FlagShip

Object Oriented Database Development System

Cross-Compatible to Unix, Linux and MS-Windows

MULTISOFT

Release 8.1

Section LNG

The whole FlagShip 8 manual consist of following sections:

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

multisoft Datentechnik, Germany

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

Section LNG

Manual release: 8.1

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

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

Copyright

Made in Germany. Printed in Germany.

Trademarks

Headquarter Address

multisoft Datentechnik
Schönaustr. 7
84036 Landshut
Germany

Phone: *(*+49) 0871-3300237

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

Web: http://www.fship.com

LNG: Introduction to FlagShip

1. Introduction to FlagShip

In this chapter, you will learn about the essential operating modes of FlagShip and the system differences between DOS and Unix/MS-Windows 32/64bit. All "Unix" descriptions apply also for 32/64bit Linux systems. The programmer's support is covered in detail in chapter 2.

> **Note:** This section gives a quick overview of the FlagShip language. It is not intended as an introduction to programming. If you are a beginning programmer, any good programming primer, especially in xBase, dBASE, Clipper or FoxBase, may be used first.

1.1 What FlagShip is

The FlagShip database development system consists of

- A powerful and flexible **programming language**, which is mostly compatible in syntax to Clipper and dBASE. Its commands and functions are mostly compatible to Clipper 5.x and Summer'87, whereby FlagShip offer **many additional features** and enhancements. A FoxBase and FoxPro compatibility is available by -fox compiler switch.
- The **FlagShip Preprocessor** acts as one pass of the Compiler. It performs syntactic source checking and translates the preprocessor directives (see section PRE) and user defined commands to UDFs (user- defined-functions). The preprocessor uses the standard include file "std.fh" by default (usually stored in <FlagShip_dir>/include), as well as other optional include files.
- The **FlagShip Compiler** which translates the preprocessed FlagShip (and also Clipper or xBase) source code into C source, checks it for syntax and plausibility, and then invokes the Unix (or Windows) C compiler to complete the translation into the machine language (also named native code).
- The accompanying **FlagShip Library**, containing
  - all necessary modules to support the language, dynamic variable scoping, statements and commands,
  - all standard functions linked as needed,
  - the runtime system to support macro evaluation at run-time,
  - the database and index engine,
  - the input/output system including graphic GUI interface,
  - the object classes,
  - a debugger enabling checking and setting actual variables and more.
- Tools for file and database handling, creation of Makefiles, as well as for porting your applications from or to the MS-DOS operating system.

1.2 Mode of operation

FlagShip is a true compiler. The language definition complies to great extent with Clipper definition, including many extras and enhancements. Thus, all conceptual differences between Clipper and dBASE apply for FlagShip, too. For a truly compatibility to FoxBase and FoxPro, use the -fpx or -fxp compiler switch or the additional package.

Most notable is the difference between a compiler (i.e. FlagShip) and an interpreter (i.e. dBASE). The main property of a compiler being that it produces executable files, not requiring any kind of a run-time module. The byproduct is protecting the source code.

The compilation of your application is done in the following phases:

- Pre-compiling the FlagShip language, syntax checking and resolving the preprocessor directives,
- Translation from the FlagShip language in the C code
- Compilation and optimizing the C source into machine language (object code) using the standard Unix C compiler or the used Windows C compiler (like BCC32 or MS-VC).
- Linking the object code together with the FlagShip and Unix or Windows libraries using the standard Unix or Windows linker.

All this steps are done automatically when invoking FlagShip, however each separate step can be individually executed with the proper commands. See more in section FSC.

The FlagShip compiler and library handles **three different i/o modes**:

GUI : graphical oriented i/o, requires X11 on Linux or MS-Windows (both 32 or 64-bit based)

Terminal: text/curses oriented i/o e.g. for console or remote terminals, same behavior as FlagShip 4.48 and Clipper.

Basic : basic/stream i/o e.g. for Web, CGI, background processing etc. The screen oriented i/o is roughly simulated for source compatibility purposes.

The i/o mode is either set at compile-time, or determined at run-time from the currently used environment. The compile-time solution is recommended when the target environment is known, it produces faster and smaller executables.

When the application is compiled with -io=a (or without -io=? switch), a **hybrid application** is created where the current environment is determined at run-time. GUI i/o is used on active X11/Win32/64, Terminal on console, Basic otherwise. The detected environment can be overridden by the command-line switch -io=g/t/b of the executable.

The GUI based executable creates automatically main window with menu bar. The setup for the window and menu is customizable and available in the *<FlagShip_dir>/system/initio.prg* and *initiomenu.prg*. Note: these functions are invoked at start-up, before user INIT FUNCTION and user-main is executed.

All the screen oriented i/o such as @..SAY, @..GET, Achoice(), Alert(), Save/RestScreen() etc. are fully supported also in GUI. Some of them are internally mapped to GUI conformable widgets (i.e. controls in Microsoft terminology) like ListBox, dialog boxes etc. or window properties like MenuBar, see more in LNG.5.3.

In GUI, both the common row/column coordinates and y/x pixel entries are supported and controlled by the SET PIXEL on/OFF or by passing an additional parameter for the most standard functions and objects. The default is SET PIXEL OFF which ensure backward source compatibility to FS4 and Clipper sources. In this mode, FlagShip calculates internally the real pixel position corresponding to the used font, see details in description of Applic and Font classes and SET FONT TO... command.

There are in fact three different classes in the FlagShip library for each specific i/o operation. The decision which class should be taken is done either by the compiler when the -io=g/t/b switch was used, or at run-time from the system environment or via command-line switch. The run-time setup is available in the source .../system/initio.prg

The in GUI mode used **event oriented programming** requires mostly full OOP programming style, which differs significantly from the idea of procedural oriented xBase programs. To avoid bothering you with a new programming style and a long learning curve, FlagShip handles all the required event actions internally, so you can fully concentrate your power on your main task - the development of professional applications.

For programming hints covering the difference between GUI, Terminal and Basic i/o mode, please refer to section LNG.5.3.

1.3 System differences

FlagShip was designed for the Unix and Windows 32/64bit operating system. But because Unix differs from one computer to the other, the actual FlagShip package you are using was adapted especially for your computer and operating system. The 32/64bit MS-Windows differs significantly to Unix, so you will need special MS-Windows port of FlagShip, to handle **the same** sources and data.

> Note: Therefore, we guarantee the full functionality on the same operating system with a release number which is the same as or higher than that given on the FlagShip distribution media (see label and Activation Card) only.
>
> We cannot make any warranty for attempting to run an executable on different OS. The produced executable may however run on the same processor type with other operating systems (e.g. XENIX executable may run on SCO Unix but seldom the reverse and never on Motorola) or other, higher OS releases (i.e. SCO Unix 3.2 runs on SCO rel 3.4 and 4.1 or HP/UX 8.07 on HP/UX 9.2 etc.).

FlagShip takes into consideration, and adapts to the differences between MS-DOS and various flavors of Unix (or to 32/64bit MS-Windows) for you, so that you do not have to bother with this. The main differences to MS-DOS operating system are:

- Composition and length of filenames: all versions of Unix and Windows support long file names (usually up to 255). Unix makes difference between lower and upper characters, the dot (.) is treated as any other character, i.e. it has no predefined meaning and can be repeated in the filename as many times as chosen. In Unix the filenames are usually given in lower case characters, directory and paths are written with "/" instead of "\". File names given in UTF8 and Unicode are not supported. FlagShip translates most of these differences automatically, see section LNG.3 and SYS.
- RAM size and overlays: Neither Unix nor Windows-32/64 has concept corresponding to DOS overlays. Instead, it uses paging and/or swapping to achieve virtually unlimited memory. Therefore, linking is straight- forward, without overlays. See more in chapter LNG.9.3 and SYS.
- Disk space and file system: Unix utilizes the concept of mounting physically separate disks into one file system (a tree-like structure of directories). Thus, from the user's point of view, all disks and partitions look like one. Asking for free disk space returns the value for the physical disk and partition on which the current directory resides. There is no equivalent to the MS-DOS drive selector (like C:) in the path specification, but FlagShip can translate these automatically to a Unix directory using the environment variable x_FSDRIVE (see LNG.3.2, LNG.9.3-4 and FSC.3.3). Windows 32/64 handles files similar to MS-DOS, it ignores lower/uppercase and supports large file names.
- Compatibility of programs: FlagShip supports the structure and semantics of Clipper or dBASE .prg files as closely as possible, no semantic or syntactic program changes are necessary. Since .fmt files are essentially program files they are fully supported, too. The syntax of commands, standard functions and preprocessor directives is compatible with Clipper'87 and 5.x. Also the Extended C system of Clipper is supported. In addition, the FlagShip programmer may also use inline C code within a .prg file or the Open C API interface. See chapter LNG.8 and the section EXT.
- Compatibility to FoxPro sources: the most differences are supported automatically by using the -fox compiler switch. See further details in section APP.
- Compatibility of database files: .dbf and .dbt file structure of dBASE III+, Fox or Clipper is fully supported by the default DBDIDX driver without any transformations. Dbase IV and V databases have to be converted (by dBase) to dBaseIII format. FoxBase and FoxPro .fpt files are supported as well, see FUN.DbCreate().

  Index files, neither .ntx (Clipper) nor .ndx (dBase), .mdx (dBase), .idx (Foxbase), .cdx (Clipper, FoxPro) are supported by the default DBFIDX, but available in the CB4* driver, see sect. REL. FlagShip's .idx structure is not compatible to the same named Foxbase index file.

  The memo files .mem are compatible to all Xbase dialects. The report an label files .frm and .lbl are also supported. See also chapter LNG.3 and LNG.9.3-5.

2. Basis of the FlagShip Language

The FlagShip language is based on the Xbase standard with all the extensions of CA/Clipper. Therefore, all Clipper'87 or 5.x programs and most other Xbase dialects (dBASE, FoxBase etc.) will be compiled by FlagShip without any modification. Only some system specific differences (see LNG.1 and LNG.9) have to be considered.

2.1 Language Specification

- Language, Compiler:

Fully source compatible	Clipper 5.0 ... 5.2 + 5.3 & 87
Almost source compatible	dBASE, FoxBase
Additional compatibility package available for	FoxPro
Common source code for DOS and Unix and Windows	yes
User defined commands and functions	yes
Preprocessor directives of Clipper 5.x	yes
Additional preprocessor directives	yes *
Macro evaluation at run-time	yes
Code blocks, objects	yes
All Clipper commands and functions	yes
Additional commands and functions	yes *
Abbreviation of commands and some functions	yes
Extended C system	yes
Inline C code within the .prg file	yes *
Translation of .prg source into the C language	yes *
Support GUI look & feel environment	yes *
Support of terminal/console based applications	yes *
Support of Web/CGI or batch processing	yes *
Hybrid application for all three modes	yes *

Function or procedure name	128 chars, first 10 chars signif.
Number of UDFs and procedures	unlimited *
Nesting of UDFs	unlimited *
Structure nesting	unlimited *
Number of nested loops	unlimited *
Expression and macro length	up to 4 KB *
Nesting of each macro	up to 255 depth *
Break from loops	yes *
Save & restore screen	yes
PC-8/OEM character set support	yes
ANSI/ISO character set support	yes *
Binary 0 in string	yes, on request *

- Files, Multiuser:

Maximal size of .dbf, .dbt, .dbv, .idx	each up to 16 Terabyte **
Fully compatible .DBF, .DBT, .FPT files	yes
Compatible .MEM files	yes
File & record locking, exclusive & shared dbf	yes
Multiple record locking	yes *
Automatic locking, user-modifiable	yes *
Network database share	yes

Multiuser, multitasking	yes *
Support for large files > 2GB	yes *
Maximal program size (executable)	up to 4 Gigabyte **
Filename Length	any size, system dependent **
Path name Length	255 characters *
Simultaneously opened files	unlimited **

- Input/Oputput:

Color and b/w terminals	yes
GUI look & feel	yes *
Supports terminfo, curses	yes *
Runs on text & X/terminal	yes *
Access to Unix and Windows using RUN	yes *
PC-8 or ANSI/ISO character set	yes *
Additional character mapping	yes *
Printer output	spooled *
Low-level input/output	yes

- Variables:

Dynamic variable scoping	yes
Public, private, arrays, static, local	yes
Typed static, local, global	yes *
Number of memory variables	unlimited *
Length of variable name	unlimited, first 10 chars are significant
Variable structure	28 bytes, incl.name and value **
Character variables	8-bit ASCII, up to 2000 MB, on heap *
Numeric variables	8 byte double float, prec. 15 digits
Date variables	4 byte long integer
Logical variables	1 byte character
Screen variables	4 byte memory pointer *
Arrays	one- and multi-dimensional
Array size	65535 each dimension *
Array element type	all variable types or other array
Store/restore of arrays in .mem files	yes *

- Objects:

Predefined classes	Get, Error, Tbrowse, TbColumn
Database classes	DataServer, DbServer, DbfIdx *
User definable classes	yes *
Inheritance of classes	yes *
Method, Access, Assign	yes *
Protected and Static methods	
Instance, protect, hidden, export	yes *
Prototyping of methods	yes *
Automatic prototypes by the FS compiler	yes *

- Database and Index:

Simultaneously opened working areas	65534 **
Simultaneously opened files	unlimited **
Indices per working area	65000 *
Relations per working area	65000 *
Records per database	4 billion
Database size	up to 16 Terabyte **
Fields per record	65000 *
Character field	8-bit ASCII, up to 64 KB *

```
Numeric field            ASCII format up to 19 digits incl.sign and deci
Date field                                        ASCII 8 digits always
Logical field                      One ASCII character ('T','F','Y','N')
Memo field (.dbf)                        10 digits pointer to .dbt block
Memo field (.dbt)                   blocks of 512 bytes, up to 32MB each
Compressed memo field (.dbv)               variable size, up to 2GB each *
Structure of DBF file                 Binary header + sequential records
                                         of fixed length in ASCII format
Structure of .IDX file             Binary header + Btree binary structure *
                                not compatible with .NTX, .NDX, .CDX files
Index key size (statement)                                     420 bytes
Index key size (evaluated)                                     238 bytes *

By OOP capsulated, replaceable database drivers                      yes *
Multiple record locking                                              yes *
Automatic index integrity check                                      yes *
Automatic locking, user-modifiable                                   yes *

- System:
Supported Unix systems                  32/64-bit, more than 50 different *
Supported MS-Windows systems 32/64-bit (NT4,2000,2003,2008,XP,Vista,7,8) *
User requirements            Unix run-time system or 32/64bit MS-Windows *
Developer requirements                FlagShip devel.system + C compiler *
Run-time royalties                                                  none
Produces stand alone executables                                     yes

Notes: *  marked items are FlagShip extensions to Clipper.
       ** marked items dependent on the Unix implementation or the
          actual kernel setting or the used Windows32/64 version.
```

2.2 Structure of a FlagShip Program

The source code of any FlagShip program consists of a standard ASCII file with the extension (the last four chars of the file name) being .prg or .fmt, which contains of an arbitrary number of program statements.

2.2.1 Language Syntax

There are a number of rules that have to be obeyed so that the compiler can understand the programmers intention and to produce valid executable code.

A program line may contain one or more statements, or a continuation of the statement given on the previous line, such as:

- executable program statements (expression, assignment)
- control structures (if, case, loops, breaks etc.)
- variable or field declaration
- module (procedure or function) declarations
- standard and user defined commands
- access to standard and user defined functions
- preprocessor directives
- C inline code
- full-line or inline comments

2.2.2 Statements

Any program statement is terminated by the end-of-line marker. The program line can be continued on the next line if the last character on the first line is a semicolon (";"). In this way a program line can be continued on any number of lines, providing that the total length does not exceed the current limit of 4095 characters.

FlagShip accepts both the Unix end-of-line syntax (line ends with LF only) and the MS-DOS and Windows syntax (line ends with CR+LF).

The formatting of the source code (e.g. indenting of control structures or adding spaces between the expression operators) by use of white space (blank or tab) increases the readability of the source code, and will be ignored by the compiler.

Case sensitivity: all elements of the FlagShip language, like commands, keywords, variables, procedure or function names etc. (except within the extend C system and the Unix file or path names) are **not** case sensitive (except the #define keywords for the preprocessor). They may

be given upper or lower case and intermixed. For clarity of used syntax, the keywords, commands and standard function will be always given in UPPERCASE in this manual.

Multiple statements: a program line may consist of one, two or more statements (commands, expressions), separated by a semicolon ";". The last valid character on the line may not be a semicolon, since this marks the continuation of the statement in the next line. For example

```
a := 1
b := 2 ; c := a + b ; d := str(c)       // same as three prg lines
text := "this " + ;                     // statement continues
        "is my text"
```

Lists: some commands or statements require or allow more than one value, argument etc. Such multiple items are separated by comma (see also LNG.2.8), e.g.:

command list :	LOCAL var1, var2 := 0, var3
argument list :	USE adress INDEX index1, index2, index3
parameter list :	(name, text, param3)
expression list:	(var1 := 1, var2 := 2, var3++, 4)
code block body:	{|x| var1 := x, var2 += var1, .T.}

Other syntax elements are explained in detail in following chapters.

2.2.3 Comments

The clarity and readability of the source code can be significantly increased by using comments to specify the program author, source status and/or to describe the module functionality, parameters required and statement results.

The comments do not increase the object or executable code size, since they are removed from the produced end code during the compilation process either by the FlagShip or the C compiler.

FlagShip supports four kinds of comments:

- Comment lines are marked with a star (*) or the NOTE keyword, the rest of the line may contain any user comment. Continuation of comment lines using the semicolon ";" is not allowed.
- Inline comments start with double ampersands && or slashes //, the rest of the line will be ignored by the compiler.
- Special comments are enclosed in /* and */ (slash-star and star-slash). These comments are accepted also within a command, statement or expression and may be continued through several lines. Nesting of these special comments is not allowed.
- Empty lines are treated by the compiler same as comment lines. They may be used to increase the source readability.

2.3 Program Files and Modules

Any source file may consist of one or more independent program modules. These program modules are called user defined procedures (UDP) or user defined functions (UDF).

Each module begins with the name declaration **PROCEDURE** <name> or **FUNCTION** <name> and ends on reaching the next UDF or UDP declaration or the end of the file. Any module returns execution control to the calling module when encountering a RETURN statement or the module end. The <name> of the UDP or UDF is any sequence of characters (A...Z), digits (0..9) and underscore (_) signs, where the first character is either alpha or underscore. Upper/ lower case makes no difference, only the first 10 chars are significant.

The first module declaration of each file may be omitted (except the -na compiler switch is used). In such a case, the FlagShip compiler will declare an "**automatic procedure**" with the same name as the source file, but without the .prg extension (see PROCEDURE in section CMD).

The whole application may consist of an arbitrary number of source files and modules. The FlagShip compiler will link all the required files and modules together with the (used) standard functions from the FlagShip and system libraries into one executable file.

2.3.1 Main Program

Each application has one main module and optionally a number of subordinate program modules. The execution of the application always starts at the beginning of the main module.

- If the declaration of the main module is omitted (auto-declaration) i.e. compiled w/o the - M<mainmodule>, the execution starts with the first executable statement of the file. The compiler switch -na may not be used in such a case.
- The main program may begin with the usual PROCEDURE or FUNCTION declaration (see 2.3.2) of any valid name. In this case, compile this source file with the -na option and specify the name of the main module by the -M<mainmodule> switch for the compiler (see section FSC and CMD).

In very small applications, the whole source code consists of the main program only. More often, the application is structured in several modules (using user defined procedures and functions), and the main program controls the activation of the required module.

Argument passing from Unix shell or Windows CMD: because the main program is controlled by FlagShip the same way as any other UDF or UDP, the main module may receive the actual arguments with which it is called using the standard format for formal parameters (PARAMETERS statement or within brackets enclosed formal parameters). All formal parameters, if any, are of type character. The argument separator for the Unix shell or Windows CMD is at least one white space (blank, tab):

```
*** file mymain.prg ***
**  PROCEDURE mymain                    // auto declared by FlagShip
PARAMETERS cmd1, cmd2, cmd3             // receives arguments

**  PROCEDURE mymain (cmd1, cmd2, cmd3) // alternative syntax
**  FUNCTION  mymain (cmd1, cmd2, cmd3) // alternative syntax

? "params received:", cmd1, cmd2, cmd3  // output to screen
? "the name of the executable is " + EXECNAME()
RETURN                                  // end of mymain
```

Invoke

```
$ FlagShip mymain.prg -o mymain
$ ./mymain                              // output: NIL NIL NIL
$ mymain test                           // output: test NIL NIL
$ mymain 1 test a b c                   // output: 1 test a
$ mymain 1 test "a b c"                 // output: 1 test a b c
```

2.3.2 Procedures and Functions

Each program module may call any number other user defined modules (UDP, UDF, extended C function) or any standard functions from the FlagShip library.

Procedures (UDP) and functions (UDF) are used to encapsulate computational blocks of code to provide readability and modularity, to isolate change, and to help manage complexity. Very similar to UDF are also methods of a class, see LNG.2.11.

The called module is activated by

- UDP: issuing the command "DO <name> [WITH <arguments>]"
- UDF: giving the UDF name followed by left and right brackets, "<name> ([<arguments>])".

A function may be used in an expression. For example:

```
DO menu                                 // call UDP "menu"
choice = input ()                       // call UDF "input"
DO output WITH 1, choice, .T.           // call UDP "output"
print (1, choice, .T.)                  // call UDF "print"
actDate = date ()                       // call std.function
```

Reaching the above "activation" statement, the actual module passes the control to the called UDF, UDP or function, passing down also the optional arguments. The called module is executed until the RETURN statement is reached (or another subsequent module is called). The end of the called module passes the control back to the calling program module.

The main difference between a UDP and a UDF is that a standard function or UDF returns a value into the calling program (which may be ignored), whereas a UDP procedure has no return value. Also, parameters are passed differently to UDFs and UDPs (by value for UDF or by

reference for UDP); for more details see sections CMD and FUN (DO, PROCEDURE, FUNCTION).

> Note: in FlagShip, the calling conventions of UDF and UDP are interchangeable, so an UDP may also be called as UDF and an UDF may be called using the DO...WITH command.

Using **aliases**: the call of standard and user defined functions (UDF) as well as procedures called by the UDF syntax can be combined with a specific database working area, in the same manner as the usage of field variables; the whole UDF has to be enclosed in parentheses. Prior to the execution of that function, the specified area is selected and stays as "actual working area" during the function execution. Returning to the called program, the previous working area is restored.

```
USE address ALIAS adr  NEW
USE custom ALIAS cust NEW
? UPPER(name + city) adr->(UPPER(name + city))        //    (1)
? UPPER(name + city), UPPER(adr->name + adr->city)    //    (2)
? UPPER(name + city)                                  // ┌─
SELECT adr                                            //  | (3)
?? UPPER(name + city)                                 //  |
SELECT cust                                           // ─┘
```

The three fragments of code labeled (1), (2) and (3) are three different ways of expressing the same thing, and produce the same result.

Argument passing: The calling program may pass any number of arguments to a UDF or UDP. Each argument is separated by a comma and can be a database field, memory variable, constant, expression, or nothing. If you do not wish to specify an argument you just put a comma, as in the following example:

```
usr_fun2 (2+3, x, , "123")
usr_fun2 (2+3, x, NIL, "123")
DO usr_proc1 WITH 5, 10, NIL, "text"
```

The third argument is here omitted, which is equivalent to using the reserved variable NIL.

The called module (UDF, UDP, C or standard function) receives the arguments into predefined variables, called formal parameters in the same order as the arguments passed (see more CMD.PARAMETERS). The main advantage of parameter passing is the standardization of program code, running with different initial values, e.g. using a simple function:

```
value  = my_add (1,  2, 3)           // "value" variable becomes 5
value := my_add (2, -1, 0)           // "value" variable becomes -2
value := my_add (2, value, 1)        // "value" variable becomes -3
RETURN

FUNCTION my_add (par1, par2, oper3)
RETURN par1 * par2 + oper3
```

Call by reference: The arguments of the "DO...WITH" call are passed by reference. This means, the formal parameter receives the address of the actual argument. Changes to the parameter within the called UDP (or such called UDF) automatically affect the argument; unchanged remain only constants, expressions and arguments of database fields. Closing the argument in brackets, passes it "by value" instead. For further details see CMD.DO, CMD.PARAMETERS.

Call by value: the arguments of a function call are passed "by value" into the called module. That means, the calling program copies the actual argument values and passes the duplicates. So changes of the parameters by the UDF affect only the duplicates and will never change the passed argument. If an argument should be passed by reference, the @ sign must be placed prior to the argument. Array names (but not array elements) and objects are always passed by reference, regardless the @ argument prefix. For further details see CMD.FUNCTION, CMD.PARAMETERS, CMD.PROTOTYPE.

Argument checking: there is no default argument/parameter type checking in the xBASE language, as in other languages. The receiving parameter can be checked within the UDF or UDP at run-time using the TYPE() or VALTYPE() function (see section FUN). But FlagShip supports typing of parameters and the UDF return value, which allows both compile-time and run-time argument checking, see 2.6.6 and CMD.PROTOTYPE. FlagShip also supports the passing of a variable number of parameters, so the number of the arguments passed may differ from the number of defined formal parameters. The number of actual parameters passed may be determined by the PCOUNT() function or using the type checking functions, e.g.:

```
FUNCTION my_add (par1, par2, par3)
IF PCOUNT() < 2                        // at least 2 param needed
   RETURN 0
ELSEIF VALTYPE(par1) <> "N" .OR. ;     // first two params not numbers
       VALTYPE(par2) != "N"
   RETURN 0
ELSEIF VALTYPE(par3) # "N"             // 3th param not number,
   RETURN par1 * par2                  // accept first two
ENDIF
RETURN par1 * par2 + oper3             // calculate all params
```

Recursion: the function or procedure call may be recursive to any depth. Recursion means, the UDF or UDP call directly or indirectly themselves. The programmer has to specify the end of the recursion to avoid "infinite" recursion (which in practice may seldom occur because the Unix system stops the program execution, if the stack and swap space where the local variables are stored is exhausted).

STATIC procedures and functions: These are UDPs or UDFs which are visible and accessible by modules within the same .prg file only. These modules are invisible for all other program files. Their names do not conflict with STATIC UDPs or UDFs having the same name within other program files. They are also not reachable through macros. For further details see CMD.FUNCTION, CMD.PROCEDURE.

