAR_LT() EXT-90
- IS_VAR_NE() EXT-90
- IS_VAR_NIL() EXT-87
- IS_VAR_NUM() EXT-87
- IS_VAR_OBJ() EXT-87
- IS_VAR_SCR() EXT-87
- IS_VAR_SPECIAL() EXT-87
- IS_VAR_STD() EXT-87
- IS_VAR_TRUE() EXT-87
- IS_VAR_TRUE_BLK() EXT-87
- ISARRAY() EXT-49
- ISBYREF() EXT-50
- ISCHAR() EXT-51
- ISDATE() EXT-52
- ISLOG() EXT-53
- ISMEMO() EXT-54
- ISNUM() EXT-55
- ISSCREEN() EXT-56

L

- Library
 - access in C..... EXT-14

M

- Macro
 - evaluate in C EXT-104

main() C function	EXT-64
Method	
- access in C.....	EXT-76
- declare in C.....	EXT-92

UDF_EXEC()	EXT-113
UDF_PROT()	EXT-115

V

O

OBJ_ACCEXEC()	EXT-94
OBJ_ASSEXEC()	EXT-94
OBJ_DECL_METH()	EXT-92
OBJ_DECL_METHACCESS()	EXT-92
OBJ_DECL_METHASSIGN()	EXT-92
OBJ_METHEXEC()	EXT-94

VAR_ARRELEM()	EXT-116
VAR_BLOCK()	EXT-118
VAR_BLOCK_COMPILE()	EXT-119
VAR_BLOCK_EVAL()	EXT-120
VAR_CHR()	EXT-121
VAR_DATE()	EXT-123
VAR_DEL_MARK()	EXT-126
VAR_DEL_ONEVAR()	EXT-124
VAR_DEL_SINCE_MARK()	EXT-126
VAR_DELETE	EXT-124
VAR_FPNUM()	EXT-150
VAR_INT()	EXT-150
VAR_INTNUM()	EXT-150
VAR_ISDECI()	EXT-127
VAR_ISDIM()	EXT-128
VAR_ISFLDPOS()	EXT-129
VAR_ISFLDWA()	EXT-129
VAR_ISLEN()	EXT-131
VAR_ISMODE()	EXT-132
VAR_ISTYPE()	EXT-134
VAR_LOG()	EXT-136
VAR_NAME_FIELD()	EXT-137
VAR_NAME_FIELD_BYNAME()	EXT-139
VAR_NAME_LOCAL()	EXT-141
VAR_NAME_LOCPAR()	EXT-141
VAR_NAME_MEMVAR()	EXT-137
VAR_NAME_MEMVAR_BYNAME()	EXT-139
VAR_NAME_STATIC()	EXT-143
VAR_NEW	EXT-145
VAR_NEW_ARRAY	EXT-147
VAR_NEW_COPY()	EXT-149
VAR_NEW_STATIC	EXT-145
VAR_NEW_STATIC_ARRAY	EXT-147
VAR_NEW_STATIC_COPY()	EXT-149
VAR_NUM()	EXT-150
VAR_OBJ()	EXT-152
VAR_SCR()	EXT-154
VAR_SPECIAL()	EXT-156

P

Parameter	
- access in C.....	EXT-141
- passing in C	EXT-78
Predefined constants in C.....	EXT-76
Print screen.....	EXT-82

S

SET_VAR_ADD()	EXT-97
SET_VAR_BLOCK()	EXT-98
SET_VAR_CHR()	EXT-99
SET_VAR_CHRLEN()	EXT-99
SET_VAR_COPY()	EXT-101
SET_VAR_DATE()	EXT-102
SET_VAR_INT()	EXT-106
SET_VAR_LOG()	EXT-103
SET_VAR_LOWER()	EXT-99
SET_VAR_MACRO()	EXT-104
SET_VAR_NIL()	EXT-105
SET_VAR_NUM()	EXT-106
SET_VAR_NUMDECI()	EXT-106
SET_VAR_SCR()	EXT-108
SET_VAR_SPECIAL()	EXT-109
SET_VAR_UPPER()	EXT-99
String	
- access in C.....	EXT-121
- length in C.....	EXT-131

Variable	
- access by name in C.....	EXT-73
- access content in C.....	EXT-75
- access in C.....	EXT-10

U

UDF_DECL()	EXT-111
-------------------	---------

- add in C EXT-97
 - array
 - element in C EXT-116
 - initialize in C EXT-147
 - size in C EXT-128
 - character
 - access in C EXT-121
 - assign in C EXT-99
 - length in C EXT-131
 - code block
 - assign in C EXT-98
 - compile in C EXT-119
 - evaluate in C EXT-120
 - code block in C EXT-118
 - compare in C EXT-74, 90
 - concatenate strings in C EXT-97
 - constants in C EXT-76
 - copy in C EXT-101
 - create in C EXT-73
 - date
 - access in C EXT-123
 - assign in C EXT-102
 - determine type in C EXT-74, 87
 - evaluated macro
 - assign in C EXT-104
 - initialize and copy in C EXT-149
 - initialize in C EXT-145
 - integer
 - access in C EXT-150
 - assign in C EXT-106
 - local
 - access in C EXT-141
 - logical
 - access in C EXT-136
 - assign in C EXT-103
 - mode of in C EXT-132
 - modify in C EXT-75
 - name
 - access in C EXT-137, 139
 - parameter access in C EXT-141
 - NIL
 - assign in C EXT-105
 - numeric
 - access in C EXT-150
 - assign in C EXT-106
 - decimals in C EXT-127
 - object
 - access in C EXT-152
 - objects
 - access in C EXT-76
 - pointer
 - assign in C EXT-109
 - screen
 - access in C EXT-154
 - assign in C EXT-108
 - special
 - access in C EXT-156
 - assign in C EXT-109
 - static
 - access in C EXT-143
 - type in C EXT-134
-

W

- Work area
 - access in C EXT-129

Notes



multisoft Datentechnik
Schönastr. 7
D-84036 Landshut

<http://www.fship.com>
sales@multisoft.de
support@flagship.de