INIT and **EXIT** procedures and functions: These are UDPs or UDFs which are automatically executed at the program begin (before invoking the main module) or at the program end

(before returning to the Unix/Windows shell). For further details see CMD.FUNCTION, CMD.PROCEDURE.

Calling Unix/Windows shell or program: a special case of performing other program tasks is the call of any other executable, script or Unix/Windows shell using the command RUN (or !, see section CMD). This is similar to invoking the executable directly from the Unix shell (or CMD-Window) or from a script or calling the system() function in C. The RUN call may include all the required arguments. When the so called executable is finished, the return code (error level) can be checked by the standard function DOSERROR(). FlagShip allows also the activation of the executable (or script) in background. For further details see CMD.RUN.

SET KEY and **ON KEY** commands specifies to call a defined user-procedure whenever the predefined key is depressed. This is often used in user- friendly-programs to execute a context specific help or to support the user with additional information. When starting a program, FlagShip automatically assigns the F1 key to the user-defined-procedure HELP, if one exists. For further details see CMD.SET KEY and ON KEY.

Pre-**validating** and post-validating functions can be used for conditional data entry or to validate the actual data entered during full screen input (see LNG.5.2, (CMD) @..GET and the CMD.READ command).

Prototyping of the UDF or UDP parameters or of the return value allows both the compile-time and run-time parameter/ assignment check and optimizes the calling sequence. The standard FlagShip functions are prototyped in the "stdfunct.fh" file. See more in LNG.2.6.6 and CMD.PROTOTYPE.

2.3.3 Code Blocks

Code blocks are special unnamed inline functions, in syntax similar to

```
[var :=] no_name_function ([parameters])
RETURN (expList)
```

Code blocks bear some similarity to UDFs and macros. They can be compiled or evaluated, may be used in compiled macros or stored in variables. Note: the compiled code blocks are significantly faster than macros or macro- compiled/-evaluated code blocks. The syntax of a code block is:

```
{ |<paramList>| <expList> }
```

where

<paramList> is an optional list of variables to receive parameters passed to a code block from an invocation of the EVAL() function, very similar to the formal parameters of an UDF. The parameter list is comma separated and must be enclosed within vertical bars ("|", chr(124), 7Chex). Variables specified in this list are declared local to the code block and are visible only within the code block definition. The code block has to contain these two vertical bars, whether the parameters are used or not.

<expList> is a list of expressions of any type. Two or more expressions must be separated by comma, see LNG.2.8. Commands, control structures or declarators are not allowed within the expression. The code block returns the last given expression.

The whole code block is enclosed in curly braces "{" and "}". Since the code block is usually translated during the compilation, its usage is significantly faster than the usage of macros. Therefore, code blocks are often used by the FlagShip preprocessor to translate standard commands, using the std.fh file.

Visibility of variables: during the execution of a code block, all LOCAL or STATIC variables of the UDF in which the block is declared, may be used. LOCAL variables of the UDF where the code block is executed, if this UDF is not the same UDF as where the code block was declared are invisible. In contrast, the usage of PRIVATE or PUBLIC variables is restricted to their visibility during code block execution, not the declaration. The variables declared in the <paramList> are visible for the code block only.

Possible operations with code blocks are:

- assignment to a variable using "=" or ":=",
- argument passing for an UDF,
- execution using EVAL(),
- multiple evaluation on array or database using AEVAL() or DBEVAL().

Examples:

```
LOCAL vartext := "other text"
LOCAL v1 := 1, v2 := 2, v3 := 3
blockVar1 := { || "text" }
blockVar2 := { || "text", vartext }
blockVar3 := { |par| QOUT(SUBSTR(par,1,3)) }

? EVAL(blockVar1)                          // text
? EVAL(blockVar2)                          // other text
EVAL(blockVar3, vartext)                   // oth
x = EVAL({|| v1++, tmp:= v1+v2, tmp+v3 }) // x = 7
DBEVAL ( {|| FIELD->fld := "xxx"} )        // REPLA ALL fld WITH "xxx"
```

Standard vs. compiled **macros** (see LNG.2.10) produce different results when used in the code block body. Standard macros get expanded at code block definition while compiled ones during the execution of the code block:

```
PRIVATE macvar := "value1"
PRIVATE value1 := "text", value2 := "other text"
bVar1 := {|| &macvar }                     // same as {|| "text" }
bVar2 := {|| &(macvar) }                   // compiled at run time

? EVAL (bVar1)                             // output: text
? EVAL (bVar2)                             // output: text
macvar := "value2"
? EVAL (bVar1)                             // output: text
? EVAL (bVar2)                             // output: other text
```

Macro-evaluated Code Block:

Code blocks may be stored in string variables (and as such also stored in database or memo fields) and evaluated using the macro operator "&", see also LNG.2.10. Example:

```
blockVar1 := "{ || 'text' }"
blockVar2 := "{ |par, tmp| tmp := SUBSTR(par,1,6), QOUT(tmp) }"

REPLACE stor_var WITH blockVar2          // store code block
x := EVAL (&(blockVar1))                 // x =  "text"
EVAL (&(stor_var), "my text...")         // out: "my tex"
bl := &(stor_var)
EVAL (bl, "other text")                  // out: "other "

bl := &(blockVar1); ? VALTYPE(bl)        // "B"
? EVAL(bl)                               // "text"
```

Note the differences between the "compiled" and "macro-evaluated" code blocks. The former is compiled by the compiler to native code, while the latter is evaluated by the FlagShip run-time system. Compiled code blocks are therefore significantly faster than the evaluated ones.

2.4 Commands

Commands are an essential part of any Xbase language (see section CMD). A command performs a specific action, similar to a call of a standard function. Commands are formed from a

- Command verb which is the command identifier and may be a keyword (like USE, READ) or a special character (like ? or @).
- The identifier may optionally be followed by one or more command clause keyword(s), like NEW, SAY, GET, PICTURE etc., which define specific actions of the command. White space separates the identifier, keywords and arguments.
- The command or keyword may optionally have argument(s) to set up the command execution with a defined value. Two or more arguments are separated by commas.

Examples of valid commands (for clarity, commands and keywords are here given in uppercase, arguments in lowercase):

```
USE                                 // identifier only
USE address                         // command parameter
USE address NEW                     // additional keyword
USE address NEW INDEX adr1, adr2    // 2. keyword with arguments
@ x,y SAY text PICTURE pict         // other command w. keywords
REPLACE name WITH "Smith", ;        // command broken
        zip  WITH 12345, ;          // into several
        joindate WITH DATE()        // program lines
```

All standard commands and keywords may be abbreviated to the four first characters. In FlagShip, most of the commands are translated to an equivalent function of the standard library. The definition of this translation is given in the include file "std.fh". This translation is also noted in the (CMD) reference.

The programmer can define his own commands and translate them into a UDF, UDP, standard function or other valid expression using the preprocessor directives #command, #xcommand, #translate or #xtranslate, see section PRE.

2.5 Control Structures

Control structures determine the logic flow of a program - the way program control moves from one part of a program to the other as the program runs. The control structures establish the order in which statements execute, the conditions under which they execute, and how often they execute.

Control structures are:

- Sequential processing: function or procedure call, command execution, RUN (see LNG.2.3 and 2.4)
- Choice constructs: IF, DO CASE (see LNG.2.5.1)
- Iteration constructs: FOR, DO WHILE (see LNG.2.5.2)
- Program interrupt constructs: BEGIN SEQUENCE, BREAK (see LNG.2.5.3)

In FlagShip, all the control structures may be nested and combined to any depth.

2.5.1 Choice constructs

If the program should execute different tasks or statements under different circumstances or conditions, one or a combination of the following choice constructs may be used:

IF...ENDIF: a program part enclosed should be executed only if the specified condition is met (see also CMD.IF).

```
PARAMETERS cmd1                          // get command-line argument
IF .not. EMPTY(cmd1)                     // is argument given ?
   ? cmd1                                // yes, print it
ENDIF
```

IF...ELSE...ENDIF: program choices to be executed if the condition is met or to be executed otherwise. The example demonstrates also nested IF constructs.

```
IF DAY(DATE()) <= 10                     // check the condition
   ? "first month period"                // print if condition is true
ELSE                                     // condition is not met
   IF DAY(DATE()) <= 20                  // check other condition
      ? "second month period"            // print if true
   ELSE
      ? "third month period"             // condition not met
   ENDIF                                 // end of second condition
ENDIF                                    // end of first condition
```

IF...ELSEIF...ELSE...ENDIF: this is an extended IF..ELSE..ENDIF construct, which is identical to nested IF..ENDIF structures or to the DO CASE...ENDCASE construct. The number of ELSEIFs is unlimited, the ELSE part may be left out.

```
choice = mymenu()                       // get required program flow
IF choice = 1
   DO my_procedure WITH "text"          // execute UDP
```

```
ELSEIF choice = 2
   my_funct ()                          // execute UDF
ELSEIF choice = 3
   my_funct3 (choice)                   // execute other UDF
ELSE
   ? "invalid choice"                   // print error msg
   QUIT                                 // and abort program
ENDIF
```

DO CASE...CASE...ENDCASE are typically used to execute multiple choices. This construct is identical to the alternate syntax IF..ELSEIF..ENDIF. There is no restriction on the number of CASE elements, but at least one must be given. The OTHERWISE clause is optional. Note: as opposed to similar constructs in other programming languages (like switch() in C), the CASE conditions are processed sequentially and always calculated at run- time. Each CASE may contain different condition checks (see (CMD) DO CASE).

```
actDate = DATE()                        // get system date
DO CASE                                 // perform specific action
CASE DOW(actDate) = 2                   // choice 1: day-of-week
   ? "today is monday...sleep well"
CASE DOW(actDate) = 1 .OR. DOW(actDate) > 6
   ? "non-programmers have a weekend today"
CASE DAY(actDate) = 1 .AND. MONTH(actDate) = 1
   ? "happy new year !"
CASE size_of_my_shoe < 6                // other choice
   ? "you're a child, please call your parents..."
   QUIT
OTHERWISE                               // above choices not met
   ? "have a nice day"
ENDCASE
```

#ifdef...#else...#endif is similar to the IF...ELSE...ENDIF structure. The significant difference is, that #ifdef...#endif are preprocessor directives which are already tested and resolved at the compile-time; the compiler then includes the code (placed between the #ifdef..#endif lines) only if the condition is met (or does not met for #else part, or with #ifndef respectively). On the other hand, the IF...ENDIF are executable statements, whereby the condition is tested (and the following code executed if the condition is met) at run-time of the application. The #ifdef directives are mostly used to include test-phase statements, or code for different platforms etc., for example

```
#ifdef FlagShip                 // defined automatically
# ifdef FS_WIN32                // defined in VFS for Windows
   ..FlagShip code specially for MS-Windows..
# else
   ..FlagShip code specially for Unix/Linux..
# endif
#else
..DOS Clipper code..
#endif
```

```
#define TEST_ONLY    // or -DTEST_ONLY compiler switch
....
#ifdef TEST_ONLY
? myvar1, myvar2
#endif
```

see more in section PRE and LNG.9.5.

2.5.2 Iteration constructs

There are two constructs for iteration, which perform looping through any number of statements or commands, until the end condition is met. Both may be aborted by the EXIT command or interrupted/continued by the LOOP command. The end condition is always calculated at the beginning of the loop pass. If the condition is not met, the loop is terminated and the control is passed to the first statement following the end-of-loop construct (NEXT or ENDDO).

FOR...NEXT (or FOR...ENDFOR for FoxPro compatibility) is usually used for incremental looping. The control structure repeats a defined count of loops, giving the start and end values. If the step interval is omitted, the default increment is one. Both end condition and the step interval may be changed within the loop (see more features in CMD.FOR).

```
DECLARE array [200]                    // declare array
FOR count = 1 TO LEN(array)            // repeat if count <= 200
   array [count] = count               // fill array elements
NEXT                                   // increase count +1
```

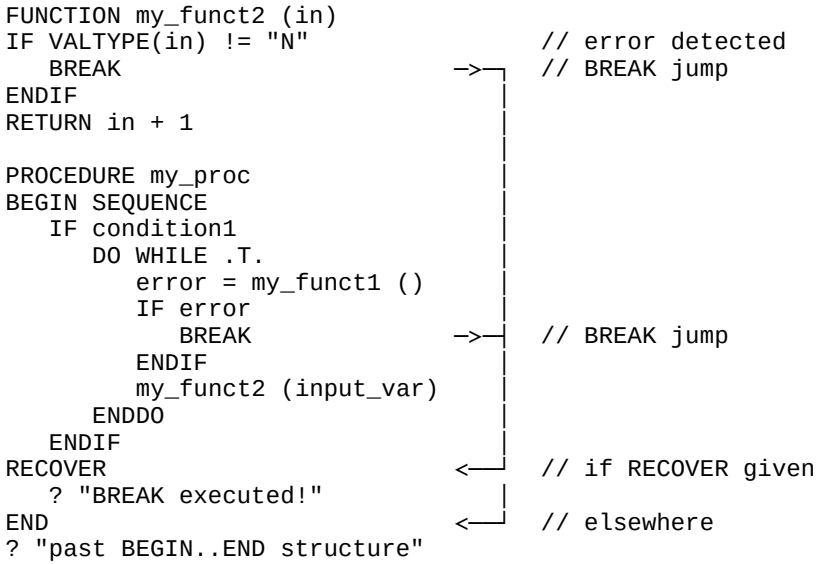
DO **WHILE...ENDDO** is mostly used for undetermined counts of loops. The repetition ends if the loop condition is not met. The following example will be executed 50 times if the database contains 50 records, or not at all if its empty (see more features in CMD.DO).

```
USE address                            // open database
DO WHILE .NOT. EOF()                   // if not end-of-file:
   ? name, zip, city                   // screen output,
   SKIP                                // fetch next record
ENDDO                                  // repeat again
```

There are also repeating functions available, like REPEAT(), SPACE(), AFILL(), AFILLALL(), AEVAL(), DBEVAL() etc. to fill a string or array with a value or to perform a specific loop action on arrays or on databases.

2.5.3 Interrupting Program Flow, Exceptions

The special control structure **BEGIN SEQUENCE...BREAK...END** allows the current program execution to be terminated and the program control to be passed to the next statement following the END construct. This break may be used from other control structures or at any depth of subordinate procedures or functions. It is similar to, but more flexible than the dBASE commands ON ERROR or RETURN TO MASTER, or the GOTO or BREAK statements in other programming languages.



For more details, refer to (CMD) BEGIN SEQUENCE.

2.6 Variables

A memory variable is a named place in memory where certain values can be stored temporarily. Memory variables can contain strings, numbers, dates, logical values, arrays, objects and code blocks. The program refers to a memory variable by its name, not its contents; the contents and type of each variable can vary without changing the name.

The variable name can contain any combination of letters (A..Z), numbers (0..9) and the underscore character ("_") whereby the **first character** is either alpha letter or underscore. Spaces and non-ASCII letters (e.g. -, /, $, #, ~, umlauts etc.) are not allowed. The name can be of any length but only the **first 10 characters** are significant. Thus, the VARIABLE_1, VARIABLE_12, and VARIABLE_13 all refer to the same variable.

The capitalization is not significant, as variable names are internally converted to uppercase. Thus, TEST_VAR, Test_Var and test_var all refer to the same variable. Avoid using reserved words (commands, keywords and standard functions) as variable names to stay compatible with other xBASE languages.

The number of memory variables that can be simultaneously used in a user program is in FlagShip practically unlimited (the Unix/Windows RAM memory size + swap area, dependent on the implementation of the operating system).

2.6.1 Variable Classes

In FlagShip, four different groups of variable exist. The lifetime of variables and their visibility depends on the variable declaration, which specifies the variable class:

- **Dynamic** (or dynamically scoped) variables: if the variable is declared as PRIVATE or PUBLIC, it will be created and fully managed at runtime. If a value is assigned to an undeclared or unknown variable, the runtime system will create a new autoPRIVATE variable. The usage of dynamically scoped variables is typical for interpreted languages, but is also fully supported by FlagShip for compatibility purposes. The disadvantage of such variables is the programmer having to keep track the mere program control of their visibility and the additional runtime overhead. Dynamic variables are available in all xBASE dialects. For further details see (CMD) PRIVATE, (CMD) PUBLIC.
- **Lexical** (also called statically scoped) variables (LOCAL and STATIC) are declared and visible only within the same module or program file. Because they are created and managed by the compiler, their usage allows to produce faster and more effective code. They are also available in Clipper 5.x. For further details see (CMD) LOCAL.
- **Typed lexical** variables (LOCAL..AS, STATIC..AS, GLOBAL..AS) are available in FlagShip (and partially in CA/VO) only. They are lexical variables with fixed storage type. Since additional runtime type checking may be omitted, usage results in very fast programs (up to 40 times faster then lexical vars). The C-like typed variables may also be directly reached

from the inline C statements without any conversion. For further details see (CMD) LOCAL...AS.

- **Field variables** are synonyms for database fields with the same name. They exist only as long as the corresponding database is open. The FIELD variable of the respectively selected database has **higher** read- precedence from the same named dynamic variable. Using the M->, MEMVAR->, FIELD-> and alias-> operators or FIELD and MEMVAR declarators prevents the misinterpretation between field and dynamic variables. Field variables are available in all xBASE dialects. For further details see CMD.FIELD.
- **Object instances** are similar to memory variables, but are tightly related with the class/object and accessible by the object selector only. See details in 2.11.2 and OBJ.1.

2.6.2 Initialization and Declaration

The variable declaration is an non-executable statement to tell the compiler the names of dynamic/field variables used in a module, or to create lexical and typed variables. The explicit declaration of dynamic/field variables avoids their misinterpretation. Note: if the compiler switch -w is used, all undeclared variables will be listed.

autoPRIVATE variables: no explicit declaration, initialization or definition of type of these dynamic variables is required. Instead, variables are created, initialized and their type defined at the moment they are used. Moreover, the type of the variable can change during program execution.

PRIVATE variables are created by the PRIVATE, DECLARE or PARAMETERS statement at runtime and initialized by default with NIL, which is the same as "undefined" in xBASE. The declaration is not mandatory. The MEMVAR declaration statement may also be used instead.

PUBLIC variables are created at runtime by the PUBLIC statement and initialized by default with FALSE. The MEMVAR declaration statement may be used in other (called) modules.

LOCAL and **STATIC** variables are created by the compiler with the declaration statement LOCAL or STATIC. Local variables are initialized with NIL by default.

MEMVAR declared variables are references to PRIVATE or PUBLIC variables, they are already initialized by the PRIVATE or PUBLIC statement.

Typed variables are created by the compiler with the declaration statement LOCAL..AS, STATIC..AS or GLOBAL..AS. C-like typed local variables are initialized with 0 (zero) by default, other typed variable by its EMPTY() value.

FIELD variables are created by the compiler, but visible and initialized automatically at run time with the actual field contents only if a database with such a field name is currently open.

2.6.3 Variable Scope, Visibility and Lifetime

If a variable is created as PRIVATE, its scope is the respective program module (procedure or function) and all modules subsequently called. Returning to the module on a higher level, releases (un-defines) the variable. If a PRIVATE variable with the same name exists in a module higher in hierarchy or is declared as PUBLIC, it will be hidden and temporarily inaccessible as long as a private copy exists.

The same rules as for PRIVATE apply also for implicit PRIVATE variables (autoPRIVATEs) and for variables declared by the DECLARE or PARAMETER statement.

A variable declared with the PUBLIC or CONSTANT command will not be released when the respective module is left. Such a variable remains visible to modules both "higher" and "lower" in the call hierarchy, but will be hidden by other PRIVATE (or PARAMETER), LOCAL, STATIC declarations.

The PRIVATE and PUBLIC variables can explicitly be destroyed (released) with the commands CLEAR MEMORY and RELEASE. These variables can be saved to a disk file with the command SAVE TO, and also restored from it with the command RESTORE FROM.

The scope of LOCAL, STATIC or TYPED lexical variables depends on where the variable is declared.

- **UDF-wide** scope: if the variable is declared within a procedure or function, the visibility is restricted to the module only.
- **File-wide** scope: if the variable is declared in a .prg prior to the first PROCEDURE, FUNCTION or other executable statement **and** the source file is compiled with the -na switch, the variable is visible for the whole .prg file. Note: Clipper 5.x doesn't support file-wide LOCAL variables, but STATICs only.
- Restricted-**application-wide** scope: GLOBAL..AS declared variables are visible the same as other typed variables (UDF or file wide), depending on where the variable is declared. The variable becomes visible also within other program files using the EXTERN GLOBAL..AS declarator there.

The LOCAL, STATIC or typed declaration will hide all other variable types with the same name. Other declarators with the same name on the same level are not allowed.

The LOCAL and typed LOCAL variables are always created and initialized on entry into the program module (or any UDF within the .prg for file-wide scope) and destroyed when returning from that module (or the .prg for file-wide scope). They are invisible in subsequently called modules, their values (or addresses) may however be passed as arguments.

The STATIC and typed STATIC variables are created and initialized on the first entry into the program module (or any UDF within the .prg for file- wide scope). The contents will be not destroyed when returning from that module (or file), but their names become invisible in

subsequently called modules (or files). The last value prior to the RETURN remains unchanged until the subsequent UDF (or file) is entered. Their values (or addresses) may however be passed as arguments.

2.6.4 Type of Variables

The type of a memory variable is determined at run-time by the contents the variable receives (except for typed variables). This type may change any time during the lifetime of a variable. The user program may at any time check the variable type by using the standard functions VALTYPE() and TYPE(). The possible types are:

- numeric (using floating point)
- numeric integer
- character
- date
- logical
- array
- screen
- code block
- object
- NIL (undefined)

Many standard functions (like STR(), SUBSTR(), VAL() etc.) require a fixed variable type and reports a run-time-error if a wrong variable type is used as argument. For comparisons (and other expressions) of two or more variables or constants, only the same or comparable variable types may be used.

Numeric Variable

A numeric variable (implicit or declared AS NUMERIC) stores its value as an 8 byte, double **float** (IEEE) number. The internal storage is hardware dependent, the precision is mostly 15 decimal places or more, the range is approx 10 ^ 307. For example:

```
pi = 3.141592653
qF = pi * r ** 2
```

Addition, subtraction, multiplication, division and modulus operations can be performed between two numeric variables. Commands SET FIXED and SET DECIMALS influences the output only, not the internal storage of numeric values. Many standard functions create or manipulate numeric variables, i.e. VAL(), ABS(), INT(), ROUND(), MIN(), MAX(), SQRT() etc.

> Tech note: FlagShip use IEEE double precision to store and calculate floating point numbers. It supports an extremely wide range of numbers, from 2.2250738585072014e-308 to 1.7976931348623157e+308. Integer values and constants are calculated by using the (precise) integer mathematic.

The internal representation of IEEE floating point are 64 bits splitted in sign, logarithm and mantissa with 14-17 significant digits. No human system of numeration can give a unique representation to every real number; there are just too many of them. So it is conventional to use approximations. For instance, the assertion that pi is 3.14159265358979 is, strictly speaking, false, since pi is actually slightly larger than 3.14159265358979; but in practice we sometimes use 3.14 in calculations involving pi because it is a good enough approximation of pi. The same approximation is known e.g. for division 1/3 which gives an endless number of decimal digits - which of course cannot be stored in the amount of available memory storage. So the stored result of this division is very-very close to the mathematical result, but need not be precise equal to. Also, the representation of a number by logarithm may be imprecise - even approximate, and known as "IT floating point representation", e.g. the number 123456 may be stored as 123456.000000001 or as 123455.999999999 (system dependant) in float representation.

Additional details: IEEE Standard for Binary Floating Point Numbers, ANSI/IEEE Std 754. New York: Institute of Electrical and Electronics Engineers (IEEE) 1985.

Conclusion: Floating numbers have (for practical use) a nearly unlimited range, but may be **imprecise**. Because of the approximation, a comparison of a numeric calculation with constant or with another calculation may fail - even if mathematically equal. If you detect any problems, use ROUND() or FIELDIS*() or compare by a small range, e.g.

```
num1 := (1 / 3) * 3    // num1 is mathematically 1, not so in IT :-)
num2 := 1
if abs(num1  - num2) < 0.0000000001   // instead of num1 == num2
  ... result is ok  ...
endif

if ROUND(num1, 5) ==  ROUND(num2, 5)
  ... result is ok  ...
endif

use mydbf                                 // field NUMBER = N,8,2
replace number with 10.01
? number == 10.01                         // most probably .F. (!)
? Round(number, 2) == Round(10.01, 2)     // .T.
? FieldIsEq("number", 10.01)              // .T.
```

See other examples in section LNG.2.9 Mathematical and Relational Operators and FUN. FIELDIS*() .

Hint: if you wish to always calculate precise, use the LOCAL...AS INTVAR type. This can be used also for decimals, e.g. the numeric "price" value N 8.2 can be stored and/or calculated in INTVAR variable as N10.0 or in N 10.0 field. It real value is simply "price * 100" and the displaying results are divided by 100. Since the supported range of INTVAR is approx +/- 2 giga = 2 billions = +/- 2,147,483,637 you will be able to store values *100 up to 20 millions with 2 deci digits by this method.

FlagShip supports also **integer** variables (implicit or declared AS INTVAR). The internal storage is long integer occupying 4 bytes (32 bits) and the representation is **precise** (as opposite to

floating numbers, see above), the valid range is system dependent, but at least -2147483648 to +2147483647 (approx +/- 2 Giga or 2 billions).

FlagShip compiler assumes (long) integer, when a constant is given without a decimal point, and a float number, when a decimal point is specified. You may explicitly **type** the variable to (double float) NUMERIC or (long) INTVAR by using the LOCAL...AS statement. If the variable is not typed, FlagShip automatically switches an integer type to floating number when required, i.e. on calculation with other float number, on integer overflow, after a division and so on.

Examples of numeric functions and associated commands:

ABS()	Returns the absolute value of a numeric expression
BIN*()	Binary operation on numbers
INT()	Converts a real/float value to integer
ROUND()	Rounds a numeric value to the specif. # of decimal places
STR(), VAL()	Converts a number to string / string to number
MIN(), MAX()	Determines the lower/greater of two numbers
%, MOD()	Returns the standard/dBASE III modulo of two numbers
EXP()	Evaluates the e^x expression
LOG()	Returns the natural logarithm of a numer. expression
SQRT()	Returns the square root of a numer. expression
SET DECIMALS	Sets the number of displayed decimal places
SET FIXED	Determines how to display numeric values
`+ ++ - -- * ** ^ / %`	Mathematical operators (see LNG.2.9)
`= == != # <> < > <= >=`	Mathematical comparison (see LNG.2.9)

Additional functions for numerics manipulation are available in the optional FS2 Tools library, see section FS2.

Character Variable (string)

A character variable (implicit or declared AS CHARACTER) in FlagShip may contain 0 to 2 billion characters in ASCII format, each with any value from 0 to 255. The binary value 0 is supported by the most string operations or by programmer's request, see LNG.2.6.5. For example

```
x = "John" + " Smith"
y = ""
long_text := mymemo
REPLACE mymemo WITH x + " " + long_text
```

defines variable x as a character variable with the value "John Smith", the variable y becomes zero length (null string). Addition and subtraction (see expressions) can be performed between two character variables.

Note 1: if the variable contents is to be stored into a database or memo field, or to create programs portable to other xBASE languages, the string length should not exceed 64 KBytes.

Note 2: the contents of the SAVE SCREEN command or SAVESCREEN() function is in FlagShip stored in special "screen" variables instead of character variables.

Assigning a memo field mymemo from above example creates a character variable long_text. Character variables may be stored into memo fields using the REPLACE command, the FIELDPUT() function or := assignment. Many standard functions create, modify or manipulate character variables, i.e. SPACE(), CHR(), STR(), SUBSTR(), LOWER(), AT(), LEN() etc.

Examples of character/string functions:

LEN(), EMPTY()	Retrieves the length of a string/detects null-string
ASC()	Converts an ASCII character to its num. equivalence
CHR()	Converts an ASCII num. equivalence to a corresp. character
STR(), VAL()	Converts a number to string / string to number
SUBSTR()	Extracts the specified part of the given string
LOWER(), UPPER()	Converts a string to lowercase/uppercase
AT(), RAT()	Returns the left/right position of a substring within a string
LEFT(), RIGHT()	Extracts the specif.no. of leading/right chars from string
LTRIM(), RTRIM()	Removes all leading/trailing spaces from a string
TRIM(), ALLTRIM()	Removes all trailing or leading+trailing spaces from a string
SPACE(), REPLIC()	Forms a string of n spaces/repeating a string n-times
STRTRAN()	Searches and replaces within a character string
STUFF()	Performs delete, insert and replace within a string
STRZERO()	Converts a num. value to a string with leading zeros
PADxxx()	Fills the beginning/end of a string with characters
TRANSFORM()	Formats an expression according to the given PICTURE
SOUNDEX()	Converts character strings to a soundex code
MEMOEDIT()	Displays or edits strings or memo fields
MEMOxxx()	Several functions for formatting a string
FS_SET("zerobyte")	Enables the embedded chr(0) in string, see 2.6.5.
+ -	String operators (see LNG.2.9)
= == != # <> < > <= >=	String comparison (see LNG.2.9)

Additional functions for character manipulation are available in the optional FS2 Tools library, see section FS2.

Date Variable

A date variable stores its value as 4 byte, long integer by calculating the days since the 1st January of the Year 1 AD. For example:

```
yesterday = DATE() - 1
birthdate = {09/12/1993}            // or CTOD("09/12/1993")
```

defines yesterday and birthdate as a date variable. A date variable can be added with a numeric variable/constant and a numeric variable/constant can be subtracted from a date variable. Commands SET DATE, SET CENTURY, SET EPOCH influence the output and input of date variables only, not the way they are internally stored. There are standard functions for date conversion or handling available, e.g. DTOC(), CTOD(), DATE(), DOW(), DAY(), MONTH(), WEEK() etc.

Examples of date commands and functions:

DATE ()	Returns the system date in form of a date value
DAY(),MONTH(),YEAR()	Extracts the day/month/year from a date value
CTOD(),DTOC(),DTOS()	Converts a date string to a data value and vice-versa
DOW(), CDOW ()	Finds the day of the week/its name for a date value
SET DATE	Sets the format for date values
SET CENTURY	Toggles the input/display of century digits for dates
SET EPOCH	Sets the epoch of date values
SET ()	Reports/sets global default settings
+ ++ - -- * /	Mathematical operators on date value
= == != # <> < > <= >=	Mathematical comparison (see LNG.2.9)

Additional date functions are available in the optional FS2 Tools library, see section FS2.

Logical (boolean) Variable

A logical (boolean) variable stores its value as a single ASCII character of value "T", "F", "Y" or "N". For example:

```
state = .T.
ok    = LEN("text") > 5
```

defines state as a logical variable with the value "true", ok becomes "false". .AND., .OR. and .NOT. (or !) operators can be applied to the logical variable. Some standard functions return logical values, i.e. EMPTY(), EOF(), FOUND(), RLOCK() etc.

Selected functions return logical values:

EMPTY ()	Determines if the result of an expression is empty
ISCOLOR ()	Determines if the terminal definition has color capabilities
FILE ()	Determines whether a file exists in the defined path
EOF ()	Finds out if there was an attempt to move past the last rec
DELETED ()	Reports if the current record is marked as "deleted"
USED ()	Determines if a .dbf is open in the selected working area
FLOCK(), RLOCK()	Locks the .dbf file/record before write access
NETERR ()	Checks the error status in multi-user environment
`.AND.  .OR.  !  .NOT.`	Operators on logical values (see LNG.2.9)

Arrays

Memory arrays are just a collection of memory variables. Every single array element is, in fact, a memory variable. FlagShip supports single- and multi-dimensional arrays. Each of the at most 65535 dimensions can contain up to 65535 elements.

Arrays have to be declared before use so that space for their elements can be reserved. This is done by the usual declaration statements DECLARE, PRIVATE, PUBLIC, LOCAL and STATIC or at runtime using the ARRAY(), ACLONE() or ACOPY() functions. The size of an array can be changed using AADD() or ASIZE(). Other functions, like AINS(), ADEL(), AFIL(), AEVAL(), ASORT() or ASCAN() insert or delete array elements, fill arrays with a value, evaluate a code block on arrays, sort or search for value in arrays.

```
LOCAL   aa[25,80], bb := {"text",3,.T.,{2,4,5}}, cc
DECLARE xx[12], yy[4][5]
PUBLIC  zz[0]
cc := ARRAY(22)
```

In the above example, two one-dimensional arrays: 'xx' with 12 elements and 'zz' with 0 elements, and two-dimensional arrays: 'aa' with 2000 and 'yy' with 20 elements, as well as array 'cc' with 22 elements were declared. The non-symmetric array 'bb' was also declared and initialized with 4 elements in the first dimension, the 4th element containing an sub-array with 3 elements.

The elements of an array are numbered starting by 1, the array size = number of array elements in every dimension is reported by LEN(array). Accessing an array, the ordinal number of the element is enclosed in square brackets, e.g. xx[5] access the fifth element of the array 'xx'. On multidimensional arrays, both the syntax yy[3,2] and yy[3][2] will access the same element of a two-dimensional array 'yy'.

Note, that various elements in the same array can be of different type. The following code:

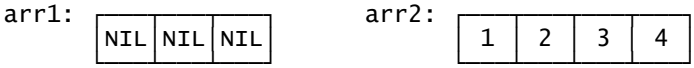
```
AFILL (xx, "")
cc := ARRAY (5)
yy[2][3] = "test"               // or:  yy[2,3] := "test"
? LEN(bb), LEN(bb[4])           // 4  3
```

```
FOR i := 1 TO LEN(aa)            // = 1 to 25
   FOR j := 1 to LEN(aa[1])      // = 1 to 80
      aa [i, j] := i * j
   NEXT
NEXT
dd := aa                         // 'dd' is a link to 'aa'
ee := ACLONE(aa)                 // 'ee' is a copy of 'aa'
```

will fill all elements of the array 'xx' with a null-string, create a new array 'cc' with 5 NIL (empty) elements, assign a string "test" to an array element - thus defining that element as a character variable, and fill the array 'aa' with the specified value. It also creates a link and copy of the 'aa' array, where change of 'dd' element changes 'aa' element as well, but changing 'ee' element lets the 'aa' element untouched, see below.

The above example also demonstrates how to determine the array size: the function LEN(array) returns the size of the first dimension, LEN(array[1]) of the second dimension, LEN(array[1,1]) of the third dimension of symmetric array, and so on.

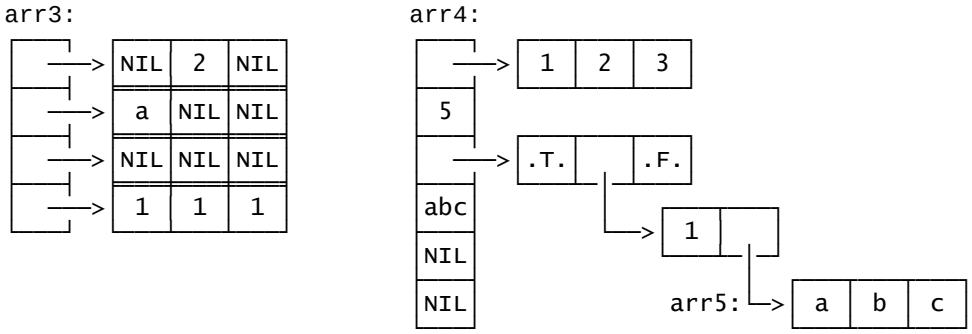
A one-dimensional array is stored in sequential order, e.g. DECLARE arr1[3] or arr2 := {1, 2, 3, 4} :



FlagShip supports both **symmetric** and nested **multi-dimensional** arrays. In fact, a symmetric multidimensional array is just a special case of a nested array. The declarations:

```
LOCAL arr3[4,3], arr5 := {"a", "b", "c"}
LOCAL arr4 := {{1,2,3}, 5, {.T., {1,arr5},.F.}, "abc",NIL,NIL}
arr3 [1,2] := 2
arr3 [2,1] := "a"
AFILL (arr3[4], 1)
```

will create the following structures:



which can be verified by:

```
? VALTYPE(arr3), LEN (arr3), LEN(arr3[1])     // "A"   4   3
? VALTYPE(arr3[1]), VALTYPE(arr3[1,2])        // "A"  "N"
? VALTYPE(arr3[2]), VALTYPE(arr3[2,1])        // "A"  "C"

? VALTYPE(arr4[1]), VALTYPE(arr4[1,2])        // "A"  "N"
? VALTYPE(arr4[2]), VALTYPE(arr4[3])          // "N"  "A"
? VALTYPE(arr4[3,2,2]), LEN(arr4[3,2,2])      // "A"   3
```

Assigning an array name to a variable using the = or := operator creates a second reference only, pointing to the same array body. To make a physical copy of arrays, use the function ACLONE() instead. Assigning an array element to a variable creates a single variable copy of the same type as the array element is. Comparing two array names using the == operator returns a TRUE result if their addresses (of the array body) are identical. To compare the contents of two arrays, element-by-element comparison, by FOR..NEXT or AEVAL(), has to be used.

```
PRIVATE arr1 := {1, 2, "test"}, arr2, arr3
arr2 := arr1
arr3 := ACLONE (arr1)
arr1 [1] := 5
arr2 [2] := "x"
? arr1 == arr2, arr1 == arr3          // .T. .F.
? arr1[1], arr2[1], arr3[1]           //  5   5   1
? arr1[2], arr2[2], arr3[2]           //  x   x   2
RELEASE arr1
? arr2[2], arr3[2]                    //      x   2
```

In FlagShip, array elements can be SAVEd to and RESTOREd from disk files, see commands SAVE TO and RESTORE FROM and the compatibility FS_SET("memcomp") function.

Selected array functions

ARRAY ()	creates an un-initialized array
ACOPY ()	copy one array into another
ACLONE ()	duplicates an array
AADD ()	adds a new array element
AINS (), ADEL ()	insert / delete an array element
ASIZE ()	resizes an existing array
AFILL ()	fill 1-dimensional array with value
AFILLALL ()	fill any type of array with specific value
ATAIL ()	returns the last element of a given array
AEVAL ()	executes a code block on each array element
AFIELDS ()	info about .dbf fields
ASCAN ()	seek within an array
ASORT ()	sorts an array
ACHOICE ()	full screen menu input/output
ADIR ()	fills arrays with info about Unix/Windows files
DIRECTORY ()	creates an array with the Unix/Windows file/directory info
DBCREATE ()	creates a .dbf of given structure described in an array

Screen Variable

FlagShip stores the contents of the command SAVE SCREEN TO... or function SAVESCREEN() in special variables of type "S". These variables save a pointer to a WINDOW structure, which may be modified at the low level by using the Extend System (see example in the _retscw() function).

> **Note:** FlagShip screen structure differs significantly from the simple video buffer of Clipper in MS-DOS. FlagShip handles all the differences automatically using the optimizations of the curses library (in Terminal i/o mode) or internally by bitmaps in GUI mode. See more information about terminals in REL release notes and the section SYS.

Normally, no program changes to other xBASE dialects (e.g. Clipper, dBASE, Fox) are necessary to store or restore screen contents. To save/read the screen contents in/from text or .mem files, character fields or memo fields, SCREEN2CHR() or CHR2SCREEN() translating functions may be used. For converting DOS screen to FlagShip and vice versa, SCRDOS2UNIX() and SCRUNIX2DOS() functions are available (but for Terminal i/o mode only).

```
LOCAL scr1, scr2
scr1 := SAVESCREEN (0,0, MAXROW(),MAXCOL())
scr2 := SAVESCREEN (10,5, 20,40)
// any temporally output
RESTSCREEN (10,5, 20,40, scr2)        // restore original
REPLACE mymemo WITH SCREEN2CHR(scr1)  // save in database
```

In GUI, the structure of the screen variable <varS> is incompatible to <varS> from Terminal i/o mode. In GUI, it is compressed or uncompressed bitmap object an hence cannot be extracted by Chr2Screen(), Screen2chr() or other low-level curses routines like ScrUnix2Dos() and ScrDos2Unix(). It also cannot be saved to or restored from a memo file. But the SAVE/ RESTORE SCREEN work by the same way in GUI and Terminal mode. See also SAVE SCREEN, SaveScreen() for further details.

Associated commands and functions:

SAVE SCREEN	Saves the screen contents to a SCREEN variable
RESTORE SCREEN	Displays a previously stored screen contents
SAVE	Saves memory variables to a .mem file
RESTORE	Retrieves memory variables from a .mem file
SAVESCREEN ()	Saves a specified screen region to a variable
RESTSCREEN ()	Restores a screen region from memory variable
CHR2SCREEN ()	Converts a string to a screen variable
SCREEN2CHR ()	Converts a screen variable to a character string
SCRDOS2UNIX ()	Converts a DOS screen content to FlagShip screen variable
SCRUNIX2DOS ()	Converts FlagShip screen variable to DOS screen output
FS_SET("memcom")	Sets full compatibility of the .mem files to xBASE

Code Block Variable

Code blocks (see also LNG.2.3.3) may be assigned to a variable, which then becomes type "B" and contains the address of a code block (similar to defining a function name). Releasing this variable doesn't release the code block (because it is a compiled piece of code), it just becomes inaccessible.

```
LOCAL blkVar := { |par| QQOUT("[", par, "]") }
EVAL (blkVar, "text")
EVAL (blkVar, "other text")
```

See also description of code blocks in chapter LNG.2.3.3.

Object Variable

Objects (see sections OBJ and LNG.2.11) are stored in special variables typed "O". This is a special array containing the instance variables. The object variable is created using a special class-creator-function, like GETNEW(), TBROWSENEW(), ERRORNEW() etc. or by instantiating the object, see chapter 2.11. The access to the object (variable) is done using the send operator ":".

```
LOCAL myget, myvar := 12345
myget := GETNEW(5,0, {|par| if(par==NIL, myvar, myvar := par)} )
myget:name := "MYVAR"
? myget:row                          // 5
myget:DISPLAY()                      // show the GET
READMODAL (myvar)                    // read
```

Note: when starting a FlagShip compiled application an empty PUBLIC GETSYS[0] array is created. GETSYS elements carry get objects created by the @...GET command. The objects (here elements of GETSYS) subsequently get cleared by the READ command.

Assigning an object variable to other variable using the = or := operator creates a second reference only, pointing to the same object body, very similar to an array assignment. To make a physical copy of an object with different data (instances), instantiate it instead.

NIL Variable

On creation, all variables (except for TYPED and PUBLIC variables) are assigned an "U" undefined type. Such "clean" variables may be checked using functions VALTYPE() and TYPE() or by direct comparison with the NIL constant. Assigning NIL to the variable will also "clean" it.

```
LOCAL      xyz
PARAMETERS par
PRIVATE    abc
IF TYPE("abc") == "U"
```

```
   abc := "first value"
ENDIF
IF VALTYPE(xyz) == "U"
   xyz := 1
ENDIF
IF par == NIL
   ? "argument has to be given"
ENDIF
```

Note: you cannot check unknown or invisible variables by NIL comparison, use TYPE() function instead.

2.6.5 Binary 0 Characters in Strings

As mentioned earlier, FlagShip uses the C programming language as a vehicle for portability. It translates the .prg statements into an intermediate ANSI C program, which may then be compiled to produce native executable code.

According to the C language definition, binary zero characters, represented by CHR(0), normally terminate a string. All significant string manipulation routines from the standard Unix/Windows libraries, like strlen(), strcat(), strcpy(), strchr(), strstr(), printf() etc. use this C convention.

For your convenience, FlagShip automatically supports the usage of binary zero characters within a string for the most used string operations and for some other operations on programmer's request. The **automatic** support of embedded \0 byte is built in:

- all assignment operators := = += -=
- all comparison operators = == != <> # $ > < >= <=
- following string handling commands and functions: ?, ??, ALLTRIM(), ASC(), AT(), BIN2I(), BIN2L(), BIN2W(), CTOD(), DESCEND(), EMPTY(), FREAD(), I2BIN(), L2BIN(), LEFT(), LEN(), LOWER(), LTRIM(), OUTSTD(), PADx(), QOUT(), QQOUT(), RAT(), REPLICATE(), RIGHT(), RTRIM(), STRPEEK(), STUFF(), SUBSTR(), TRIM(), UPPER()

The following operators and standard functions **optionally** support the embedded zero byte when FS_SET ("zerobyte", .T.) is set:

- FREADSTR(), FREADTEXT(), STRLEN(), STRPOKE(), STRTRAN(), STRZERO(), TRANSFORM(), _parclen(), _retclen(), _storclen()

The CHR(0) containing strings will be handled according to the standard C convention in all other functions, commands and macros. The same is true, when the embedded-zero-option is disabled (the default setting) using FS_SET("zerobyte", .F.) in the optionally supported operators or functions. This means, the resulting string will be shortened up to (but excluding) the first binary zero and the operators/functions will only accept strings in this shortened format.

2.6.6 Variable Type Declaration

Typing of variables (and typing/prototyping the UDF parameters and return value) informs the compiler, that this variable can contain only the specified data type, e.g. float numeric, integer, string, date, code block, object and so on. This increases the stability of the application significantly and may also increase the execution speed up to factor 40 (when using C-like types).

Already at the compile-time, the compiler checks assignments to and comparison of **typed** variables, if the argument type is known at this time (i.e. when assigning a typed variable, constant or prototyped function). The same is valid for arguments passed to prototyped function. If the types are the same or compatible (e.g. NUMERIC and INTVAR), an automatic conversion is performed. If the types are incompatible, a compiler error occurs. If the argument type is unknown at this time, and the -w2 or -w3 switch is set, these ambiguous variables are reported by the compiler as warnings.

At the run-time of the application, the assignment to a typed variable, its comparison and arguments passed to prototyped function are also checked. On incompatible data types, a run-time error occurs, which avoids later crash otherwise.

When the variable or parameter contains different data types, you may declare it AS USUAL or avoid the typing/prototyping at all. You have then to check the data type manually by using TYPE() or VALTYPE() functions, to avoid the run-time error "datatype mismatch" later.

Examples (see also CMD.LOCAL..AS and CMD.PROTOTYPE):

```
LOCAL varN AS NUMERIC
LOCAL varI AS INTVAR
LOCAL varC AS CHARACTER
LOCAL var1, var2          // = AS USUAL

varN := 10 + 0.5          // ok, converted to NUMERIC float
varI := 10.55             // ok, compiler warning, converted to 10
varI := varN              // ok, compiler warning, converted
varN := "any string"      // compile-time error
varC := varN              // compile-time error
var1 := varN              // ok, var1 is not checked, since USUAL
var2 := var1 + "string"   // run-time error (num + char)
varN := SUBSTR(varC,1,2)  // compile-time error, if stdfunct.fh used
                          // otherwise run-time error (num := char)
```

2.7 Literal Constants

A constant is a fixed value used to initialize variables, to be passed as an argument to a UDF or to be compared with another variable.

During the compiler phase, FlagShip optimizes the constant usage by pre-calculating known expressions. Hence, the statement a := b + 5 + 7 will be translated as a := b + 12 or the statement x := "ab" + "cd" as x := "abcd". Further optimization is also done by the C compiler.

The constant types are:

- numeric
- character
- date
- logical
- array
- NIL (undefined)

Numeric constant

A numeric constant is an optionally signed decimal number represented by ASCII characters in the code. It will be converted to an 8 byte, double float for NUMERIC variables and to long integer for INTVAR variable type. An usual numeric variable stores its value as an 8 byte, double float (almost IEEE) number. The format of internal storage is hardware dependent, but the precision is usually 15 decimal places or more, the range is approx +/- 10 ^ 308. An INTVAR variable can store -2 147 483 648 to 2 147 483 647. Example:

```
3.141592653589793
123
-0.3
3 + 0.5 / 9
```

The FlagShip compiler assumes a (long) integer, when a constant is given without a decimal point, and a float number, when a decimal point is specified. Of course, it is then converted to the type of the NUMERIC or INTVAR variable, if such is declared (e.g. by the LOCAL...AS statement).

Character constant

A character constant is any sequence of characters delimited with the special character pairs: double quotes (""), quotes (''), brackets ([]) and typographical quotes (` '):

```
"double quotes"
'single quotes'
```

```
[square brackets]
`back quote and single quote'
"result's contents " + [is "one" ] + 'constant string'
```

The opening delimiting character determines which character must be used to close the pair. Between the two delimiters, all the other alternative delimiters can be used. A pair of delimiters with no characters between them is considered as null string. Its length is zero. Example of null strings:

```
""    -or-    ''
[]    -or-    ``
```

Binary 0 characters = CHR(0) are also supported by most string operations in FlagShip, see LNG.2.6.5.

You may enter **special characters** within the **constant** as \nnn where the nnn is an octal representation (three digits, each 0…7) of the character, e.g.

```
u_iso := "xx\374yy"        // = xx<u_umlaut_iso>yy,   len(u_iso) = 5
u_asc := "xx\201yy"        // = xx<u_umlaut_ascii>yy, len(u_asc) = 5
```

which is equivalent to

```
u_iso := "xx" + chr(252) + "yy"   // 252 deci = 374 octal
u_asc := "xx" + chr(129) + "yy"   // 129 deci = 201 octal
```

On the other hand, if you need to store backslash "\" followed by 3 or more digits in a string constant, you will need to split this constant to avoid this implicit special character conversion, or use \134 for the backslash itself, or use the constant PATH_SLASH e.g.

```
str1 := "xx\" + "374yy"              // = xx\374yy, len(str1) = 8
str2 := "xx" + chr(92) + "374yy"     // = xx\374yy, len(str2) = 8
str3 := "xx\134374yy"                // = xx\374yy, len(str3) = 8
str4 := "xx\aa\yy"                   // = xx\aa\yy, len(str4) = 8
str5 := "xx" + PATH_SLASH + "374yy"  // = xx\374yy, len(str5) = 8
```

If the backslash is a part of PATH constant in MS-Windows, you may simply use slash instead, since FlagShip automatically converts slash to backslash (or backslash to slash in Linux) during the file access, so you may use

```
myFile := "/xx/374yy.txt"            // same as "\xx\374yy.txt"
```

Since the conversion is done by the FlagShip preprocessor, you may in doubt check the <programname>.bp output created by -a compiler switch.

See also SET SOURCE ANSI/ISO, Ansi2oem() and Oem2ansi() for ISO/ASCII conversion and the -iso compiler switch in section FSC.1.3 and in LNG.5.3, LNG.5.4 (internalization), as well as the compilable examples in <FlagShip_dir>/examples/ umlauts.prg or pc8lines.prg.

If you wish to enter long string constants exceeding your source-code editor width, use continuation on subsequent lines, e.g.

```
cLong := ".......10.......20.......30.......40.......50" + ;
         ".......60.......70.......80.......90......100" + ;
         ".......110......120......130......140......150" + ;
         ".......160......170......180......190......200"
```

and so forth. FlagShip compiler will then concatenate these lines into one large string constant. The size of a string constant is limited in FlagShip compiler by 8KB (i.e. approx. 100 lines each 80 characters), but the most C compilers have lower limits (e.g. MS-VC only 2KB, Unix compilers usually 8-16KB). You may exceed these limits by splitting the constant (i.e. using concatenation of FS variables), e.g.

```
cLong := ".......10.......20.......30.......40.......50" + ;
         ".......60.......70.......80.......90......100"
cLong += ".......110......120......130......140......150" + ;
         ".......160......170......180......190......200"
```

which then works unlimited up to the total string size of 2GB. Note: for repetitive character constants, it is much more efficient to use SPACE(n) or REPLICATE("...",nn) functions, or an addition of shorter constants (variables) than one large string constant.

Date constant

Date constants denote any valid date expression. Examples of a date constant are:

```
{31.12.1991}
{12/31/1990}
{01-12-1990}
{31.08.91} + 3 * 7
{0.0.0}             // empty date, same as ctod("")
```

The delimiters are curly brackets { }, but the actual format of the date enclosed depends on the respective SET DATE, SET EPOCH and SET CENTURY flags. The digit delimiters in the constant between the day, month and year are dot, slash and dash (see also CTOD()). A valid date range in FlagShip is 01/01/0001 to 12/31/9999.

Logical constant

A logical constant is a single ASCII character delimited by periods. Upper and lower case T, F, Y, N are allowed, e.g.:

```
.T.
.n.
```

NIL constant

The NIL constant is a synonym for "undefined value". It may be used to check un-initialized variables or to un-initialize a variable.

```
FUNCTION xyz (abc)
IF abc != NIL                       // argument entered ?
   abc := NIL                       // yes, ignore it
ENDIF
```

Array constant

Array constants (also called literal arrays) are used to set up an array and initialize its elements. The array constant is defined by curly brackets { } with comma separated elements/initializer. The array constant may include other constant types or valid expressions.

```
LOCAL   arr1 := {1}                             // arr1[1] = 1
LOCAL   arr2 := {5, 6, "text", DATE()}          // arr2[4]
PRIVATE arr3 := {{1,2}, {3,4}, {5,6}}           // arr3[3,2]
PUBLIC  arr4
LOCAL   arr5 := { }                             // empty array
arr4 := {DATE(), TIME(), {1,2,.T.}, "X"}        // arr4[4] non-symmetric
strArray  := "{1, 2, .T.}"
privArray := &strArray                          // macro creation
? VALTYPE(privArray), LEN(privArray)            // A  3
? VALTYPE(arr4[1]), LEN(arr4), LEN(arr4[3])     // D  4  3
? VALTYPE(arr4[3]), VALTYPE(arr4[3,3])          // A  L
? VALTYPE(arr2), LEN(arr2), VALTYPE(arr2[4])    // A  4  D
? LEN(arr3), VALTYPE(arr3[1]), LEN(arr3[1]), ; // 3  A  2
  VALTYPE(arr3[1,1]), arr3[3,2]                 // N  6
aeval(arr4, {|n| qqout(VALTYPE(n) + " ") } )    // D  C  A  C
? LEN(arr4), LEN(arr4[3]), LEN(arr4[4])         // 4  3  1
? VALTYPE(arr5), LEN(arr5)                      // A  0
```

2.8 Expressions

An expression is a

- memory variable,
- constant,
- database field

or any of these combined by one or more operators or function calls. The operator may be an inline-assignment, concatenation, comparison, mathematical or logical evaluation. Expressions always resolve to a character, numeric, date, logical or screen type.

Parentheses for grouping ( ) may be nested to any depth. The only limitation to the expression size is the buffer of FlagShip and the C compiler (most often 4 Kbytes). For readability, spaces or tabs may be included between the operators. Examples of valid expression usage:

```
x := y + z / ( 2 + w)
SKIP (newpos - oldpos +1)
? "Since last update " + ;
   STR(lastdate - DATE()) + " days have passed."
DO myproc WITH 55, DATE() +1
```

Expression List

Where a single expression may appear, like arguments of commands, declarations, body of code blocks etc., an expression list may also be used. The expression list is similar to a parameter, argument or declarator list.

The expression list consists of two or more commas separated independent expressions, enclosed in parentheses (which may be omitted in code block bodies). The list is evaluated from left to right. The rightmost expression will be passed, returned or used in place of a single expression. Examples:

```
a := (b:=3, c:=4-b)                        // 1)  b := 3
                                           // 2)  c := 4 - b
                                           // 3)  a := c
d := {|par| a++, b += par, QQOUT(b) }      // 1)  a := a + 1
                                           // 2)  b := b + par
                                           // 3)  QQOUT (b)
```

2.9 Operators

FlagShip uses five different operator groups: assignments, mathematical, relational, logical operators and character concatenation. There are also other special operators available: the macro-operator (see LNG.2.10) and the send-operator (see LNG.2.6.4 and 2.11).

Assignments

Assignments are used to assign a value to a variable of any type. The resulting variable type will be that of the value or expression being assigned.

Operator	Operation	Description
=	assignment	stores a value or an expression <exp> into a variable, identical with the command STORE. If specified within an expression, it is interpreted as the equality comparison operator.
:=	in-line assign	similar to the = operator, but can be specified within expressions and can be used in a declaration statement to initialize the specified variable. Should be preferred over = for a better readability, since = is also comparator.
+=	addition	identical to <var> := <var> + <exp>
-=	subtraction	identical to <var> := <var> - <exp>
*=	multiplication	identical to <var> := <var> * <exp>
/=	division	identical to <var> := <var> / <exp>
%=	modulo	identical to <var> := <var> % <exp>
^=	exponentiation	identical to <var> := <var> ^ <exp>
**=	exponentiation	identical to <var> := <var> ** <exp>
+	unary +	unary positive <varN>
-	unary -	unary negative <varN>
!	unary .NOT.	unary reverse <varL>

If the resulting variable <var> is not visible or does not exist, an auto-PRIVATE variable is created and assigned the result of <exp>. If the reference to the resulting variable is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), the variable is always assumed to be MEMVAR. If the resulting variable is typed (see 2.6.6), the validity of the assignment is checked at compile-time and run- time.

Assigning an variable of type N, I, C, D, L to other variable via = or := operator creates a physical copy of the source with an own memory storage.

Assigning an array or object variable to other variable using the = or := operator creates a second **reference** only, pointing to the same array or object body. To make a physical copy of an array, use ACLONE(). To make a copy of an object with different data (instances), instantiate it with the same class. Note: all objects of the same class always use the available class code (methods) shared, i.e. the instantiation does not duplicate the code but the data only.

When **assigning** a FIELD variable, it must be defined using the FIELD statement or the FIELD-> or alias-> must precede the variable name (otherwise an autoPRIVATE variable is created).

Compound assignments (`+=, -=, *=` etc.) perform the corresponding operation before doing the assignment. Compound assignments may be used on FIELD variables in the same way as with `:=` assignments.

Examples:

```
LOCAL value := 10, ok := .T.
FIELD fvar
avar := bvar := cvar := value ** 3
IF (olddate  := (DATE() - 31)) = CTOD("12/20/79") ; ... ; endif
if !ok ; ? "error" ; endif          // print error if ok is .F.
address->name:= "Miller"
FIELD->date  := address->tempdate := olddate
fvar  = "text"                      // REPLACE fvar WITH "text"
value = 55
cvar  = (avar - 20) * value
ok    = value  = 10                 // ok = .F.    value = 55
ok    = value := 10                 // ok = 10 (!) value = 10

bvar *= cvar - 10                   // bvar := bvar * (cvar - 10)
? SQRT (value += 6)                 // out: 4   value = 16
value = -value                      // value = -16
```

Mathematical Operators

Mathematical operators perform operations on numeric variables, expressions, or fields (except for post/pre-increment and post/pre-decrement which modify memory variables only). The following operators are possible:

Symbol	Operation	Description
+	addition	returns <expN1> incremented by <expN2>
-	subtraction	returns <expN1> decremented by <expN2>
*	multiplication	returns <expN1> multiplied by <expN2>.
/	division	returns <expN1> divided by <expN2>. If the divisor <expN2> is zero, the division operation brings out a warning and returns zero.
%	modulo	returns remainder of <expN1> divided by <expN2>. If the divisor <expN2> is zero, modulo brings out a warning and returns 0
^ or **	exponentiation	raises <expN1> to the power of <expN2>
++var	pre-increment	increases the value of <var> by one, before the value gets used. Changes memory variables only.
var++	post-increment	increases the value of <var> by one, after the value is used. Changes memory variables only.
--var	pre-decrement	decreases the value of <var> by one, before the value gets used. Changes memory variables only.
var--	post-decrement	decreases the value of <var> by one, after the value is used. Changes memory variables only.

The pre/post increment/decrement changes the content of memory variables only; database fields remain unchanged. Expression will be evaluated, the result copied into temporary variable, pre-increment or pre-decrement increases (or decreases) the temporary variable by 1 and this temporary variable is then used as such in the statement rest, see example below.

Mathematical operators may also be used with date expressions but in the following ways only:

```
1. <expD> = <expD> + <expN>    // or: <expD> += <expN>
2. <expD> = <expD> - <expN>    // or: <expD> -= <expN>
3. <expN> = <expD> - <expD>
```

Examples:

```
LOCAL value
FIELD fval
value := (3 + SQRT(9)) * 2              // 12
value := 3 + SQRT(9) * 2                //  9
? value++, value                        // 10  11
? ++value, value                        // 12  12
? value++ * 2, value                    // 24  13
? DATE() - CTOD("07/15/93")             // 47

USE test
replace fval with 20
value := 10
? ++fval, fval                          // 20 20
? ++fval + value, fval                  // 30 20
? ++(fval + value), fval                // 31 20

num := 4
value := ++num * 2 / 3                  // => ((4 + 1) * 2) / 3
? num, value                            // 5  3.33
num := 4
value := ++(num * 2) / 3                // => ((4 * 2) + 1) / 3
? num, ++(num * 2), (num * 2)++, value  // 4  9  8  3.0
```

Note that all mathematical operations are performed using floating point arithmetic (and coprocessor, if available) for all NUMERIC variables and constants. Due to internal HW storage and rounding of (double) floating numbers, you may get different results when truncating the integer part of a complex expression from e.g. simple numeric constants.

Example: the comparison `INT(SQRT(3**2)) == 3` etc. may result in .F. on some hardware, since SQRT(3**2) may return 2.9999....9999 instead of 3. Similarly, `CHR(INT(9^2/3))` may result in either CHR(27) or CHR(26).

Therefore, the always safe way to get a hardware independent integer part of a complex mathematical expression is either:

- adding a small fragment to it, e.g. INT(SQRT(3**2) + 0.01) == 3
- or using the ROUND() function, e.g. ROUND (SQRT(3**2), 0) == 3
- or using the INTVAR typed variables

Simple mathematical operations like increment, addition, subtraction, multiplication etc. will seldom produce rounding differences.

See further tech details about floating point numbers in section LNG.2.6.4

Relational Operators

Valid relational operators are listed below. The expressions being compared must be of the same type.

Operator	Operation returns TRUE if:
<	<exp1> is less than <exp2>
>	<exp1> is greater than <exp2>
=	<exp1> is equal to <exp2>, conditional match on strings
==	<exp1> is exactly equal to <exp2>, exact match
!= or # or <>	<exp1> is not equal to <exp2>
<=	<exp1> is less than or equal to <exp2>
>=	<exp1> is greater than or equal to <exp2>
$	<expC1> is contained in <expC2>, i.e. is subset (substring) of <expC2>

An equality comparison by a NIL constant will return TRUE only if the second operand is an "empty" variable (or another NIL literal).

Numeric values are compared either on integer or floating basis. If both operands are integer, the comparison is precise on integer basis. If one of the operators is floating number, the comparison is performed on floating point basis which may be imprecise; it may fail even if mathematically correct. In such a case use the Round() or FieldIs*() functions, see example below and full details about IEEE floating point (with corresponding hints) in section LNG.2.6.4 (Numeric variables).

The comparison of **date** expressions is performed on the magnitude or on the underlying date value.

When comparing **character** (and memo) expressions, the comparison is based on the underlying ASCII code. Alphabetic characters in the PC-8 ASCII code, which is standard nowadays, have an ascending order (e.g., the code for "A" is 65, for "Z" is 90 and for "a" is 97).

> Note: the Unix environment variable LANG and/or the sorting table used by FS_SET("loadlang"..."setlang") may change the result of the < or > string comparison.

The SET EXACT command also affects string comparisons (except for the == operator):

- when EXACT is OFF (the default), two character strings are compared for equality according to the following rules:
  - if <expC2> is a null-string, return TRUE for =, >=, <=
  - if <expC1> is a null-string, return TRUE for <, <=, <>, #, !=
  - if LEN(<expC1>) < LEN(<expC2>), return FALSE.
  - compare all characters in <expC2> with <expC1>. If all chars are identical (for the = operator), return TRUE, otherwise FALSE.
- when EXACT is ON, two character strings must match exactly, except for trailing blanks, to be equal. See details in SET EXACT.

A comparison using the == operator is not affected by SET EXACT and return TRUE only, if all characters and both lengths are exactly the same.

For a true string equality comparison, use `a == b` or `!(a == b)` resp., since both are independent of the SET EXACT status. Note also that the `!(a == b)` syntax is not the same as `a != b` and therefore the results may differ if SET EXACT is OFF. The `!=`, `#` and `<>` operators are fully equivalent.

If the operators are typed variables (see 2.6.6) or constants, the validity of the expression is checked at compile-time and run-time.

Examples:

```
ok := 1 >= SQRT(55)                          // .F.

use mydbf                                    // field NUMBER = N,8,2
replace number with 10.01
? number == 10.01                            // most probably .F. (!)
? round(number, 2) == round(10.01, 2)        // .T.
? FieldIsEq(number, 10.01)                   // .T.

ok = "abc" $ "string contains abcdef"        // .T.

                                        //  SET EXACT OFF  SET EXACT ON
? "abc"      = "abcdef"                 //    false  .F.     false  .F.
? "abc"      = "abc   "                 //    false  .F.     true   .T.
? "abcde"    = "abc"                    //    true   .T.     false  .F.
? "abc  "    = "abc"                    //    true   .T.     true   .T.
? "abc"      = ""                       //    true   .T.     false  .F.
? ""         = "abc"                    //    false  .F.     false  .F.
? "abc"     == ""                       //    false  .F.     false  .F.
? "abc  "   == "abc"                    //    false  .F.     false  .F.
? "abc"      # "abcdef"                 //    true   .T.     true   .T.
? "abcde"    # "abc"                    //    false  .F.     true   .T.
? "abc"      # ""                       //    false  .F.     true   .T.
? !("abcde" == "abc")                   //    true   .T.     true   .T.
? "abc"      < "abcdef"                 //    true   .T.     true   .T.
? "abc"      < "abc   "                 //    true   .T.     false  .F.
? "abcde"    < "abc"                    //    false  .F.     false  .F.
? "abc  "    < "abc"                    //    false  .F.     false  .F.
```

```
? "abc"      < ""                    //    false  .F.     false  .F.
? ""         < "abc"                 //    true   .T.     true   .T.
? "abc"     <= "abcdef"              //    true   .T.     true   .T.
? "abc"     <= "abc   "              //    true   .T.     true   .T.
? "abcde"   <= "abc"                 //    true   .T.     false  .F.
? "abc  "   <= "abc"                 //    true   .T.     true   .T.
? "abc"     <= ""                    //    true   .T.     false  .F.
? ""        <= "abc"                 //    true   .T.     true   .T.
```

Logical Operators

Logical (boolean) expressions may be combined using the .AND., .OR. and .NOT. (or !) operator. All operators may be entered in lower or upper case.

Operation	By a long evaluation, when the -z compiler option is used, returns:
`.NOT. x -or- !x`	.T. if expression x is FALSE
`x .AND. y`	.T. if both expressions x and y are TRUE
`x .OR. y`	.T. if either expression x or y is TRUE

Operation	By a short, optimized evaluation (**default** w/o the -z compiler option and on run-time evaluation), returns:
`.NOT. x -or- !x`	.T. if expression x is FALSE, .F. otherwise
`x .AND. y`	result of y expr if x is TRUE, .F. otherwise
`x .OR. y`	result of y expr if x is FALSE, .T. otherwise

Examples:

```
LOCAL yes := .T., no := .F.
? yes .AND. no                    // .F.
? yes .AND. my_funct()            // depends on result of my_funct()
? no  .AND. my_funct()            // .F. (my_funct() is NOT entered)
? yes .OR.  my_funct()            // .T.
DO WHILE .NOT. EOF()              // same as: WHILE ! EOF()
   ? name, city
   SKIP
ENDDO
```

Character Concatenation

Character operators concatenate character strings. See chapter 2.6.5 for the usage of embedded chr(0) in strings.

Symbol	Description
+	Used to concatenate two character expressions, space characters included.
-	Used to concatenate two character expressions. Trailing spaces in the first expression are moved to the end of the resulting expression.
+=	identical to <varC1> := <varC1> + <expC2>
-=	identical to <varC1> := <varC1> - <expC2>

Example:

```
? "this is " + "my text "         // "this is my text "
? "this is " - "my text "         // "this ismy text  "
```

Operator Precedence

When evaluating expressions with two or more operations that are not explicitly grouped together with parentheses, FlagShip uses an established set of rules to determine the order in which the various operations are evaluated. These rules, called precedence rules, define the hierarchy of all of the FlagShip operators.

When more than one type of operator appears in an expression, all of the sub-expressions are evaluated for each precedence level before sub- expressions at the next level are evaluated. All function arguments are evaluated from left to right before the call itself is performed.

Except for multiple in-line assignments, all operations at each level are performed in order from left to right; multiple in-line assignments are performed from right to left.

Note that although the FlagShip language provides a specific order of precedence for evaluating expressions, it is better programming practice to explicitly group operations using parenthesis for readability, to be certain that what executes meets your expectations, and to remain compatible to other xBASE languages.

The order of precedence for the operators, by category from highest to lowest, is as follows:

1. Parentheses and special operators: the order of precedence for expression evaluation can be overridden using parentheses. When parentheses are present in an expression, all sub-expressions within parentheses are evaluated first using the precedence rules described in this section, if necessary. If the parentheses are nested, the evaluation is done starting with the innermost pair and proceeding outward. Special operators (like the

send operator `:`, square brackets `[ ]`, braces `{ }`, macro & and call-by-reference @ operator) have the next precedence level.

2. Pre-increment and pre-decrement: both operators in this category (++ and `--`) exist at the same precedence level and are performed in order from left to right.

3. Mathematical: when more than one mathematical operator appears in an expression, all of the sub-expressions are evaluated for each precedence level before sub-expressions at the next level are evaluated. All operations at each level are performed in order from left to right. The order of precedence for the mathematical operators is as follows:
   - unary positive and negative (+, -)
   - exponentiation (**, ^), right bounded
   - multiplication, division, and modulus (*, /, %)
   - addition and subtraction (+, -)

4. Relational: all of the relational operators exist at the same precedence level and are performed in order from left to right.

5. Logical: like the mathematical operators, the logical operators also have an established order of precedence. When more than one logical operator appears in an expression, all of the subexpressions are evaluated for each precedence level before subexpressions at the next level are evaluated. All operations at each level are performed in order from left to right. The order of precedence for the logical operators is as follows:
   - unary negate (.NOT. or !)
   - logical and (.AND.)
   - logical or (.OR.)

6. Assignment: all of the assignment operators exist at the same precedence level and are performed in order from right to left. For the compound operators, the non-assignment portion (e.g., addition or concatenation) of the operation is performed first, followed immediately by the assignment. The increment and decrement assignment operations exist at their own level of precedence, and are not part of assignment category.

7. Post-increment and post-decrement: both operators in this category (++ and `--`) exist at the same precedence level and are performed in order from left to right.

The non-assignment portion of the compound assignment operators (e.g., the final multiplication portion of *=) exists at level 3, and the assignment portion exists at level 6, e.g.:

```
a *= b + 2 * c++

is evaluated as

_temp = (2 * c)
c = c + 1
a = (a * (b + _temp))
```

2.10 Macros

The macro operator in FlagShip allows runtime evaluation (interpretation) of expressions and text substitution within strings. Whenever the macro (&) operator is encountered, the operand is submitted to a special runtime interpreter referred to as the macro evaluator that can process expressions, but not statements or commands. FlagShip supports three kinds of macros:

a. **Standard macro** and text substitution by the syntax:

`&<varC>[.]`	simple macro
`&<varC>.text`	composed macro
`text&<varC>[.text]`	composed macro
`"text &<varC>[.] text"`	text substitution

where <varC> is a character variable. The period (.) is the macro terminator and is used to indicate the end of the macro variable and to distinguish the macro variable from the adjacent text in the statement. Any space, TAB, comma or end-of-line character will also terminate the macro. In composed macro variables, the name is created at runtime by concatenating the literal <text> together with the result of the macro evaluation (see example).

b. **Compiled macro** expression using the syntax:

`&(<expC>)`

where <expC> is a character expression that must be enclosed in parentheses. In this instance, the expression is evaluated first, and the macro operation is performed on the resulting character value. This allows using contents of fields and array elements within macros or to evaluate stringified code blocks.

c. **Macro functions**

```
MacroEval(<expC>)
MacroSubst(<expC>)
```

where the first is a special case of compiled macro, see section FUN.

After macro expansion, the resulting expression has to be less than 4095 characters (Clipper: 254 chars) in length. Otherwise, a run-time error will be generated.

Since macro is an operator, it is accepted as a part of an expression or as assignment (i.e. it is ok at the right site of the statement or within a command - replacing it string value), but not stand alone (i.e. not ok as a command alone or at the left site of an assignment).

Content of the Macro-Variable

Macros **cannot** contain commands, command clauses or parts of commands (same as in Clipper, but opposite to interpreters like dBase or FoxPro), but may contain function calls. They may also be used for command or keyword arguments, as an alternative (older) syntax to the newer, parenthesized (and faster) option. Commas within macros are not allowed, except for the SET COLOR command, or as a parameter separator in UDFs. Example:

```
myselect = "25" ; color := "W+/B,N/W"
dbfname := idx1 := "address"; idx2 := "custom"
SELECT &myselect

USE &dbfname INDEX &idx1, &idx2   // ok, same as: USE (dbfname) ...
SET COLOR TO &color               // same as: SET COLOR TO (color)
cond1 = "x < 2.71"
cond2 = ".not. eof()"
cond3 = cond1 + " .AND. " + cond2
DO WHILE &cond1 .AND. &cond2      // or: DO WHILE &cond3
   x = x + 0.5
   skip
enddo

FOR ii = 1 to 274                 // create and access composed
   xxx := ltrim(str(ii,3))        // macro variables:
   myvar&xxx.a := ii * 10         // myvar1a...myvar274a created
NEXT
yyy := "var45a"; q1 := "my"; q2 := "var59"
? myvar20a, my&yyy, &q1.&q2.a     // 200, 450, 590

? &(INDEXKEY(1)), &(CHR(INKEY(0)))
myvar = "DTOC(DATE())"            // convert date to char,
? &myvar                          // since myvar has to be <varC>,
datvar = DATE()                   // or use the macro expression,
? &("DTOC(datvar)")               // like this
cdatvar = "DATE()"
? &("DTOC(" + cdatvar + ")")      // or this
REPLACE saveblk WITH "{ |exp| QOUT(exp) }"
blkvar := &(FIELD->saveblk)
:
EVAL (blkvar, DATE())
EVAL (blkvar, "mytext")
```

but

```
dbfname = "address"
cCmd = dbfname + ' NEW EXCL'    // !! dBase or FoxPro syntax,
USE &cCmd                       // macro is command part, cannot process

cCmd = 'USE &dbfname NEW EXCL'  // !! dBase or FoxPro syntax,
&cCmd                           // macro is a command, cannot process

cCmd = 'DbUseArea(.T., dbfname, , .T.)'            // ok
cCmd = 'DbUseArea(.T., "address", , .T.)'          // ok
cCmd = 'DbUseArea(.T., "' + dbfname + '", , .T.)'  // ok
lOk = &cmd                                         // this is ok
```

Type of Macro-Variable

LOCAL and STATIC variables are allowed only within simple macro variables <varC>, but not for composite or nested macro variables. Variables in composite and nested macros are always searched for in the PRIVATE and PUBLIC tables only. If no such variable exists, a run-time error will be generated. LOCAL and STATIC variables (e.g. <varC>) cannot be used if the stringified macro is passed to a functions, like QOUT("&varC"). On the other hand, the usage of QOUT(&varC) is o.k., because the argument evaluation is done prior to the function call.

On arrays, macros can be used with PRIVATE and PUBLIC arrays and array elements, e.g.:

```
arrname := "myarray"
maxelem := 10
elemnam := "myarray[5]"
PUBLIC &arrname.[maxelem]            // creates myarray[10]
&elemnam = DATE()                    // myarray[5] := DATE()
&arrname.[2] :="test"                // myarray[2] := "test"
&(myarray[2]) := "new value"         // test := "new value"
```

Nested Macros

Nested macros are allowed, FlagShip supports nesting macros in other macros to a depth of 256, e.g:

```
PRIVATE cvar := "first", cvar1 := "second"
PRIVATE avar := "&bvar", bvar := "c" + "var"
? &avar                              // nesting level 3, prints: first
? &bvar.1                            // evals cvar1,     prints: second
bvar += "1"; ? &avar                 // evals cvar1,     prints: second
```

Text-Substitution

FlagShip also supports macro text-substitution. This means, a macro is possible within string constants and will be evaluated when

- the macro operator & is immediately followed by character(s) without blanks, and
- the characters behind the macro operator may be evaluated as a valid and assigned PRIVATE or PUBLIC character variable.

In all other cases, the ampersand (&) is printed and no macro evaluation takes place. For example:

```
xyz = "string 2"
count = "1"
? "this is &xyz within the &count.st string"
```

will be printed as "this is string 2 within the 1st string", but:

```
? "High Tech & Co " + " and &undefined_var"
```

will be printed as "High Tech & Co and &undefined_var"

If you want to substitute macro in string variable by the same way as in the string constant, use the MacroSubst(cVar) function instead, see section FUN and examples there.

Macros in Code Blocks

If a macro is used within a code block, two different cases may occur: the standard macro is evaluated at the time of the block definition; the expression macro is evaluated later, at the time of the execution of the code block.

```
PRIVATE mvar := "test", test := 55, newname := "DATE()"
bvar1 := { || &mvar   }              // assigned as { || test }
bvar2 := { || &(mvar) }              // not expanded at defin.time
mvar = "&newname"
? EVAL(bvar1)                        // out: 55
? EVAL(bvar2)                        // out: 07/15/93
```

Linking Macro-invoked Function

When referencing user defined functions UDF or UDP (and some standard functions) in standard (stringified) macros but not elsewhere, declare EXTERNAL <udfname> in order for the linker to include them into the executable, in case the file containing <udfname> is not linked implicitly.

2.11 Objects and Classes

FlagShip fully supports Clipper's and VO's implementation of OOP (object oriented programming) classes. FlagShip also provides facilities for defining and manipulating **user defined** objects as a data type. The OOP syntax used here is compatible to CA-VisualObjects and is also in most parts applicable for CA-Clipper (and/or Classy). A detailed description of the commands and objects mentioned in this chapter is given in the manual sections CMD and OBJ.

In addition to providing the syntax and semantics for using your own classes, FlagShip 4.4 comes with several predefined system classes, each with a corresponding creator function

```
Get class,      -+  compatible to
Error class,     |  CA/Clipper 5.x
TBrowse class,   |  and FlagShip 4.3
TBColumn class  -+

DataServer class   superset of CA/VO DataServer
DBserver class     compatible to CA/VO DBserver

DbfIdx class       equivalent to DBserver
AsciRDD class      subset of DBserver, ASCII driver

CB4CDX class,   -+
CB4NTX class,    |  RDDs for CodeBase
CB4NDX class,    |  and Fox, Clipper, or dBASE
CB4MDX class    -+
```

2.11.1 Class and Object Definition

Objects in FlagShip are complex data structures with predefined instance variables and methods to access them. The object variable has some similarity to an array variable, whereby the object elements contain both data and code. The data element is named instance, and the code element is a method.

The **CLASS** statement declares a user-defined class name to the compiler and begins the definition of the data portion of the class. The **METHOD** statement declares executable code (function) tied to and encapsulated in the class. You can define classes as self-contained units or inherit instances (data) and methods (code) from another class, called a superclass. Such a derived class is called a subclass. Inheritance is accomplished using the **INHERIT** clause of the CLASS statement. Using inheritance, you establish a tree-like hierarchy among the classes you define.

To **create an object**, i.e. **instantiate a class**, you name the class followed by the instantiation operator { } or alternatively invoke the (by compiler automatically generated) creator function named classNameNEW ( ) :

```
<oVar> := <className> { }                    -or-
<oVar> := <className> {<argumentList>}       -or-
<oVar> := <className>New ( )                 -or-
<oVar> := <className>New (<argumentList>)
```

In FlagShip, you may also **prototype** a class, which declares the presence and structure of the class to the compiler. When you access an object element (instance) or invoke a method, the FlagShip compiler will optimize the object access at compile-time, which significantly speeds up execution. The compiler learns of the class structure by means of a previous CLASS declaration (in the same source file), or the PROTOTYPE CLASS and PROTOTYPE METHOD statements (which may also be given in an #include file, see example in 2.11.5) when the class is declared in another file. If the class is neither declared nor prototyped, the name resolution is performed at run-time (resulting in decreasing execution speed).

> **Hint:** To make your job as easy as possible, the FlagShip compiler (pass 2) automatically generates/modifies a prototype file named reposit.fh and places it in the current working directory. Every time a CLASS, METHOD, ACCESS or ASSIGN statement is encountered, the appropriate prototype is appended to this file. After pre-compilation, this "reposit.fh" file (or a file of your choice) may then be simply #include'd in your source files or globally in a local copy of the std.fh file. See also section FSC.1.1.3.

Static class: By default, classes declared with the CLASS statement have application-wide compile-time visibility (see also prototyping above). A STATIC CLASS is available only to the source file in which it is declared, and will not conflict with other same named STATIC classes in other files. Static class will not be included in the reposit.fh prototyping file by FlagShip. The METHODs of a static class are declared in the same way as for usual classes, but have to be declared in the same source file.

The class itself has no lifetime, since it serves as a declarator only. An instantiated class i.e. the created object has the same lifetime as the variable carrying it (Local, Public, Static etc., see 2.6.3)

2.11.2 Instances

Following the CLASS statement, you can declare **instance variables** by using the INSTANCE statement. Each instance variable has a defined place in the object structure (similar to an array element) and holds the internal object data. The name of the instance variable is used to access its contents at run-time by means of the "send" operator ":" or by using an Access or Assign method. During class declaration, only a place-holder and, optionally, the variable type is specified (in the same way as for local variables). At object creation time, the instance variables are initialized to the predefined default values (if specified), or are set to any value in the INIT() method. Otherwise, they are preset to NIL (or the default value for the type declared).

Some of the instance variables are used only internally in the class methods, and are not visible to the user. You may specify special object functions (Assign and/or Access methods)

to be able to handle these instances outside of the class. The visibility of the instance variables is defined during their declaration:

INSTANCE declares regular instance variables that are visible only for the class itself (e.g. in methods) and subclasses inheriting thereof. These instances are "late bound", you can override them with ACCESS and ASSIGN methods of the same name.

```
CLASS Personal                              // class name
INSTANCE Name := space(20)                  // initialized instance
INSTANCE Age AS IntVar                      // typed instance
```

PROTECT INSTANCE (or just PROTECT) declares instance variables with the same visibility as the regular instances. The difference is, that protected instances are "early bound" and cannot be overridden with ACCESS and ASSIGN methods of the same name. Example:

```
CLASS Personal
INSTANCE Name AS Char
PROTECT INSTANCE salary
PROTECT Children := 0 AS IntVar
```

HIDDEN INSTANCE (or just HIDDEN) declares instance variables that are visible only for the class itself (e.g. in its methods), but not in subclasses inheriting thereof. As with protected instances, they are early bound and cannot be overridden. Example:

```
CLASS MySubClass INHERIT Personal
HIDDEN INSTANCE WifesName := ""
```

EXPORT INSTANCE (or just EXPORT) declares instance variables that are visible also outside of the class, as externally accessible property of objects. The compiler awareness of exported variables is the same as of the class itself (see prototyping above), the run- time scope and visibility is the same as of the carrying object. Exported instances are early bound and cannot be overridden. Example:

```
CLASS Personal
EXPORT  INSTANCE Name := space(20)
EXPORT  INSTANCE Age AS IntVar
PROTECT INSTANCE salary := 0
```

2.11.3 Methods, Access, Assign

Methods are predefined functions (UDF) which perform actions on the object. They too are accessed by name via the send operator (see 2.11.5), and executed using the optionally given arguments. The corresponding object element does not contain the code itself, but only a pointer to the UDF. Therefore, an inherited class contains only the pointer to the same functional code. There is no code duplication or redundancy.

Methods are declared outside of the CLASS entity using the METHOD statement followed by their (UDF) code. The compiler automatically ties the method to the specified class. Methods are very similar to functions. They also may have parameters, declarations, programming

statements, and return values. The most significant difference is in how they are invoked (via send operator, see 2.11.5) and their visibility (same as the carrying object variable, see 2.6.3). Unlike UDFs, which return NIL per default, the METHOD returns its object (SELF). Example:

```
CLASS Personnel
PROTECT Salary                                  // invisible instance
...
METHOD GetSalary() CLASS Personnel              // method declaration
RETURN Salary                                   // = RETURN self:salary
```

Assign and Access are special cases of methods. They also are declared outside of the CLASS entity using the ASSIGN or ACCESS statement followed by the functional code, similar to METHODs or UDFs. They are used to refer to non-exported instances ensuring their encapsulation. Both Assign and Access methods therefore represent a "virtual instance variable", whereby the Access and/or Assign method name may be equivalent to the name of a regular instance variable or a regular method (but must differ from names of EXPORTed instances). Unlike with exported instances, you may specify either an access (read) or an assign (write) "instance" only, and may perform validation checking prior to replacing the instance.

ACCESS declares a method that is automatically executed each time you access an instance of the same name, using the

```
[<result> :=] <className>:<instName>
```

syntax.

ASSIGN declares a method that is automatically executed each time you assign a value to an instance of the same name, using the

```
<className>:<instName> := <value>
```

syntax. The <value> is passed as an argument to the assign method. Example for read-only access to personnel's name, first name, verified write-access to Update and read/write access to IDno:

```
CLASS personnel
  INSTANCE name                         // stores name, first
  EXPORT   idno                         // visible instance
  INSTANCE update

ACCESS name() CLASS personnel           // hides the instance access
  RETURN name                           // returns instance variable

ACCESS firstName() CLASS personnel      // virtual instance variable
  RETURN ltrim(substr(name, at(" ",name)))

ASSIGN update(date) CLASS personnel     // hides the instance
  update := if (valtype(date)=="D", date, date())
  RETURN                                // = return SELF
```

Init() method: if a method named INIT(..) is specified, it is called automatically during object creation. The parameters correspond to arguments given within the class instantiation operators { }. Common uses of the init() method are to initialize instance variables, allocate memory for special instances, create subsidiary objects and set relationships between objects. The init() method is generally designed **not** to be called manually, but from the object creator only. Per default, all instance variables are initialized to NIL (or the empty default values for the declared type), or the assigned values by the object creator function <className>New( ). The method must return the object SELF. See example in chapter 2.11.5. Init() is very similar to class initializer (creator) in C++

Axit() method: if a method named AXIT() is specified, it is called automatically by the run-time garbage collector, just before the object variable gets destroyed and the occupied memory freed. It occurs generally at the end of the object-variable lifetime (e.g. by encountering the RETURN statement when freeing and destroying all LOCAL variables). Common uses of the Axit() method are to reverse Init() method's doing, e.g. release relationships between objects, close assigned databases, de-allocate memory etc. Note that usual instance variables are de-allocated automatically; you will only need Axit() to free memory for very special instances, explicitly allocated e.g. with _xalloc() in Init(). The axit() method is generally designed **not** to be called manually, but automatically from the run-time system only. The method may return any value, usually logical or NIL, but not SELF. Axit() is very similar to class destructor in C++

NoMethod() method: to prevent a runtime error when a method's name is not found in the class, FlagShip's run-time automatically invokes the NOMETHOD (methName [,arguments...]), if present. Example:

```
METHOD NoMethod(cMethName,p1,p2,p3,p4,p5) CLASS MyClass
LOCAL param := "()"
if pcount() > 1
   param := "(par1" + if (pcount > 2, "..." +        ;
            "par" + ltrim(str(pcount()-1)), "") + ")"
endif
if upper(cMethName) == "ERRORMSG"
   alert ("Unknown method " + cMethName + param +   ;
          " in " + procname(1) + str(procline(1),4))
   quit
else
   self:errorMsg ("Unknown method " + cMethName + param +   ;
                  " in " + procname(1) + str(procline(1),4))
endif
return NIL
```

NoiVarGet() and **NoiVarPut()** methods: to prevent a runtime error when an instance variable's name is not found or is not visible, FlagShip's run- time automatically invokes the NOIVARGET (varName) or NOIVARPUT (varName, assgValue) methods, if present. This feature is very useful when creating virtual variables dynamically at runtime, e.g. accessing a field by name in the DBserver class. The method should return the real or default value of the given instance variable. Examples:

```
METHOD NoiVarGet(cVarName) CLASS MyDBserver
if USED() .and. FIELDPOS(cVarName) > 0
```

```
   return FIELDGET(cVarName)
endif
self:errorMsg ("Unknown field/instance " + cVarName +    ;
               " in " + procname(1) + str(procline(1),4))
return NIL
METHOD NoiVarPut(cVarName, value) CLASS MyDBserver
if USED() .and. FIELDPOS(cVarName) > 0
   self:rlock()
   self:fieldput(cVarName)
   self:unlock()
   RETURN .T.
endif
self:errorMsg ("Unknown field/instance " + cVarName + ;
               " in " + procname(1) + str(procline(1),4))
return .F.
```

2.11.4 Naming convention

Because the instances and methods are tied to the specified class, they will not conflict with the same names of different classes, nor with usual memory variables or regular UDFs. Instances and methods may also carry the same names in the same class - except for the access/assign methods, the name of which must differ from exported instances of the same class. You may choose any name according to the usual naming convention (see chapters 2.6 and 2.3), except for the reserved names for special methods INIT(), AXIT(), NOMETHOD(), NOIVARGET() and NOIVARPUT(). For compatibility purposes, names of instance variables and assign/access methods **are** abbreviated to 10 significant characters (see 2.6), names of methods **are not** abbreviated and are significant in the fully declared length (as opposed to the usual UDFs, see 2.3). The capitalization (upper/ lower case) is not significant.

2.11.5 Using Objects

Send operator: The ":" operator sends selector messages to or receives them from a specified object. Such messages access a variable or perform a special object action. The general syntax is

```
<object>:<selector> [ ([<argumentList>]) ]
```

SELF keyword: can only be used in methods and refers to the object itself, the object of which method is executing. Using the SELF: prefix with instance variables or access/assign method is optional. You may also use the SELF: prefix to distinguish between the instance variable and the same named local variable within the method, since the local variable hides the instance. The SELF: keyword may be abbreviated by "::".

SUPER keyword: can only be used in methods and refers to the class that is the nearest ancestor of the method's class. It is meaningful only if the current object class inherits from another class, otherwise the self:NOMETHOD() is invoked.

Access of an object's instance: A visible (exported) instance or an access method is addressed by using the object variable name (or the access method or the SELF/SUPER keyword while in method), a send operator and the instance/access name, syntactically <object>:<instance> or <object>:<accessMethod>, e.g.

```
LOCAL nColumn, oGet := GETNEW()
nColumn := oGet:col              // col is an exported instance
? oGet:pos                       // pos is an access method
```

Assign value to an object instance: A visible (exported) instance or an assign method is addressed by using the object variable name (or the assign method or the SELF/SUPER keyword while in method), send operator and the instance/assign name, syntactically <object>:<instance> or <object>:<assignMethod>, e.g.

```
LOCAL nColumn := 25, myget
myget := GETNEW()                // create object
myget:col := nColumn             // col is assign method or exp.inst.
myget:row := 2                   // row is assign method
myget:NAME := "testvar"          // assign an instance variable
actpos := myget:POS              // access an instance variable
myget:DISPLAY()                  // perform action (method)

var1 := "cargo", var2 := "right"
myget:&var1 := "my text"         // macro on instance var
myget:&var2()                    // macro with method
```

Invoking object methods: the object method is invoked by the object variable name, a send operator, the method name, parentheses and optional arguments, syntactically <object>:<method>( ) or <object>:<method>(<arguments>). When another method of the same object is invoked within a method, use the SELF keyword instead of object name, syntactically SELF: <method> ([<arguments>]). See examples below.

Availability and visibility: FlagShip checks the availability of the instance variable or method both at compile-time as well as at run-time. The compile-time check is possible only if the class is already declared in the same .prg module or if class prototyping is used. If the instance or method is unknown at compile-time, the slower run-time addressing is used. If it is unknown at run-time, and the NoIVarGet() or NoIVarPut() method is not declared, an error occurs.

For available instance variables and methods of predefined object classes, see the section OBJ.

Examples of class declaration, prototyping and usage (see additional examples in the section OBJ and REL):

```
*** file test1.prg ***

#include "test1.fh"                          // may be omitted here
CLASS Employee                               // see note a. below
   INSTANCE Name
   EXPORT   Phone AS usual
   HIDDEN   IdNo := 0 AS IntVar
```

```
   PROTECT  Salary AS Numeric

ACCESS Name CLASS Employee                        // hides Name inst.
   return Name

ASSIGN Name(cValue) CLASS Employee                // hides Name inst.
   if valtype(cValue) != "C" .or. empty(cValue)
      alert ("cannot assign Employee:Name with wrong value")
   else
      Name := cValue                              // replace instance
   endif
   RETURN Name                                    // Instance value

METHOD Init (nId, nSalary) CLASS Employee
   Name   := space(20)
   IdNo   := if (valtype(nId) == "N", nId, 999)
   Salary := if (valtype(nSalary) == "N", nSalary, 0)
   RETURN                                         // returns SELF

ACCESS FirstName CLASS Employee                   // virtual instance
   LOCAL name := SELF:Name                        // local vs. instance
   if !(" " $ name) .and. !("," $ name)           // First name available?
      return ""                                   //  no
   endif                                          // Otherwise, extract it
   name := strtran(name, ",", " ")                //  from Name,First,...
   name := ltrim(substr(name, at(" ", name)))     //  or   Name First ...
   name := if (" " $ name, substr(name, 1, at(" ", name)), name)
   RETURN alltrim(name)

METHOD Salaries(user, newValue) CLASS Employee
   if user # "Smith"                              // permission for
      return -1                                   // Smith only
   endif
   if valtype(newValue) == "N" .and. newValue > 0
      Salary := newValue                          // replace requested
   endif
   RETURN Salary

METHOD NameSalary (user) CLASS Employee
   LOCAL name                                     // local hides inst.
   name := SELF:name                              // local and instance
   if SELF:Salaries(user) > 0                     // invoke method
      name += " " + str (SELF:Salaries(user))     // store local
   endif
   RETURN name                                    // return local var

FUNCTION start()                                  // main entry
LOCAL personal[10]                                // array of objects
DO test2 WITH personal                            // call TEST2
QUIT
*** eof ***

*** file test2.prg ***
PROCEDURE test2 (personnel)
LOCAL user := space(20), input
LOCAL ii, idNo AS INT
```

```
LOCAL oEmpl AS Employee                          // typising the object
LOCAL getsys := {} AS GET                        // local, nested GET
#include "test1.fh"                              // class prototypes

for ii := 2 to len(personnel)                    // Creates objects,
   personnel [ii] := Employee {ii, ii * 100}     //  invoking also the
next                                             //  init() method
personnel [1] := EmployeeNew (1, 100)            // Alternative syntax

@ 10,10 say "Your name  " GET user
READ
while lastkey() # 27
   @ 12,10 say "Employee ID" GET idNo RANGE 1,len(personnel)
   READ
   @ 13,10 CLEAR TO 16,60
   oEmpl := personnel [idNo]
   @ 13,10 say "Name  : " + oEmpl:Name           // access
   @ 14,10 say "First : " + oEmpl:FirstName      // access
   @ 15,10 say "Phone : " + oEmpl:Phone          // export.instance
   @ 16,10 say "Salary: "
   ?? personnel[idNo]:salaries(user)             // method
   if empty(personnel[idNo]:Name)                // access
      input := space(20)
      @ 13,10 say "Name  :" GET input VALID !empty(input)
      READ
      personnel[idNo]:Name := input              // assign
   endif
enddo
RETURN
*** eof ***

*** file test1.fh, see also note (a) below ***
PROTOTYPE CLASS Employee
   INSTANCE  Name
   EXPORT    Phone  AS usual
   HIDDEN    IdNo   AS Intvar
   PROTECT   Salary AS DOUBLE
PROTOTYPE ACCESS Name CLASS Employee
PROTOTYPE ASSIGN Name(cValue) CLASS Employee
PROTOTYPE ACCESS FirstName CLASS Employee
PROTOTYPE METHOD Init (nId, nSalar) CLASS Employee
PROTOTYPE METHOD Salaries(user, newValue) CLASS Employee
PROTOTYPE METHOD NameSalary (user) CLASS Employee
*** eof ***

*** Compile: FlagShip test*.prg -na -m -w4 -Mstart
```

Notes for the above example:

a. During compilation of test1.prg, FlagShip automatically creates an #include file named "reposit.fh" containing the prototypes of the classes declared there (see also FSC.1.4.3). So you may also use this file by simply renaming it to "test1.fh" (or specify the -r=test1.fh switch) instead of creating it manually.
b. If the #include "test1.fh" statement (including the prototypes) is omitted in test2.prg, the application will also run fine but slower, since the object entities (instances and methods)

have to be determined at run-time by searching the internal name tables for every access. See more in chapter 2.11.6 below.

c. If the file test2.prg were a part of test1.prg (or if the class declaration would be in the same source file), the prototyping would not be necessary.

See also section RDD.2 and the *<FlagShip_dir>/system/smallrdd/smallrdd.prg* file for an example of a small RDD driver, or the files in *.../system/ ascirdd* or *.../system/cb4rdd* directories for a large C program.

2.11.6 Performance Hints

There are two different ways how the objects and their entities are accessed and addressed:

a. **Late binding**: the (address of each) object entity is searched by the FlagShip run-time system during execution. This is the default rule, when the object/class structure (prototype) is unknown for the FlagShip compiler. If the search fails, the NoiVarGet, NoiVarPut or NoMethod class method is invoked, if such exists. If not so, a run- time error appears. This process is comparable to invoking a UDF or accessing a variable by macro, and is therefore significantly slower than the compile-time address resolution.

b. **Early binding**: the entity addresses are already resolved at FlagShip compile-time, which is similar to resolving UDF or variable addresses. It allows much (3 to 5 times) faster access than the run-time searching. The requirement is, that the compiler is aware of the class structure (assigned to an object) latest when encountering the entity, which is provided by the programmer by:

   - giving the variable a type of the class, e.g. LOCAL oDbf AS MyServer, and
   - announcing the object structure to the compiler via prototyping (e.g. #include "MyServer.fh"); but is not necessary if the class was previously declared in the same file. See also example in chapter 2.11.5 above, in section REL, and the PROTOTYPE statement.

   Note: for proper early binding, all properties in the header (.fh) file **must** match with the declaration of instances, methods and access/assign methods within the .prg file with CLASS declaration. You may create or check the/your .fh file by using -rc -r=... compiler switches, e.g. "FlagShip -m -c -rc -r=myclass.fh myclass.prg" which will create the myclass.fh file from class declarations available in myclass.prg

The above rules apply for all objects, the standard classes (e.g. GET, TBROWSE, DBSERVER etc.) as well as for user defined classes. The standard prototypes are declared in files named getclass.fh, errclass.fh, tbrclass.fh and dataserv.fh. The DBFIDX class is specified in the dbfidx.fh file, all of them summarized in the "stdclass.fh" file.

To take advantages of the early binding, you may #include the prototype file(s) in your .prg sources, or #include "stdclass.fh" in the (local copy of the) std.fh file. Using the -w4 compiler switch, you may verify the early vs. the late binding.

Note, that giving the variable a class type (e.g. LOCAL ... AS ...) does **not** instantiate the class, but informs the compiler (or the run-time system) about the class, which is later assigned to this variable.

2.11.7 Converting Class(y) syntax

If your source uses Classy add-on library for user defined objects in Clipper, you will need to make few changes only:

Declaration Class(y)		Declaration FlagShip and VO
CREATE CLASS <name>	->	CLASS <name>
EXPORT : VAR <instance>	->	EXPORT <instance>
ENDCLASS	->	n/a (remove)
method declarations	->	PROTOTYPE <method> CLASS <class>
METHOD <class>:<method>	->	METHOD <method> CLASS <class>

and instantiate your object

```
obj := <class>:new()        ->   obj := <class>New(....)
                            or   obj := <class>{...}
```

If your source is instead VO based, no syntax change is required, since FlagShip use the same semantic and syntax, and corresponds to Clipper 5.x instantiation of standard classes.

3. Files

In both Unix and MS-Windows, files are referenced by names. For the naming convention, see chapter 3.1. FlagShip uses the following file types for input and output:

Database Files

A database file has the ".dbf" extension as default. It contains database records, preceded by a header with record structure definitions. The complete file structure is fully compatible with that of Clipper and other Xbase dialects such as dBASE III plus. For more information see chapter 4.

Memo Fields Files

have a fixed ".dbt" extension (in the default RDD driver) which cannot be changed. They encapsulate the contents of all memo fields from the corresponding database file. Their structure is fully compatible with that of Clipper or other Xbase dialects.

The ".dbv" extension is used for field types V, VC, VCZ, VB, VBC, see more in FUN.DbCreate()

FoxPro's ".fpt" files are supported as well. See chapter 4. FlagShip supports also variable memo fields with the ".dbv" extension.

Index Files

An index file contains keys and pointers to the corresponding records in the .dbf file. The default extension depends on the used RDD driver, standard is ".idx". Neither Clipper ".ntx" nor dBASE ".ndx", ".mdx" or FoxPro ".idx", ".cdx" file structure are supported by the default "DBFIDX" driver, available in the included CB4* database drivers, see section RDD.

For program portability and to perform existence checking of the index file, you should use the standard INDEXEXT() function, since it returns ".NTX" for Clipper and ".idx" for FlagShip. Using the FlagShip global setting FS_SET("translext", "NTX", "idx") will convert all file names from .ntx to .idx automatically. For more information see chapter 4.

Memory Files

Memory files contain memory variables and arrays stored by the SAVE command. These variables can be read from the memory file by means of the RESTORE command. FlagShip supports storing & restoring of arrays as well. The name, type, length and contents of a memory variable or an array element are preserved. The default file extension is ".mem". Clipper and dBASE ".mem" file structure is supported, for compatibility see also FS_SET("memcompat").

Report Files

Report files, default extension ".frm", contain the necessary information for the REPORT FORM and REPORT EDIT command to produce report forms. The structure of the file is fully compatible with that of the Clipper/xBASE ".frm" file.

Label Files

Label files, default extension ".lbl", contain the necessary information for the LABEL FORM and LABEL EDIT command to produce (address) labels. The structure of the file is fully compatible with that of the Clipper/xBASE ".lbl" file.

Program Files

FlagShip programs are plain ASCII files with a ".prg" extension. They can contain any number statements and user defined procedures or functions. Names specified in SET PROCEDURE (and SET FORMAT) commands are actually program file names. For the Extend C system, and to storing the intermediate C code, files with the extension ".c" are used. See also LNG.2.3 and LNG.8.

Include Files

Include files are similar to a standard .prg files. Since they are invoked by the preprocessor statement `#incude "filename.ext"`, any extension may be used. The default extension for FlagShip include files is ".fh". Usually, the include files contains often used preprocessor directives, command translations, definitions, etc. For more information, see section PRE.

Format Files for READ

Although files with an ".fmt" extension are in the interpreted xBASE dialects intended for screen format definitions, any valid FlagShip command can be included. A .fmt file is actually program file and will be compiled as such.

Printer Files

Printer output from FlagShip is always done through a printer file to facilitate Unix/Windows multitasking printouts and spooling. This file will usually be printed using the "lp" command in Unix. Direct output to a specified device, like /dev/tty01 (or COM2: in Windows) is possible but discouraged. See also LNG.3.4. In GUI mode with PrintGui(.T.) active, the printer output goes directly to selected printer (driver) at request via PrintGui() or oPrinter:GUIexec().

Other Files

Output files created by various commands (for example: COPY TO file SDF, TEXT TO file, SET ALTERNATE TO file, etc.) are plain ASCII files with a default ".txt" extension. You may access (read, write) any file using the low-level file system (see LNG.3.5), or some of them also via the ASCIRDD driver.

3.1 File Names in Unix vs. DOS

Unix file naming conventions allow any ASCII character to be used in a file name. The maximum number of characters comprising depend on the Unix implementation, but is at least 14, usually 255. To abbreviate long Unix names to the DOS convention, FS_SET("shortname") may be used. Both the path and file name are case sensitive, but file names given in your .prg source can be in Linux automatically transformed to lower or upper case by FS_SET("lower" or "upper") and/or FS_SET("pathlower" or "pathupper") as already pre-defined in the "fspreset.fh" (see below).

In Unix, there is no such concept as extension part of file name or MS-DOS drive selector (like C:). A dot or colon is a "normal" part of file names like any other character. Any number of dots can be in a file name. The same is valid for embedded space or special characters in the file name. You may use "ls -lb *" to display file names including special characters. Note: FlagShip support emulation of the MS-DOS drive selector via environment variable x_FSDRIVE, see LNG.3.2 for details.

When accessing or creating standard files, FlagShip supports the file naming concept of MS-DOS to retain compatibility with other xBASE programs. So, wherever an extension can be assumed, it need not be explicitly specified. However, unconventional extensions can be used as well. A file extension starts with a dot and contains up to three additional characters.

This may cause some confusion for the Unix/Linux programmer, but a file name without extension cannot be specified for the standard FlagShip input and output (but is supported in the low-level file system). For example, using the command `USE TEST`, the file "`TEST.dbf`" will be assumed and looked for, and `USE test` will look for "`test.dbf`". Using the command `USE TEST.XYZWQ` the "`TEST.XYZWQ`" file will be expected; but in `USE test.` the file `test.` instead of the MS-DOS file `test` or `TEST` (without a dot) is assumed. This is due to differences in the operating systems and cannot be avoided.

Please note that Unix **distinguishes** between upper and lower case letters, thus "`Test`" and "`test`" are different files. However, FlagShip will automatically **convert** (during the file opening process) all upper case letters in a file name to lower case when FS_SET ("lower") is used, or to upper case, when FS_SET ("upper") is used. The statement

```
#include "fspreset.fh"
```

already includes FS_SET("lower"), see details in LNG.9 The same is available for path names by using FS_SET ("pathupper") or FS_SET ("pathlower")

The **MS-Windows port** of FlagShip uses standard Windows naming convention. The maximum length of file name with drive and path is 255 characters, neither the path nor file name are case sensitive. Conversion to upper or lower case by FS_SET() is accepted as well but the access remain unaffected since Windows names are case insensitive; it is however suggested if you plan later port to Unix or Linux.

3.2 Directory and File Access

Unix, like MS-DOS and Windows, supports a tree-like structure of directories. Anywhere where a file name can be specified, a path name can be specified too. Subdirectories are separated by a slash "/" character. The maximum length of a path name in FlagShip is 250 characters.

FlagShip accepts paths in both Unix and DOS/Windows syntax, i.e. using a slash "/" or backslash "\". During file access, FlagShip for Unix/Linux will automatically replace any backslash character (MS-DOS or Windows path separator) to a slash character (i.e. "\" to "/"). FlagShip for MS-Windows will replace slash character by backslash (i.e. "/" to "\").

> **Note:** You may need to avoid using backslash "\" and file name starting by three numbers, since this may be interpreted as special character (\011 is chr(9), see LNG.2.7). For example, `USE \aa\01123xy` will be interpreted as USE "\aa<tab>23xyz", so the file will not be found. You may safe use slashes e.g. `USE /aa/01123xyz` will work fine in Linux and Windows.

Path names, like file names, are case sensitive on Unix. You may choose to convert all paths given in the program automatically to lower or upper case during the file access by using FS_SET("pathlower") or FS_SET("pathupper") switches. You alternatively may insert the command

```
#include "fspreset.fh"
```

at the begin of your main, which already includes FS_SET("pathlower"), see details in LNG.9. It is accepted also in MS-Windows but irrelevant, since Windows file and path names are case insensitive.

As already mentioned, Unix has no equivalence for the MS-DOS drive selector (e.g. A:, C: etc.). Separate disks (and sometimes also floppies) are "mounted" onto the main tree structure. There also exist utilities to access DOS format floppies or partitions, like doscp, dosls, etc. See Unix "man dos".

Nevertheless, FlagShip provides drive letter support by the environment variable x_FSDRIVE, where x is the required upper case drive selector. If such an environment variable is set to a path, the drive selector, colon included, in the given path name will be substituted by the specified Unix path during file access in FlagShip. (See example in LNG.9.5.)

FlagShip includes standard commands and functions to access the directory tree or to determine and manipulate the files available there:

Command / Function	Description
COPY FILE TO	duplicate a file
ERASE / DELETE FILE	delete file
RENAME ... TO	rename file
FILE()	check if file is available
CURDIR()	reports current working directory
DISKSPACE()	available disk space
MEMOREAD(), MEMOWRITE()	read/write text file
RUN DIR	print the directory using Windows command

RUN ("ls -l *")	print the directory using Unix command
ADIR(), DIRECTORY()	determine files & attributes
USE, INDEX... etc.	database & index access, see LNG.4
FOPEN(), FCREATE(), FCLOSE()	low-level open/closing a file
FSEEK()	low-level repositioning of a file
FREAD(), FREADSTR(), FWRITE()	low-level read from, write to file

Many of these commands and functions also support the Unix wildcard convention, i.e. *, ?, [ ] as well as Windows wildcard convention * and ?.

When using ADIR() or DIRECTORY(), keep in mind the difference in access rights on DOS / Windows vs. Unix (see LNG.3.3 and section FUN).

3.3 Access Rights

The Unix system supports a high level of data security using "access permissions" as standard. To access files from the FlagShip application, you should note:

1. Each file (and directory) has an access right for the file owner, associated group and other users. The actual rights may be listed with "ls -la", for example:

```
$ ls -la *.dbf *.DBF *out

-rwxr-x--x  1  john       programm  456789  Aug 15 17:05 a.out
drwxr-x--x  2  peter      programm      12  Feb 10 10:51 mydir
-rw-rw-r--  1  hugomayer  personal  298765  Jul 20 15:10 adress.dbf
-rw-------  1  dummy      programm     127  Jan  8  9:15 Adress.DBF
-rw-rw----  1  peter      personal    1239  Jun 24 10:08 person.dbf
| └─┘ └─┘ └─┘     └──────┘ └──────┘
| |   |   |           |          └──────── name of the assoc. group
| |   |   |           └─────────────────── name of the file owner
| |   |   └─────────────────────────────── access: others
| |   └─────────────────────────────────── access: group
| └─────────────────────────────────────── access: owner
└───────────────────────────────────────── directory (or other special) mark
```

   The access rights may be changed by the file owner or by the "superuser" (supervisor) or "root" using the Unix commands "chmod" or "chown". You may also change these from a running application using e.g. RUN chmod 0660 person.dbf (here: change access to -rw-rw---- rights) provided that you are the owner of the file.

2. FlagShip will open a standard file (e.g. database or index) successful only if this file has at least "rw" access rights for

   others : access for all users allowed
   group : access for the same group member only
   owner : access limited for the file owner only

   Using the READONLY clause in USE command, the database file must have at least the "r" access right. For index files, "rw" is required regardless the USE clause.

   The directory with your databases and indexes must have at least "rwx" access rights.

3. The index and .dbt (or .dbv) files should have at least the same rights as the associated database.

4. When a new file is created by FlagShip (for example INDEX ON ..., COPY TO ...), the parent/child heredity principle will be used: the new "child" file (index, database) inherits the rights of the actual database ("parent").

5. If a new file is created without a "parent" (e.g. CREATE ..., FOPEN(), PRINT TO... etc.), the new file obtains the standard access rights; your actual "creation mask" (umask) is used. The application may change these permission rights thereafter e.g. RUN ("chmod 0660 newdbf.dbf")

6. Be careful when using the (not very common) "mandatory locking" right. The RLOCK() or FLOCK() functions will then lock the whole database also for read access from other users, until UNLOCK is executed. Therefore, on execution of the USE command, a warning in the "developer" mode occur (see FS_SET()).

In **MS-Windows**, files (and paths) may have "system", "hidden" and "read-only" permissions, which behaves similarly. The permissions are managed by ATTRIB command or via properties/security options.

3.4 Printer Output

In GUI mode, you may print to any available printer (also GDI) device connected by parallel, serial, USB or (W)LAN interface, or shared over network. This is very similar to common CUPS in Linux or Winspool in Windows. To do so, simply invoke **PrintGui(.T.)** to start printer buffering, optionally parallel to screen output. With **PrintGui()** w/o parameter, you will start the printer output. See further details in section FUN.PrintGui() and CMD.SET GUIPRINTER

An alternative printer output, available in any i/o mode (GUI, Terminal or basic) is available via SET PRINTER command:

In Unix or Windows server, there is usually more than one user logged in at the same time, or the logged-in user may execute several tasks parallelly, same as in current Windows versions. FlagShip therefore automatically supports the "spooled" output of Unix or Windows to any printer to avoid garbage being output by the printer. Each output following SET PRINTER ON will be redirected to a special spooler file. The Unix printout must be done later using e.g.

```
$ lp -dlaser  <filename>             --or--
$ cp  <filename>     /dev/lp1        --or--
$ cat <filename>  >  /dev/lp0
```

or directly from the application by "RUN lp -dlaser <filename>" etc. This method of output is similar to the MS-DOS printer output via "PRINT <filename>".

In MS-Windows, you may use standard "COPY <filename> PRN" command to print the spool file, or use printer object to access Windows driver, or specify e.g. SET PRINTER TO LPT1

FlagShip creates the name of the output file automatically:

```
<main program name>.<process id>
```

e.g. "address.123" or "xyz.5647". Within the application, you may determine this file name using the function FS_SET("printfile"), see also FS_SET() and SET PRINTER examples.

Normally, the spool file is created in the current directory. To create it in another directory, set the environment variable FSOUTPUT, e.g.

```
$ FSOUTPUT=/usr/spool ; export FSOUTPUT
```

Direct output to any device is also possible using the command SET PRINTER TO <device>, where <device> is any valid device, like /dev/lp0, /dev/tty02 etc. in Unix & Linux, and PRN or LPT3 etc. in MS-Windows.

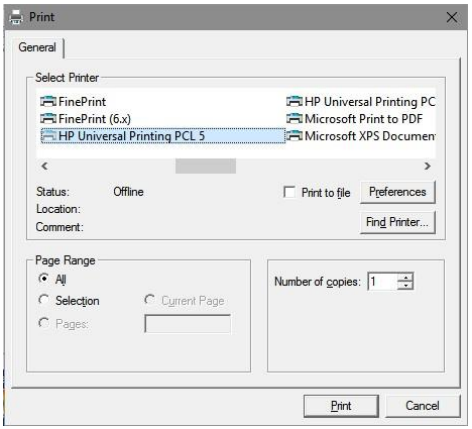
You may tune the printer driver by FS_SET("prset"). Note that some printers requires CR + LF (= carriage return + line feed) for line break instead of LF (line feed) used by default. In such a case add the statement

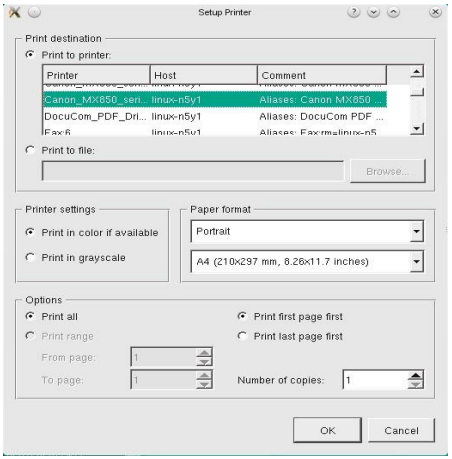
```
FS_SET("prset", { chr(13)+chr(10) } )
```

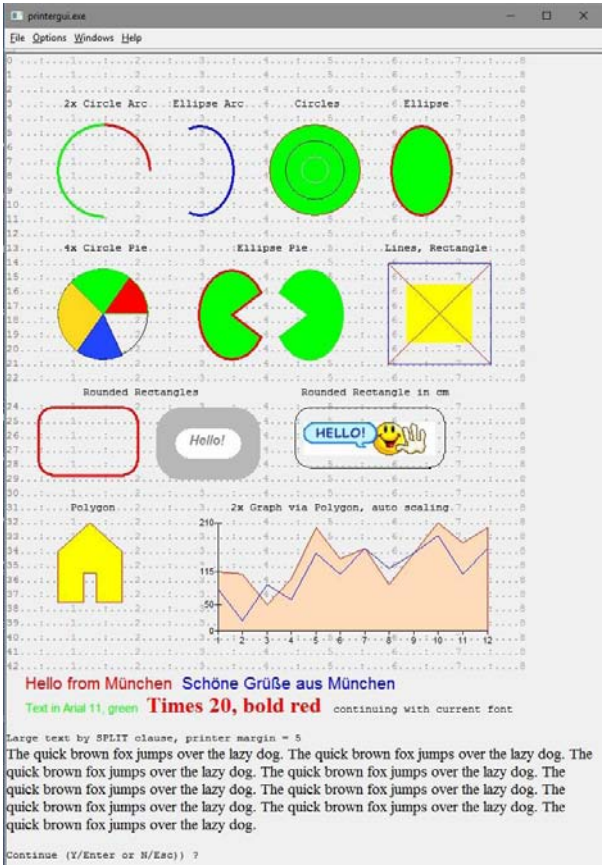
before your printer output statements.

In GUI, the printer output may be done upon user's request nearly automatically via Menu->Print selection or programatically by the Printer class, available via already instantiated oPrinter object. See also LNG.5.1.6 and section OBJ.Printer for further details.

See examples in *<FlagShip_dir>/examples/printergui.prg* and *printer.prg*







3.5 Low-Level File System

FlagShip also supports direct access and manipulation of text and binary files. Because of the low-level access, such usage requires at least basic knowledge of the Unix (or MS-Windows) file system, i/o low level handling and the structure of the processed file.

The low-level file system bypasses the FlagShip settings for the high- level routines like SET PATH and SET DEFAULT, but the extended settings and conversions using FS_SET("lower", or "pathlower"...) are considered.

Like in the programming language C, low-level functions are available in FlagShip:

- Open a file: before use, a file has to be opened by an FOPEN() or FCREATE() function. The input arguments are the file name and an optional access right. On success, the function returns a "file handle", which is a number used in subsequent functions.
- Read a file: an open file will be sequentially read by FREAD(), FREADSTR() and FREADTXT(). Note: binary zero CHR(0) bytes are supported; for buffer manipulation using standard string functions refer to chapter LNG.2.6.5.
- Write a file: a predefined string or character buffer will be written into the opened file using the FWRITE() function. Writing binary zero CHR(0) bytes from the buffer are supported.
- Position to a new file offset: the file byte pointer is positioned to the first (0) byte in a newly opened file, and past the last read/written byte on subsequent operations. Use FSEEK() to re-position the current byte offset within the file.
- Close a file: the low-level opened file will be closed using FCLOSE() or automatically on program end, break or termination.
- Check whether the operation has been successful should be performed using the FERROR() function.

FREAD() and FWRITE() can also be used to process data on **serial port** (tty device device on Unix). Complete serial port access is available in the FS2 library.

Keep in mind that the database locking mechanisms; FLOCK(), RLOCK() etc. cannot be used for this low-level system. Should locking be required, the special low level locking using FLOCKF() may be used instead.

3.6 Large File Support

In the most operating systems (also in Linux and MS-Windows), FlagShip supports access to and manages files larger than 2 Gigabytes (up to petabytes). It is enabled by default, i.e. SET LARGEFILE is ON. This is compatible also to available DOS databases up to approx. 1.5 GB. To achieve backward compatibility to available databases below 2GB use SET LARGEFILE OFF at the begin of your application, latest before open and access the data(base).

4. The Database System

The FlagShip Database Management System (DBMS) is a collection of commands and functions to handle databases, fields and indexes. It takes care of keeping binary compatibility to other xBASE systems like Clipper, dBASE, FoxBase etc. The DBMS supports the full handling and program control of concurrent network, multiuser and multitasking access using standard Unix or MS-Windows locking mechanisms. Commands to export or import other ASCII files into databases are available.

Because all the database (and index) handling functions are object oriented and encapsulated within the RDD (replaceable database driver), you may freely exchange the default, or add another "database engines" of your choice (see sections OBJ and RDD for details).

4.1 Databases

The database file (also called a "database table") consists of sequential records of fixed length. For storing strings of variable length, additional memo-field-file(s) may be specified in the database structure. New records are always added at the end of the file. Records cannot be removed from the file, rather they will be marked for deletion and the whole file can then be "purged" with the PACK or ZAP command.

A record consists of fields of fixed length (except for data of memo- fields, since stored in separate file). The length of a record is equal to the total length of all its fields plus one (the deleted flag). There are no record or field delimiters. The length of the fields (and thus the length of the record) is defined at database file creation time. The database is practically a two-dimensional table with fixed width (fields) and variable length (records). All data in records is in ASCII format. Database (and index) size exceeding 2 GB is supported by SET LARGEFILE ON, available per default, for backward Clipper and FoxPro compatibility use SET LARGEFILE OFF, see also chapter 3.6.

At the beginning of the .dbf file is the header (in binary form). It contains information about last update date, number of records, header size, etc.

The total length of a database file can be calculated as:

```
dbf_len = 33 + no_of_fields * 32 + no_of_records * record_len
```

where

```
record_len = sum (field_lengths) + 1
```

See also example in FUN.DBSTRUCT().

Selected Database Commands and Functions

Database access	
USE .. [index] [exclusive] [alias]	open database (and indices)
USE	close actual database
CLEAR ALL, CLOSE ALL	close all open databases
CLOSE, CLOSE DATABASES	close all/the actual database
USED()	is the database opened ?
DBF()	retrieve the database name
SELECT [scope] [alias]	select working area 1...255
SELECT 0, USE...NEW	select next free working area
SELECT()	get the number of actual working area
ALIAS()	get the actual alias or dbf name
SET LARGEFILE ON	enables support for files > 2GB
SET RELATION to ... into	relate two or more databases
DBRELATION()	retrieve the actual relation
DBRSELECT()	retrieve the child relation
APPEND ... from [sdf] [delimited]	add data from ASCII or database file
COPY TO... [sdf] [delimited]	copy database to dbf or an ASCII file
COPY STRUCTURE to...	create an empty database
COPY STRUCTURE to... extend	create filled structure database
DBCREATE()	create a database from an array
CREATE	create a structure database
CREATE ... FROM	create a database from a structure .dbf
JOIN with ... TO ... for [fields]	join two databases into a new one
SORT ... on .. to	sort database
UPDATE on...from...[replace]	update actual database from another
LABEL FORM [for] [while] [to]	label output from a .LBL file
REPORT FORM [for] [while] [to]	report output from .FRM file
HEADER()	get the size of a .dbf header
LUPDATE()	get the last date of a .dbf update

Access to database records	
SKIP [range]	relative record movement
GO, GOTO [record] [top] [bottom]	absolute record movement
APPEND BLANK	append an empty record
DELETE [for] [while]	mark record as deleted
RECALL [for] [while]	unmark record as deleted
PACK	remove all deleted records
ZAP	remove all records from database
AFIELDS()	fill array with info about dbf struct.
DELETED()	is the record deleted ?
BOF()	attempt to pass the database top ?

EOF()	end of database passed ?
LASTREC(), RECCOUNT()	database record count
RECNO()	current database record number
RECSIZE()	database record size

Access to database fields

var = DBFfield	assign field to memory variable
var = alias->DBFfield	assign field of an aliased database to var
var = DBFfield + 30	database field within an expression
fn (DBFfield, var)	database field as argument
FIELDGET(), FIELDGETARR()	get specified field or all fields in record
FIELDPUT(), FIELDPUTARR()	replace specif.field or all fields in record
REPLACE [range] field .. WITH	replace contents of database field
@ ... GET / READ	change database field from user entry
AVERAGE ... TO	average of numeric fields
COUNT ... TO	count records fulfilling condition
SUM ... TO	sums a range of numeric fields
TOTAL ON ... [fields] TO	sums specified fields into a new dbf
DBEDIT()	screen oriented display of dbf fields
FIELD(), FIELDNAME()	get the name of a database field
FIELDLEN(), FIELDDECI()	determine the field size
DISPLAY	display database field(s)
LIST	list database

Search, filtering

SEEK, FIND	fast index oriented search for data
SEEK EVAL	search index for any data
SET SOFTSEEK on/off	toggles relative seek on/off
FOUND()	was the searched for record found ?
LOCATE for [while]	locate data sequentially
CONTINUE	continue the previous locate search
SET FILTER to	only the specified data is visible

Management of memo fields

charVar = MemoField	assign a memo field to a string variable
REPLACE field WITH charVar	change the contents of a memo field
MEMOEDIT()	screen oriented output/editing of memo
MEMOLINE()	get a line from a memo field
MEMOTRAN()	change a soft to a hard CR/LF wrapping
MLCOUNT()	count the lines in memo field
MLPOS()	seek for some text in a memo field
MEMOREAD()	read in a text from a Unix/Windows file
MEMOWRIT()	write the text to a Unix/Windows file

Index management	
INDEX ON	create a new index file, sorted on a key
INDEX ON...FOR...	create a new index file of selected keys
REINDEX	recreate all index files for the current dbf
SET INDEX TO	assign an index file(s) to the current dbf
SET ORDER TO	set specified index as main sort criterion
SET UNIQUE off/on	turn on/off writing multiple record keys
DESCEND()	reverse sorting order
INDEXEXT()	get the extension of the index file (.idx)
INDEXDBF()	report the associated dbf file
INDEXKEY()	report curr. sorting criterion of index file
INDEXORD()	report the current main index
INDEXCOUNT()	report the no. of current assigned indices
INDEXNAMES()	create array containing act. index names
INDEXCHECK()	check the integrity of index to dbf

Network, multiuser, multitasking	
SET EXCLUSIVE on/off	open databases exclusive/sharable
SET AUTOLOCK on/off	allow automatic record/file locking
SET COMMIT ...	performance tuning for flushing data
SET MULTILOCKS on/off	allow multiple record locking
USE ... [exclusive/share]	open act.database exclusive/sharable
RLOCK(), FLOCK(), AUTOxLOCK()	record or file locking for write access
UNLOCK [ALL], DBUNLOCK()	free record or file locking
COMMIT, DBCOMMITALL()	flush all buffers onto disk
SKIP 0, DBCOMMIT()	flush current database onto disk
NETERR()	check success of dbf opening/appending

Creating a Database

A new, empty database can be created on-line by the application using the function DBCREATE() or commands CREATE and CREATE FROM. The associated .dbt (or .dbv) file is created automatically, if a MEMO or variable MEMO field(s) is/are used. Example:

```
LOCAL dbfstru := {{"NAME",     "C", 25, 0}, ;
                  {"FIRST",    "C", 20, 0}, ;
                  {"ADDRESS", "C", 30, 0}, ;
                  {"BORN",     "D",  8, 0}, ;
                  {"EARN",     "N",  8, 2}, ;
                  {"NOTES",    "M", 10, 0}, ;
                  {"IMAGES",   "VB",10, 0}  }
DBCREATE ("employee", dbfstru)
USE employee
? RECCOUNT(), FCOUNT()               // 0  6
```

For more examples see (CMD) CREATE, CREATE FROM, COPY STRUCTURE, COPY STRUCTURE EXTENDED and (FUN) DBCREATE.

4.2 Database Records and Fields

The **field** is the atom of the database file. It is always of fixed length which determined at creation time. Its length cannot be changed (rather, a new file with the new structure has to be created and then data copied from the old file). There are five types of fields. Regardless of type, their content is always ASCII and is binary compatible to Clipper or xBASE databases on DOS.

Field names: All database fields are named. The field name can be up to 10 characters long, must begin with an alphabetic character. The remaining characters can also be numbers or the underscore ("_") sign. No embedded blanks are allowed. Access to the field name is done by name, optionally prefixed by an alias (for further details see chapter LNG.4.4).

Delete flag (1 byte) holds the information, if the record was deleted by DELETE, i.e. should the record be invisible when SET DELETED is ON.

Character field (C): Maximum size of a character field is 65535 characters. It can contain any combination of letters. In fact, a single character can have any value from 0 to 255.

Numeric field (N): Maximum number of digits is 19 including decimal point and minus sign. The number is represented with ASCII characters. The actual precision of the number in numerical computations is 15 decimal places.

Date field (D): always takes 8 bytes to store. The full date, including century is stored in ASCII format without delimiters.

Logical field (L): takes up 1 byte. Its content can be one of " ", "F", "f", "T", "t", where T or t represents TRUE, everything else a FALSE value.

Memo field (M): A memo field is used to store text of variable length. Space is allocated in 512 byte blocks. Memo field declarator in .dbf file is always 10 bytes long. It contains (in ASCII format) a number of starting sector (of 512 bytes) in the associated .**DBT** (memo) file where the memo text begins. If the memo text exceeds 512 bytes, it continues in next sector and so on. FlagShip supports also .**FPT** fields, compatible to FoxPro. These behaves same as .DBT but have segments of 64 bytes (modifiable, see DbCreate() function). The maximal storage size of one memo field per record is limited by xBase specification to 64 KB. If you need to store strings larger than 64kb, use two or more memo fields (or .DBV Variable Memo filed), see example in CMD.REPLACE. The memo field string may contain any ASCII character value 1..255 but not CHR(0) = 0x00 and CHR(26) = 0x1A characters, since these terminates the memo field. But you may save it via MemoEncode() and read it later with MemoDecode() or use the CharPack() and CharUnpack() functions from FS2 Toolbox.

Variable Memo fields (V*) are FlagShip unique feature. The data are stored in separate file with a .**DBV** extension, similarly to Memo fields. But as opposite to usual Memo fields, these variable Memo fields may contain any character (0..255) and are not limited in size (up to 2 GB are supported for ea record). You therefore may store either character or binary data like binary images here. In addition to, FlagShip automatically compress the data by using LZH or RLL compression algorithm, if specified and applicable.

Record Order

All database records are internally numbered from 1 to LASTREC(), the actual record number can be determined with RECNO(). New records are always added on at the end of the database. Moving the record pointer with the SKIP command will access the physically next record (1 to 2 then 3 etc.); moving it by GOTO will directly access the required record number. To locate a defined field value, a sequential search using LOCATE and CONTINUE is available.

When an index file is used, the "visible" record order changes according to the sorting criteria. The physical record numbering remains however unchanged, but the SKIP command moves from the record containing "A" to the record containing the next index criteria "B" etc. so the physical movement on indexed database is e.g. to record 5,2,7,1 etc. Very fast searching commands SEEK and FIND are available.

Accessing the Database Records and Fields

The usage of database fields is very similar to the usage of memory variables, after the record pointer is positioned on the required database record (using SKIP, GOTO, SEEK, LOCATE etc.).

The contents of the database field (of the currently positioned record) may be stored (assigned) in a variable, used directly in expressions, or passed as argument to standard functions, UDF or UDP.

The command APPEND BLANK is used to append a new, empty record to the selected database. To replace the field contents of current record with a new value, use the command REPLACE, FIELDPUTxxx() functions or the := assignment.

The command DELETE will mark the whole record as "deleted". Such a record will be "invisible" if the switch SET DELETED is ON. To un-delete such a record, use the command RECALL. With PACK you can irreversibly remove all "deleted" records from the database.

To avoid ambiguity between field usage and PRIVATE or PUBLIC variables with the same name, the alias-> (or area-> or FIELD->) selector or the FIELD declaration will specify the name explicitly as a database field. To prefer a PRIVATE or PUBLIC variable, use the MEMVAR-> or M-> selector or the MEMVAR declaration. For further details refer to LNG.4.4.

If both the selector and declaration are omitted, the FIELD is preferred on access from, but a memory variable is used on assignments to. LOCAL or STATIC variables always have preference to the fields with the same name when the alias-> selector is not used.

Accessing the Memo Field

Memo fields store variable length character data in the associated file, named the same as the database, but with a .DBT (or .FPT) extension, see 4.2. The handling of memo fields is the same as of character fields. To store the contents of a memo field in a variable, use the = or := operator, to replace a memo field with a new value, use the REPLACE command, FELDPUT() function or an aliased assignment with the = or := operator.

The memo field string may contain any ASCII character value 1..255 but not CHR(0) and CHR(26) characters, since these terminates the memo field. If these characters are used in the saved data, use MemoEncode() to store such strings in the memo field and MemoDecode() to read it from.

The memo field is of variable size (in 512 bytes segments) and is per xBase definition limited to 64 kBytes (65536 bytes). If you need to store strings larger than 64kb, use two or more memo fields to store it. On REPLACE, split the content via substr() to segments shorter than 64kb; on access simply concatenate these memo fields. For example:

```
cLongStr := replicate("x", 102400)                  // 100 kb
REPLACE FIELD->MEMO1 with LEFT  (cLongStr, 65000)
REPLACE FIELD->MEMO2 with SUBSTR(cLongStr, 65001)
...
cLongStr := FIELD->MEMO1 + FIELD->MEMO2
? len(cLongStr)                                     // 102400
```

Accessing the Variable Memo Field

Variable Memo fields are similar to usual Memo fields offering additional features. They store variable length character and binary data (BLOB) in the associated file, named same as the database but with .DBV extension, see 4.2. These variable Memo fields may contain any character (0..255) and are not limited in size (up to 2 Gigabytes for each record are supported). You therefore may store either character or binary data like binary images here. In addition to, FlagShip may automatically compress the data by using LZH or RLL compression algorithm, but only if the resulted assigned size is smaller than original. On access, the reverse decompression is done automatically too.

The handling of variable memo fields is the same as of character fields or usual Memo fields. To access or store the contents of a variable memo field in a variable, use the = or := operator. To replace the variable memo field with a new value, use the REPLACE command, FELDPUT() function or an aliased assignment with the = or := operator.

You may even use variable character fields in index, e.g. INDEX ON padr(VARFLD, 50) where VARFLD is the variable memo field. Note that you always will need to adapt the result to fixed index size, as shown in this example. Keep in mind that the access may be slightly slower than to regular char fields.

See an example in the <FlagShip_dir>/examples/images.prg file for further use.

4.3 Working Areas

Each opened database is associated to a selected working area. The working area contains information about the database name, the area number, actually opened accompanying index files, actual filter criteria and so on. FlagShip supports up to 65000 simultaneously opened databases, each associated with up to 15 index files and each with up to 8 relations to other databases.

> **Note:** In Linux, the physical amount of opened and/or shareable files depends on the actual setting of the Unix kernel. For further details see the Unix system administrator documentation and "man" pages for sysadmsh, system and so on (OS dependent). Check "`cat /proc/sys/fs/file-max`" and increase it by e.g. "`echo 5000 > /proc/sys/fs/file-max`".
>
> In MS-Windows, FlagShip supports up to **2000 open files** per application simultaneously.

Working areas are numbered 1 to 65534. To select one, the command SELECT is available. SELECT 0 is reserved for selecting a new, unused working area. On the program start, working area 1 is pre-selected.

To open a new database,

a. Select any unused working area, using e.g. SELECT 5 or SELECT 0 for automatic selection. This is only needed when the NEW clause in the following step is omitted.
b. Issue the command USE dbfname, optionally with the clause NEW and/or other clauses, specifying the alias name, associated indices etc. (see more command USE).

If the same alias is already defined in another working area, a run-time warning will occur. Opening a database in a working area will automatically close any previously existing one along with all its indexes and relations.

For an overview of commands and functions associated to working areas, databases and indexes, see sections QRF, CMD and FUN.

Concurrent Database Access

The same physical database can in FlagShip be simultaneously used in different working areas. For such special cases, a different ALIAS is required. The handling of concurrently opened databases within one application is nearly the same as sharing databases in multiuser/multitasking mode (see 4.8.7): SHARED mode is required, RLOCK() or FLOCK() have to be executed before write access (or SET AUTOLOCK enabled for the automatic locking). Some database global operations (like PACK, ZAP) are not allowed on concurrently open databases.

When the developer mode is on, using FS_SET("devel", .T.), you will get developer's warning at opening the same database twice within the same application. This avoids unintentional concurrent database use, resulting often in programmer's confusion.

You may check concurrently (multiple) open databases using IsDbMultiple() function. To determine whether the same database is used by another user or process or executable, use UsersDbf()

4.4 Aliases

When opening a database file for use, an alias can be specified. For example:

```
USE address ALIAS adr NEW
USE billing NEW
```

will open database files "address.dbf" and "billing.dbf" for use in different working areas, which will be referenced by the "adr" or "billing" aliases as in:

```
SELECT adr
SELECT billing
```

or the field access

```
cur_name = adr->name
bil_name = billing->name
? "Bill #", billing->number, "for Mr." + adr->name
```

Using the `alias->` operator allows database fields to be accessed or to perform expressions on otherwise unselected work areas. The last example also illustrates the use of aliases in expressions.

A user-defined or standard function or any other expression can also be executed as an aliased expression by preceding it with an alias and enclosing it in parentheses (see chapter 2.3.2), like

```
adr->(myskip(2))
? billing->(EOF())
```

Special Aliases:

In addition to standard ALIASes generated by the USE command, special alias selectors are available:

- Field variable: `FIELD->` or `_FIELD->` prefix specifies that the given database field must be addressed. This must always be done when REPLACEing the field contents in the `=` or `:=` operator. The equivalent alternative is using the FIELD declaration statement.
- Working area numbers: the number of the working area may be used as alias if enclosed in parentheses, e.g. `(5)->fnam` for the field "fnam" used in working area 5.
- Working area letters: for compatibility to dBASE, FlagShip supports the working area letter A,B,...,L which correspond to the working areas 1,2,...12. The syntax `A->fname` is equivalent to `(1)->fname`, L->fname to (12)->fname etc.
- Memory variable: a `M->` or `MEMVAR->` prefix specifies that the given PRIVATE or PUBLIC memory variable must be addressed. This must always be done when variable has to be made to point to a database field on access. An equivalent alternative is using the MEMVAR declaration statement.

Examples:

```
LOCAL   city
PRIVATE name
SELECT 3
USE address ALIAS adr              // contains field "name"
name          := name              // ─┐
name          := FIELD->name       //  │ all the statements
M->name       := name              //  │ are equivalent
MEMVAR->name := FIELD->name        //  │
MEMVAR->name := adr->name          // ─┘

SELECT 2
USE custom
MEMVAR->name := adr->name          // ─┐
MEMVAR->name := (3)->name          //  │ name from address
MEMVAR->name := C->name            // ─┘
MEMVAR->name := name               // ─┐
name          := custom->name      //  │ name from custom
M->name       := B->(UPPER(name))  // ─┘
adr->name     := custom->name      // REPLACE address.dbf
FIELD->name   := (3)->name         // REPLACE custom.dbf
custom->name := M->name            // REPLACE with PRIVATE

city          := adr->city         // LOCAL and STATIC
custom->city := city               // without MEMVAR->
```

4.5 Indices, Sorting

The physical order of database records (see 4.2) can be changed according to the required sorting criteria, like ascending for name, descending for the zip code etc., using the SORT command.

Indexing the database using INDEX ON or REINDEX is similar to SORTing it, but does not changes the physical order of its records. It creates a separate index file instead, containing the required sorting criteria. Each database may be associated with up to 15 different index files.

Using a filtered index (created with the FOR or WHILE clause) will result in significant speed increase compared to SET FILTER.

A new index is created using the command INDEX ON <exp> TO <file>. The maximum length of the expression statement <exp> is 420 bytes including white spaces, and of the evaluated index key 238 bytes. The sorting expression <exp> is stored in the index header and is retrievable by using INDEXKEY(). Once created indices may be re-indexed using the REINDEX command.

To assign one or more index files to a database, use the command USE...INDEX <file,file> or SET INDEX TO <file,file>. The first index specified becomes the controlling sorting criteria and controls the visible (logical) record order. The controlling index can be changed any time using SET ORDER TO, without re-opening the index files. To determine the actual order, use INDEXORD(). The indices are closed automatically when closing the associated database, or explicitly with CLOSE INDEX or SET INDEX TO.

The main advantage of indices compared to SORTing is, that many different sorting criteria are available, the index files are automatically updated on changes of the database and a very fast searching SEEK (or FIND) command is available to finding the required data. Also, a "soft" searching for the "exact or the next closest item" is available using the SET SOFTSEEK switch. FlagShip supports additionally a "scan for any value in index" using the SEEK EVAL command.

The results of searching by LOCATE, SEEK and FIND can be determined with FOUND() or EOF() functions.

Index Integrity Checking

Since the associated index file represents the "register" of the database, and all database movements refer to the indices, the index file must always be up-to-date and must match the database.

If one or more index files are assigned to the opened database using SET INDEX TO ... or USE...INDEX... commands, or the associated functions respectively, FlagShip will update all of

these indices automatically on any database modification. Using the command SET ORDER TO, the controlling index may be switched or disabled.

Thus, by assigning **all** relating indices to the opened database while modifying the database (i.e. changing the field contents, appending new records, deleting/un-deleting, PACK-ing and ZAP-ing), the integrity is preserved.

On the other hand, when the database gets changed, expanded or packed without the corresponding index attached, and the index assigned later, the field "contents" stored in the index file will not reflect the database contents - the index integrity will be corrupted.

FlagShip checks the **integrity** of indices by comparing an internal "modification" counter (stored in the index header) of the index with the same counter in the database. When they are different, only the REINDEX, PACK and ZAP commands may follow the index assignment. Otherwise, the first movement in the database will cause a warning (in developer mode, if the record count is equal) or a run-time error. This avoids fatal breaks in the middle of transaction, known in Clipper or FoxPro. The application can check the index integrity itself using the INDEXCHECK() function and perform silently re-indexing.

However, this index integrity checking disables the automatic movement to the database top, so when you are using SET FILTER or SET DELETED ON, you may need to issue GO TOP or DbGoTop() after open the database or assigning new indices. You may force the GO TOP movement automatically by SET GOTOP ON or Set(_SET_GOTOP,.T.) - which will behave same as Clipper but disables the possibility of silent, programmed index recovery.

The most common cause of corrupted index integrity is:

1. Database was previously open without index (or with not all indices) and modified by

   REPLACE <field> WITH <value>
   <field> := <value>
   FieldPut(), FieldPutArr()
   @..GET <field> / READ
   APPEND BLANK, DbAppend()
   Browse(), TbrowseDb(), Tbrowse()
   PACK
   ZAP
   APPEND FROM...
   UPDATE ON..FROM...

2. Database is multiply open (see LNG.4.3) and modified (same as 1), where the second work area did not open all required indices
3. Database was modified by another FlagShip application, which has not open all required/used indices
4. Database structure (it key field) was modified, but not re-indexed

5. Index key contain memory variable(s) which are differently set, or uses different relations. This generally should be avoided.
6. Database and index was changed, but not COMMITed yet to be visible by concurrently running application.
7. Database was modified by another, non-FlagShip application, which does not support FlagShip indices
8. Database was modified by another RDD driver or directly by own C (or Java or other low-level etc.) function

FlagShip will detect problems in (1) to (5) and report it by INDEXCHECK(), but understandably cannot detect automatically problems caused by (6..8). In such a case, run-time error (usually RTE 331) occurs, when corrupted index key is detected. See also example in INDEXCHECK(). Corrupted index file will be fixed by REIDEX or INDEX ON..TO..

4.6 Searching, Filtering

In FlagShip, there are three different possibilities to locate the required data in the database:

- Sequential searching, using the commands LOCATE and CONTINUE, steps the database record-by-record and compares the requested data given by LOCATE with the actual field contents. If the data is found, the searching stops and the function FOUND() returns TRUE, EOF() returns FALSE. The CONTINUE command continues the search for next applicable data. If the data is not found, EOF() returns TRUE and FOUND() return FALSE. The sequential search may be used for both indexed or un-indexed databases.
- Index-sequential searching, using the commands SEEK or FIND, is much faster than the LOCATE (CONTINUE) command, especially on large databases. It is usable on indexed databases only. To check the success, use FOUND() and EOF(). Only the first applicable database record will be found (if any), more records with the same index criteria may follow and are directly found by SKIP. FlagShip also supports "closest similar" index searching using the SET SOFTSEEK switch. See more in (CMD) SEEK, SKIP, INDEX etc. To build a FILTER criteria into index key, use INDEX ON...FOR|WHILE...
- Index-sequential index scanning, using the command SEEK EVAL, is similar to the sequential searching but is performed directly on the index key, and is therefore significantly faster than the LOCATE command. Using SEEK EVAL, you may scan for any value in the index key (also for a substring), or continue the previous search. It is usable on indexed databases only. To check the success, use FOUND() and EOF(). See more in (CMD) SEEK EVAL, INDEX etc.

Both searching methods, SEEK or LOCATE, may also be used to determine whether or not an item exists.

When SET FILTER is set, or SET DELETED is ON, the database seems to contain only the filtered records. Other records, which do not match the filter criteria, are automatically skipped over. The filter criteria affects LOCATE, SEEK and SEEK EVAL as well.

Using SET FILTER TO <exp> and GOTO TOP is similar to the LOCATE command. The usage of FILTER slows database access significantly, because all the filtered-out records must nevertheless internally be read.

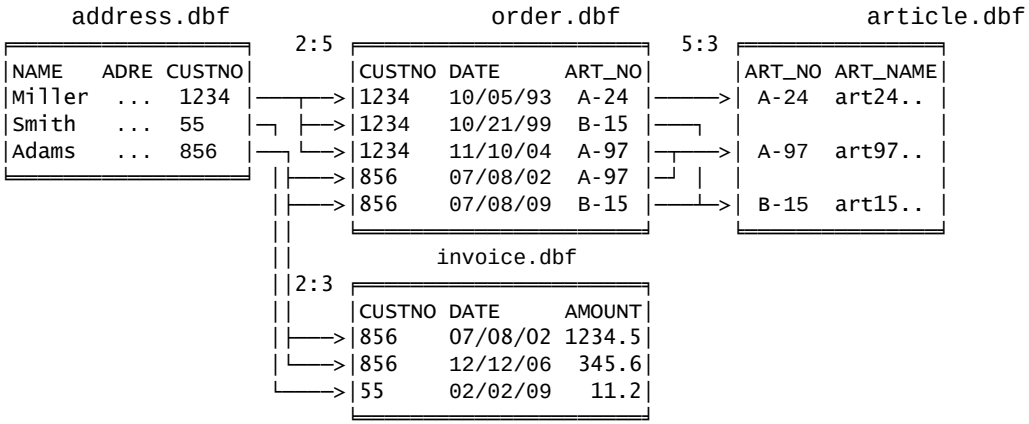
On indexed databases, SEEK and DO WHILE <filter_exp> is a better alternative. The fastest method is the usual invocation of SEEK (or FIND, SEEK EVAL) commands on "filtered" indices created using the FOR clause of INDEX ON.

4.7 Relations

When designing a database system, you create several separate database files (tables) to avoid redundancy. In a relational xBASE system, the tables (databases) may be connected together or refer to each other in any required order. There are no restrictions or prerequisites well known in other, hierarchical DBMS.

Relations are also called "joins" in database theory. A relation is a link between two database files on a key field that they both contain. The primary file is called the "parent", the secondary files are "children".

It is common, to create e.g. the parent customer address database, with relations to separate databases (children) of invoices, orders, stock, prices etc. After the databases are linked, one customer record may refer to one or more invoices, orders or price records. Moving the database pointer in the parent file moves it to records with the same key expression in each child.



In the above example, the address.dbf is the parent for order.dbf and invoice.dbf, the order.dbf is a child of address.dbf and the parent of article.dbf and so on. As you may see, there are relations 1:0, 1:n, and n:1 possible. The identical keys for the links are CUSTNO and ART_NO. The relation links are set with the SET RELATION command, e.g.

```
USE address  NEW SHARED
USE order    NEW SHARED INDEX order_cust
USE invoice  NEW SHARED INDEX invoice_cust
USE article  NEW SHARED INDEX article_no
SELECT address
SET RELATION TO custno  INTO  order    [ MULTIPLE ]
SET RELATION TO custno  INTO  invoice  ADDITIVE
SELECT order
SET RELATION TO art_no  INTO  article  [ MULTIPLE ]
SELECT address
GO TOP
```

```
WHILE !eof()
   ? address->name, address->custno, order->date, order->art_no, ;
     article->art_name, invoice->date, invoice->amount
   SKIP
ENDDO
```

With SET RELATION, FlagShip supports 1:1 relations per default. That means, SKIP on address (here Miller) locates the corresponding record in order.dbf (here CUSTNO=1234) and in article.dbf (here ART_NO=A-24). You may then select the related database(s), and skip thru for subsequent data. The next skip on address.dbf (here Smith) locates his related data in invoice.dbf, and so forth.

For your convenience, FlagShip also supports `1:N:N` relations by using the MULTIPLE clause in SET RELATION command (similar to SET SKIP in FoxPro). In this case, GO TOP or SKIP on address.dbf locates the corresponding record in order.dbf and in article.dbf (Miller -> 1234 10/05/93 -> A-24 art24..), same as with 1:1 relation. But subsequent SKIP on address.dbf will process all 1:n related databases first, i.e. it checks here for subsequent data in article.dbf and since not available, skips in order.dbf and returns (Miller -> 1234 10/21/99 B-15 -> none). Next SKIP returns (Miller -> 1234 11/10/04 -> A-97 art97..) and so forth, as long as the same relation key(s) matches for the master. Thereafter, next record in address.dbf (master) is selected. Note that with 1:n:n relations, you will need (in worst case) to skip (records-in-parent * records-in-child1 * records-in-child2)-times to reach eof(). See complete example in *<FlagShip_dir>/examples/relat_one2n.prg*

Although links set by SET RELATION are comfortable and easy to handle, they may **slow** program execution. Any time the address.dbf is skipped, all relations in child links also have to be moved. If the link is required in some special cases only (like a user request via hot-key), the "soft" link is a faster option: the child record is simply positioned by SEEKing the parent key. For further details see in (CMD) SET RELATION and SEEK.

4.8 Multiuser, Multitasking

FlagShip supports networking as well as multitasking and multiuser programming and file access.

Multiuser means that one, two or more users can execute their programs at the same time. The same user may also run the same or different application in different tasks (multitasking). Users are connected to the Unix or Windows server using dumb or intelligent terminals via serial port, Ethernet and so on. The terminal is often a PC computer running a terminal emulation program such as Procomm, Telix, Windows Telnet, Putty, PC/NFT etc.

Because multitasking/multiuser mode is comparable to networking in MS-DOS, the FlagShip language uses **the same** statements for your convenience, to support networking and multitasking/multiuser. So if your application is ready for network in Clipper or xBASE, no changes are necessary to run it as a multiuser/multitasking program with FlagShip, independent on the used target platform.

In addition to user programmable locking, FlagShip supports also **automatic** (but modifiable) record and file locking. Therefore, also application written for single-user execution on MS-DOS are directly usable in FlagShip and multiuser/multitasking environment. The only pre-requirement is to SET EXCLUSIVE OFF or open the databases via USE...SHARED.

The following network and multiuser/multitasking commands and functions are available for a manual locking control:

- Exclusive or shareable database access with SET EXCLUSIVE ON/OFF, USE ... EXCLUSIVE / SHARED
- File locking with FLOCK () or AUTOFLOCK()
- Record locking with RLOCK () or AUTORLOCK()
- Unlocking with UNLOCK, DBUNLOCK() or AUTOUNLOCK()
- Checking the success with NETERR()
- Flushing the buffers with COMMIT or DBCOMMIT()

Additional functionality is available by invoking the RDD methods, see sections OBJ and RDD.

The **AutoLock** functions AUTORLOCK(), AUTOFLOCK() and AUTOUNLOCK() are activated automatically when a lock is required, SET AUTOLOCK is ON (the default) and a manual lock was not performed.

The programming rules for running applications in multiuser and multitasking mode with FlagShip are the same (or even more simple), as for networking with Clipper or other xBASE dialects:

1. Put the statement **SET EXCLUSIVE OFF** in front of your main module. All databases (and indices) that will be opened thereafter, may be shared from different applications or users. An alternative is the **SHARED** clause in each by USE opened database which should/must be shared (applies also for AutoLock).

2. After opening the database and indices, **check** the status and success using the NETERR() function. To perform this check, don't open database and index(es) within the same statement (USE dbfname INDEX indexname), but use two statements (USE .../SET INDEX TO ...) instead (applies also for AutoLock). Example:

```
USE dbfname SHARED
DO WHILE NETERR ()
   ? "waiting for opening"
   INKEY (2)
   USE dbfname SHARED
ENDDO
SET INDEX TO index1, index2
IF NETERR()
   ? "wrong indices or permission !"
   QUIT
ENDIF
```

3a. The database must **exclusively** by opened for global changes of the whole database or index with:

- ZAP
- PACK
- REINDEX, DBREINDEX()

using the statement USE dbfname EXCLUSIVE or the usual USE while being in the SET EXCLUSIVE ON mode. This exclusive status will be successful only if the database isn't in use by other applications (applies also for AutoLock).

3b. The database should by FLOCKed or, better, opened exclusively, when the same index is/may also be used by other users/processes during the execution of

- INDEX ON...TO..
- DBCREATEINDEX(), ORDCREATE()

In **AutoLock** mode, AUTOFLOCK() is invoked automatically, if required.

The only exception of this rule is, when you create a temporary index, not used by other user/process. In such a case, you may avoid the FLOCK check (or the invocation of AUTOFLOCK) by using the NOLOCK clause of INDEX ON. Example:

```
SET EXCLUSIVE OFF                          && see 4.8.1
IF !FILE ("adr1" + indexext())             && see 9.3
   USE adress EXCLUSIVE                    && excl. modus
   DO WHILE NETERR ()                      && success ?
      ? "waiting for opening"              && - no,
      INKEY (2)                            && - wait and
      USE adress EXCLUSIVE                 && - try again
   ENDDO
   INDEX ON STR(zip,6) + name TO adr1
   INDEX ON DESCEND(orderno)  TO adr2
   USE
ENDIF
USE adress                                 && share modus
DO WHILE NETERR ()
```

```
   ? "waiting for opening"
   INKEY (2)
   USE adress
ENDDO
SET INDEX TO adr1, adr2
```

After INDEX ON, the index file remains in an EXCLUSIVE mode until the creator process closes it. The index file may also be locked "exclusively" by the user, issuing the SET INDEX ... EXCLUSIVE command.

4a. Each write access on .dbf or .dbt files with

- REPLACE
- FIELDPUT(), FIELDPUTARR()
- FIELD->dbffield := exp (assignment)
- GET / READ (on database fields)

- DELETE, RECALL
- UPDATE
- EVAL FIELDBLOCK() and FIELDWBLOCK() with parameters

must be locked using record or file locking and unlocked thereafter. You may use either record (RLOCK) or file locking (FLOCK) to update only one record, you should use file locking FLOCK to update multiple records, such as REPLACE ALL ... and so on. After the write access is completed, you should unlock the record or file (free it) using the UNLOCK command or the equivalent DBUNLOCK() function. To commit the changes to disk, use COMMIT or DbCommit(), preferably before UNLOCK.

If the **AutoLock** feature is not disabled, and the record or file is not locked by the programmer, FlagShip automatically invokes the AUTOxLOCK() function before the write access and the AUTOUNLOCK() function thereafter.

```
** Example for programmable locking:
select 5
@ 1,2 say "Name"    get NAME            && = dbf field
@ 1,2 say "Surname" get FIRSTNAME       && = dbf field
do while !RLOCK()                       && wait until the record
   sleepms(100)                         &&   locking is ok, and
enddo                                   &&   sleeps 0.1 sec if not
READ                                    && process the @..GETs
UNLOCK                                  && free the lock
:
:
do while .not. FLOCK()                  && wait until the
enddo                                   &&   file locking is ok
replace next 10 NUMBER with 5
recall all
UNLOCK                                  && free it
```

4b. The lock and unlock is effective on the **selected** database only. So if you change more then one database, you must lock each separately. If you use relations, you must also lock the corresponding database to make changes there. The alias selector (see

LNG.4.4) may be used to lock other than the actual working area. All locked databases may be freed together using UNLOCK ALL.

The **AutoLock** feature will do the lock and unlock automatically, if required, also on related databases.

```
SET EXCLUSIVE OFF
:
SELECT 10
USE adress ALIAS adress
... NETERR()   check
SET INDEX TO adr_name
** SELECT 20
USE custom NEW
... NETERR()   check
SET INDEX TO cust_name
:
SELECT adress
SET RELATION TO name INTO custom
:
SELECT adress
DO WHILE ! RLOCK()                      && lock ADRESS database
ENDDO
DO WHILE ! custom->(RLOCK())            && lock CUSTOM, WA 20
ENDDO
REPLACE adress->name WITH "Smith"
REPLACE custom->name WITH adress->name
COMMIT
UNLOCK ALL                              && free all locks
```

4c. The statement **APPEND BLANK** is an exception. Because FlagShip does not allow locking non-existing records, the new record will be automatically locked by APPEND BLANK or DBAPPEND(). You must unlock this record yourself using the UNLOCK statement thereafter; otherwise the lock remain active until the next APPEND BLANK, RLOCK(), FLOCK(), or AUTOxLOCK() is executed. When the multiple record locking is enabled (via SET MULTILOCKS), only UNLOCK [ALL], DBUNLOCK() or FLOCK() will release this lock. Therefore, it is always save to use COMMIT and UNLOCK after replacing data of the newly added record.

It is recommended to check the success of the append statement with the NETERR() function:

```
SELECT adress
APPEND BLANK
DO WHILE NETERR()                       && try again
   APPEND BLANK                         && until success
ENDDO
REPLACE ... with ...
COMMIT
UNLOCK
```

4d. All records, also from locked (or AutoLocked) databases, may be read by the same or other program, with the exception of databases opened in exclusive mode. Exclusively opened database can be read by the "owner" program only.

4e. Record and file locking (or AutoLocking) and exclusive opening also becomes automatically effective on .dbt memo files. The index files will be locked by FlagShip automatically only for the short moment during updating (exception the INDEX ON command, see 3.b above).

5. FlagShip stores the actual .dbf record in internal working area buffers. Three mechanisms will **flush** these internal buffers to Unix (Windows) and/or on the disk:

5a. The internal buffer of the actual working area (and associated indices) gets flushed (if changed) to, or updated from the Unix (or Windows) buffer, and is thereafter available to other users using:

- SKIP and any other record movement (GOTO, SEEK etc.), also when moving to the same record by SKIP 0 and GOTO RECNO()
- UNLOCK, DBUNLOCK() or AUTOUNLOCK()

5b. Additionally to the 5.a, the data of the Unix (or Windows) buffers will be physically asynchronously (in the background) flushed to the disk using:

- AUTOUNLOCK() (executed by the AutoLock feature)
- SKIP 0
- CLOSE
- USE
- SET INDEX
- DBCOMMIT()
- DBUSEAREA()
- DBCLEARINDEX()

5c. When issuing the following commands, all the used FlagShip working areas and Unix/Windows buffers will be written immediately (synchronously) to the hard disk:

- COMMIT, COMMIT ALL
- DBCOMMITALL()
- CLOSE ALL, CLOSE DATABASES, DBCLOSEALL()
- QUIT
- user abort (Ctrl-K) or abort by a run-time error

5d. In a multiuser/multitasking environment on Unix or Windows workstation or server, any command of the 5.a or 5.b group flushes/updates the internal FlagShip buffer. On a distributed network (like NFS), using the 5.c commands are the best way to update these buffers.

It is wise to flush the buffers just before UNLOCKing the record or database file.

6. Killing the process by "**kill -9**" or by shut-down (init 0, haltsys), (or by aborting task by Task-Manager in MS-Windows) flushes the Unix (or Windows) buffers to harddisk only, but not the internal FlagShip ones. So by being in REPLACE mode, or not issuing any of

the commands of group 5a...5c after the data changes, may **violate the data integrity** of the database and/or its indices.

7. If the same physical database is **concurrently used** in different working areas of the same application, some restrictions apply:

   - The database must be used (opened) in SHAREd mode,
   - The ALIAS names have to be different,
   - Each opened database occupies one file handle, which applies also for the concurrently used ones. If one of the concurrently used databases is closed, the file handle remains reserved until all instances (work areas) of the same physical database are closed by the current application.

   The handling is equivalent to the use of shared databases, except

   - You should **NOT** access both concurrent databases SIMULTANEOUSLY within the same operation, e.g. REPLACE alias1->field WITH alias2->field (whereby alias1 and alias2 point to the same physical database opened in different working areas). In such a case, use transfer variables, e.g. myVar := alias2->field ; REPLACE alias1->field WITH myVar The same applies also for all other database operations, performed directly or indirectly (e.g. via UDF or code blocks).

     Warning: unpredictable errors (in worst case including a data corruption) may occur otherwise within this simultaneous access on the same database opened in different working areas, since neither the header, nor the database or index can be internally protected from each other during the operation. This is due to a system dependent limitation (Unix and Windows locking mechanism).

   Since the concurrent usage of the same database within the application is used in very special cases (or accidentally) only, FlagShip creates developer warning (i.e. it is displayed in the development mode only, see the FS_SET("develop") function), when such a concurrent usage is detected during the open operation.

8. For simultaneous use of Unix/Linux and MS-Windows based FlagShip applications (usually on SAMBA system), you need to invoke SET NFS ON before the first USE and/or use COMMIT or DbCommitAll() for synchronous flushing, instead of asynchronous DbCommit().

FlagShip offers a unique data **integrity check**, see chapter LNG.4.5. See also <*FlagShip_dir*>/*examples/fsadress.prg* for an example of a network, multiuser/multitasking application.

5. The Input/Output System

For communicating between the user and the application, FlagShip makes comprehensive input and output routines available. This includes keyboard and full screen input and output, as well as i/o to a printer-spooler or file.

Overview of the input/output commands and functions:

Screen oriented input	
@ .. GET [picture] [range] [valid]	\ formatted in/output at row, column
READ [save]	/ for any data type with on-line valid.
@ ... SAY ... GET ...	combined input and output for READ
@ ... GET ... CHECKBOX	create Checkbox processed by READ
@ ... GET ... COMBOBOX	create Combobox processed by READ
@ ... GET ... LISTBOX	create Listbox processed by READ
@ ... GET ... PUSHBUTTON	create Pushbutton processed by READ
@ ... GET ... RADIOBUTTON	create Radiobutton processed by READ
@ ... GET ... RADIOGROUP	create group of Radiobuttons for READ
@ ... GET ... TBROWSE	create Tbrowse entry for READ
CLEAR GETS	delete all open GET fields
KEYBOARD ...	simulation of keyboard input
ON [ANY] KEY ...	execute procedure when key is depressed
ON ERROR ...	simulates FoxPro behavior
ON ESCAPE ...	simulates FoxPro behavior
PUSH/POP KEY	save/restore ON KEY and SET KEY status
SET ANSI	automatic translation of PC8 <-> ANSI chars
SET BELL on/off	bell tone on/off
SET CHARSET	automatic translation of PC8 <-> ANSI input
SET CONFIRM on/off	GET and MENU input with/wo return key
SET COLOR to ...	set a specified color
SET DELIMITERS to ...	set delimiters of GET fields
SET ESCAPE on/off	allow aborting GET...READ with Esc key
SET EVENTMASK	specifies events considered by Inkey()
SET FORMAT to ...	select the format procedure for READ
SET GUIALIGN	align all @..GETs at the same column
SET INPUT	enables/disables the keyboard input
SET INTENSITY on/off	display GET fields w.stand/enhanc attrib.
SET KEYTRANSL	automatic translation of PC8 <-> ANSI input
SET SCOREBOARD on/off	toggle status line display
SET FUNCTION to ...	assign string to FN-key
SET KEY to ...	execute proced. when key is depressed
INKEY()	reads a character from the keyboard buffer
INKEY2STR()	translates inkey number to readable string
INKEYTRAP()	same as Inkey() but process SET KEY trap
ISCOLOR()	are colors available ?

MAXCOL(), MAX_COL()	available screen columns
MAXROW(), MAX_ROW()	available screen rows
MEMOEDIT()	output/editing of memo fields or strings
ONKEY ()	redirect key to UDF, simil.to SET KEY/ON KEY
READEXIT(), READINSERT()	controls READ and MEMOEDIT
READVAR()	get name of input var. in GET, MENU TO
SETCOLOR()	report/redefine the current color setting
SETCOLORBAckgr()	report/redefine the GUI background
UPDATED()	changes within GET/READ ?

Screen and printer output

? and ??	sequential output
?#, ??# and ??##	sequential output to stderr
@ ... SAY [picture]	formatted output at row, column
@ ... BOX	draw a box with user defined edge chars
@ ... DRAW	draw lines in GUI mode
@ ... TO [double]	draw a box with single/double lines
@ ... CLEAR [to ...]	clear a region of the screen
CLEAR screen, CLS	clear the whole screen
SAVE SCREEN [to]	save the contents of the screen
RESTORE SCREEN [from]	restore the saved screen contents
SAVESCREEN(), RESTSCREEN()	save/restore a part of the screen
TYPE ... [to ...]	type the content of a text file
TEXT ... ENDTEXT	display a block of text
EJECT	send a form feed to the printer (file)
ALERT()	display dialog/message window
ANSI2OEM()	convert ANSI string to OEM character set
SETPOS()	set the cursor to specified position
DEVPOS()	set cursor or printer to specif. position
COL(), ROW()	reports the current cursor position
PCOL(), PROW(), SETPRC()	get/set the current printer head position
QOUT(), QQOUT()	sequential output, same as ? and ??
OUTSTD(), OUTERR()	seq.output to stdout/stderr, same as ??
DISPBEGIN(), DISPEND()	set/reset the buffering of screen output
DISPOUT()	display data on act. screen position
DRAWLINE()	same ad @..DRAW
INFOBOX()	display infobox dialog, similar to Alert()
INKEY2STR()	translates inkey number to readable string
PRINTGUI()	start/stop printing in GUI mode
SETCURSOR()	set/report the cursor mode
SETCOLOR(), ISCOLOR()	set/get the current color setting
SETCOLORBAckgr()	report/redefine the GUI background
SCROLL()	enable screen scrolling
SET ANSI	automatic translation of PC8 <-> ANSI chars
SET ALTERNATE to [file]	redirect the ?, ?? output

SET ALTERNATE on/off	enable/disable redirecton
SET BELL on/off	toggle the bell warning on/off
SET COLOR to ...	set specified color
SET CONSOLE on/off	toggle screen output on/off
SET CENTURY on/off	set the century in/output for dates
SET CURSOR on/off	enable/disable the screen cursor
SET DBREAD/DBWRITE	auto translation of PC8 <-> ANSI chars
SET DEVICE to screen/print	redirect the @..say.. output
SET GUIALIGN	align all @..GETs at the same column
SET GUICOLORS	enable default colors also in GUI mode
SET GUITRANSL ASCII	automatic ASCII -> ISO conversion
SET GUITRANSL BOX	draw semi-graphic PC8 @..BOX chars in GUI
SET GUITRANSL LINES	draw semi-graphic PC8 @..TO chars in GUI
SET GUITRANSL TEXT	draw semi-graphic PC8 chars in GUI mode
SET FONT	set new GUI output font
SET MARGIN to	set the left margin for printer output
SET OUTMODE	designates how to display chars < 32
SET PIXEL	enable default coordinates in pixel
SET PRINTER	specify the printing device, enable/disable
SET PRINTER to ...	redirect the ?, ?? output to printer
SET SCRCOMPRESS	enable compress of SAVE SCREEN images
SET SOURCE	automatic ASCII or ANSI translation
SET ZEROBYTEOUT	designates how to display \0 character
REPORT FORM	report output from .FRM file
LABEL FORM	label output from .LBL file
DISPLAY	display database field(s)
LIST	list the database contents

Cursor and mouse handling

COL(), ROW()	reports the current cursor position
MAXCOL(), MAXROW()	reports the available screen size
SETPOS()	set the cursor to specified position
SET CURSOR, SETCURSOR()	enable/disable the screen cursor
MPRESENT()	reports mouse availability (GUI only)
MCOL(), MROW()	reports the current mouse cursor position
MSETCURSOR()	determine/set mouse visibility and/or shape
MSETPOS()	set a new position for the mouse cursor
MSTATE()	return the current mouse state
MHIDE(), MSHOW()	hide/show mouse cursor, set shape
MLEFTDOWN(), MRIGHTDOWN()	reports status of left/right mouse button
MSAVESTATE(), MRESTSTATE()	save/restore the current state of a mouse
MDBLK()	reports/set speed threshold of the mouse

Menu processing	
@ ... PROMPT [message ...]	\ output menu items,
MENU TO	| get the user choice,
SET MESSAGE to ... [center]	/ specify additional help text
ACHOICE()	pop up menus, with UDF control
SET WRAP on/off	set menu item wrapping on/off

Data validation	
@ ... GET...WHEN cond	conditional data entry, with UDF support
@ ... GET...RANGE cond	data entry with numeric bounds check
@ ... GET...VALID cond	data entry with validation check or UDF
DATEVALID ()	test the given date for validity

FlagShip extensions for input/output	
FS_SET ("inmap")	define a keyboard mapping
FS_SET ("outmap" or "map")	define a screen output mapping
FS_SET ("terminal")	determine the cur. terminal and mapping
FS_SET ("loadlang")	load sorting/language table
FS_SET ("setlang")	set sorting/language table
FS_SET ("printfile")	determine the name of the printer file
FS_SET ("typeahead")	control the curses output
FS_SET ("shortname")	truncates file names for MS-DOS compatibility
REFRESH	refresh the screen output

Run-time mode	
APPIOMODE()	returns the current i/o mode (G/T/B)
APPMDIMODE()	determines whether compiled in MDI mode
APPOBJECT()	get the Application object
ISGUIMODE ()	checks if application is running in GUI mode

Mainly in GUI mode used commands and functions	
SET ANSI	automatic translation of PC8 <-> ANSI chars
SET CHARSET	automatic translation of PC8 <-> ANSI input
SET GUIALIGN	align all @..GETs at the same column
SET GUICOLORS	enable default colors also in GUI mode
SET GUITRANSL ASCII	automatic ASCII -> ISO conversion
SET GUITRANSL BOX	draw semi-graphic PC8 @..BOX chars in GUI
SET GUITRANSL LINES	draw semi-graphic PC8 @..TO chars in GUI
SET GUITRANSL TEXT	draw semi-graphic PC8 chars in GUI mode
SET FONT	set new GUI output font
SET PIXEL	enable default coordinates in pixel
SET SCRCOMPRESS	enable compress of SAVE SCREEN images
SET SOURCE	automatic ASCII or ANSI translation

APPIOMODE()	returns the current i/o mode (G/T/B)
APPMDIMODE()	determines whether compiled in MDI mode
APPOBJECT()	get the Application object
COL2PIXEL ()	converts columns into pixels
COLOR2RGB ()	transforms color string or object into RGB
DRAWLINE ()	same ad @..DRAW
GETALIGN ()	align all @..GETs at the same column
INFOBOX ()	display infobox dialog, similar to Alert()
ISGUIMODE ()	checks if application is running in GUI mode
LISTBOX ()	set/process listbox or combobox OOP
PIXEL2COL ()	converts pixel value to columns
PIXEL2ROW ()	converts pixel value to rows
PRINTGUI()	start/stop printing in GUI mode
PUSHBUTTON ()	set/process push button OOP
RADIOBUTTON ()	set/process radio button OOP
ROW2PIXEL ()	converts rows into pixels
SETCOLORBACKGR()	set/get background color in GUI mode
STRLEN2COL ()	retrieves the true length of string in cols
STRLEN2PIX ()	retrieves the true length of string in pixel
STRLEN2SPACE()	retrieves number of spaces to fill a string

5.1 The Output System

Program output can be specified for the screen, console, printer or file. Sequential and screen-oriented output may be interchanged as needed.

5.1.1 Sequential (Console) Output

The output commands and functions, which operate sequentially (in raw mode) are sometimes called "console" output. They don't need to be positioned to a specific screen or printer row and column. The output always starts on the actual screen/printer position. The console operations are:

Command / Function	Description
? and ??	sequential output with/wo new line
QOUT(), QQOUT()	sequential output, same as ? and ??
ACCEPT	output a prompt, accept a string input
INPUT	output a prompt, accept any input
WAIT	output a prompt, wait for input
TEXT ... ENDTEXT	display a block of text
DISPLAY	display database field(s)
LIST	list the database contents
TYPE	type the contents of a text file
REPORT FORM	report output from .FRM file
LABEL FORM	label output from .LBL file
EJECT	send a form feed to the printer (file)
SET OUTMODE	designates how to print chars < 32
SET ZEROBYTEOUT	designates how to display \0 character

Output of these console operations may also be simultaneously re-routed to a printer or an ASCII file using the SET ALTERNATE, SET EXTRA and SET PRINTER commands. Most of the console commands support the TO PRINTER or TO FILE clause for preferred output re-direction. To disable simultaneous output to the screen, use the SET CONSOLE OFF command.

For the console output on the screen, FlagShip interprets/executes some special characters instead of outputting their graphical screen equivalents:

```
chr(7)   sounds a BELL, beep                          instead of: (graph)
chr(8)   execute BACKSPACE, move one char left        instead of: (graph)
chr(9)   execute TAB, move to the next tab position   instead of: (graph)
chr(10)  execute LF, move to the next line            instead of: (graph)
chr(13)  execute CR, move to the first column         instead of: (graph)
```

All other characters printed to the screen will be mapped, in Terminal i/o according to the TERM description in *<FlagShip_dir>/terminfo/FSchrmap.def* (see chapter 5.1.4 and section SYS).

5.1.2 Full-screen Output

Some commands and functions are designed to operate in full-screen mode. The output begins on the required (or actual) row/ column position.

Command / Function	Description
@ ... SAY	formatted output at row, column
@ ... GET	formatted output for data entry by READ
@ ... PROMPT	output a menu item for MENU TO
@ ... TO	draw a box with single/double lines
@ ... BOX	draw a box with user defined edge chars
@ ... DRAW	draw lines in GUI mode
DEVOUT()	display data on current screen position
DISPOUT()	display data on current screen position
@ ... CLEAR	clear (a region) of the screen
CLEAR SCREEN, CLS	clear the whole screen
SAVE SCREEN	save the contents of the screen
RESTORE SCREEN	restore the saved screen contents
SAVESCREEN(), RESTSCREEN()	save/restore a part of the screen
SETPOS()	set the cursor to specif. position
DEVPOS()	set the cursor or printer to specif. posit.
COL(), ROW()	reports the current cursor position
MAXCOL(), MAXROW()	reports the max. available column/row
PCOL(), PROW(), SETPRC()	set/report the current printer head posit.
SET CURSOR, SETCURSOR()	set/report the cursor mode
SETCOLOR(), ISCOLOR()	set/report the current color setting
SETCOLORBAckgr()	report/redefine the GUI background
SCROLL()	enable/perform screen scrolling

The command @..SAY and function DEVPOS() operate in the same manner for printer output, when SET DEVICE TO PRINT is specified. To reroute output to file instead of printer, use the SET PRINTER TO.. command.

> **Note:** the screen oriented output in Terminal i/o uses the Curses package of the Unix system (or pdcurses in Windows). This package usually clears the screen at the program begin and at the time of the termination. Since this may disturb some special process modes (e.g. applications running in background or requires to redirect stdout to file), it is possible to disable Curses in the INIT function cursinit() if you do not use full screen i/o; see details in section SYS.2.7. Even better for such purposes is to use the Basic i/o mode by -io=b compiler or command-line switch, see also section LNG.1.2 and FSC.1.3

5.1.3 Special Output

The following output performs a special action:

Function	Description
DISPBEGIN()	start the buffering of screen output (Terminal i/o)
DISPEND()	disables the buffering, output buffer
OUTSTD()	sequential output to stdout, same as ??
OUTERR()	sequential output to stderr, same as ??#
?#, ??# and ??##	sequential output to stderr

The function DISPBEGIN() disables the actual direct screen output. All subsequent sequential or screen oriented output will be stored in an internal buffer and printed to the screen when executing the function DISPEND(). It may be used to prepare complex screen output for slow terminals in the background.

Interrupting the execution by ^O, ^K or by a run-time error prints the hidden buffered screen output too.

The functions OUTSTD() and OUTERR() work the same as QQOUT() or the ?? command, except for the re-routing possibility to a printer or file. The OUTSTD() sends the output to stdout (the standard screen), the OUTERR() sends it to stderr instead. Normally, the stdout and stderr reflect the same output device. With the sh (bourne) or ksh (korn) shell, the stderr program output may be rerouted to an other device or file, e.g.:

```
$ a.out 2>/dev/tty2
$ a.out 2>/usr/myfile.txt
```

The OUTSTD() is processed by the curses library, and therefore should not be rerouted.

5.1.4 Terminal Output and Mapping

In the GUI mode, the ? and ?? commands use the associated FONT and character set, see SET FONT command and FONT class, e.g. oApplic:Font.

To support screen output on any terminal in Terminal i/o mode, FlagShip uses the standard Unix interface library (n)curses and the terminal description terminfo. The extended terminfo description *<FlagShip_dir>/terminfo/FStinfo.src* supplied with your FlagShip system, enhances the standard terminfo with function keys, extended cursor keys and so on. All the extended terminal descriptions begin with the prefix FS, like FSansi, FSVT100 etc. See section REL for available enhanced terminals.

> **Note:** the Unix system searches for the terminal description in the default directory /usr/lib/terminfo or in the one given by the environment variable TERMINFO.

Many Unix terminals cannot directly display the whole PC-8 character set, but have the graphic characters available in the "alternate" mode. Such PC-8 characters may be mapped to the

equivalent character in another mode or to a similar, available ASCII character, like "→" to ">" or "ü" to "u" etc. The default mapping for the predefined terminals (see sect. REL) is available in the ASCII file *<FlagShip_dir>/terminfo/ FSchrmap.def*

> **Note:** On starting up, FlagShip searches for the character mapping file in the actual directory and in /usr/lib/terminfo. The search path may also be given using the environment variable SCRMAP. If the file is not found, or if the actual TERM setting is not included there, no output mapping will be done. During program execution, the mapping file and terminal name may be redefined, using FS_SET("outmap").

Proper terminal setting is essential for the correct functionality of your application. Example:

```
[ $ TERMINFO=/usr/home/myterm; export TERMINFO ]
[ $ SCRMAP=/usr/home/myterm; export SCRMAP     ]
  $ TERM=FSansi; export TERM
  $ a.out
```

A wrong terminal setting may cause garbage using the screen-oriented output, colors, mapping etc. Make sure that additional Unix mapping (ttymap, mapchan etc.) is disabled. You preferably may use newfswin, newfsterm or newfscons scripts (FSC.6.7) which sets corresponding environment. See more information about terminals in section SYS and the system-specific information in section REL. The environment is described in section FSC.

5.1.5 Colors

FlagShip supports both monochrome and color terminals. The availability of the color output depends on your hardware and the used operating system. See additional information in the Release Notes (sect. REL) if your system does not fully support colors.

The color capability is defined in the actual terminal description, see above chapter 5.1.4 and section SYS.

During program execution, you may examine the color capability using the function ISCOLOR() and set/change the preferred colors using SET COLOR TO or SETCOLOR(). Example:

```
IF ISCOLOR()
   SET COLOR TO "W+/B,N/BG"
ELSE
   SET COLOR TO "W+/N,N/W"
ENDIF
CLEAR SCREEN
@ 10,20 SAY "Hello world"

** or the same in another notation:
SETCOLOR (IF (ISCOLOR(), "W+/B,N/BG", "W+/N,N/W"))
CLEAR SCREEN
mytext = "Hello world"
@ 10,20 SAY mytext
@ 11,20 GET mytext
```

In GUI mode, the color support via SET COLOR or COLOR clauses is disabled by default to ensure proper GUI look & feel. You may enable it at any time by setting SET GUICOLOR ON. Alternatively, you may override this setting by using the GUICOLOR clause available in the most commands (like ?, ??, @..SAY etc), or the corresponding parameter in the translated function. You also may set the default background by using the SETCOLORBACKGR() function.

5.1.6 Printer and File Output

All of the sequential output operations (5.1.1) and some of the screen- printed ones (5.1.2) may be redirected to printer either directly via the **PrintGui()** function or SET PRINTER TO command, as well as to an ASCII file by using the SET PRINTER, SET ALTERNATE, SET EXTRA or SET DEVICE commands.

Because of the multi-user and multi-tasking capability of Unix and Windows, FlagShip doesn't output on the printer directly, but uses a **spool file** instead. This avoids garbage being printed if several users printed at the same time. This spool file may be printed at any time (during the execution of the application or later), see LNG.3.4 and SET PRINTER command.

If necessary, direct printer output is available by using SET PRINTER TO /dev/lp0 etc. or by SET PRINTER TO LPT3 in Windows. Also, re-direction to other devices, such as a second screen, is possible in Linux/Unix with SET PRINTER TO /dev/tty2a and so on.

In special cases, you may also redirect the stdout directly to file; see details in section SYS.2.7.

Printer output from **GUI based application**: in the *<FlagShip_dir>/system/initiomenu.prg* file, there is pre-defined menu entry "File->Printer Setup" which opens a dialog window for the preferred printers and default drivers detected on your system. Once the statement SET PRINTER OFF was detected and something was printed to the spooler file, also the menu entry "File->Print" become available. The user may then simply click on this menu entry to perform the print action, which is pre-defined in the initiomenu.prg but freely re-definable, see the description of InitIoPrint() in initiomenu.prg, and example in the *<FlagShip_dir>/examples/printer.prg* and *printgui.prg*

You have different choices to print:

- re-direction of screen output parallely to printer via PrintGui()
- output directly to printer port / device
- output to spooler file and spool/print it from application
- output to spooler file and spool/print it externally
- output to spooler file and spool/print it by default/selected driver

Each step is described in detail below.

5.1.7 Printer Output to a remote printer

Printer output via Ethernet: when you want to redirect the local printout (at the Unix server) to a remote printer which has an Ethernet printserver module installed, you simply set lp (or lpr) to the printer IP address.

In MS-Windows, you may print to any other shared printer too, either by specifying the share name in oPrinter class for GUI based application, or by redirecting the share by NET USE e.g. to LPT3 either in CMD or from the application by RUN ("NET USE \\other\user\laser5 LPT3:") and printing via SET PRINTER TO LPT3:

Printer output via Terminal emulator: you may also print remotely via terminal emulator (e.g. from MS-Windows 9x/NT) when the emulator supports transparent printer redirection via VT escape sequences. An example is in the section CMD:SET PRINTER. An usable overview of terminal emulators is e.g. on http://winfiles.com/apps/nt/terminals.html (we have tested the CRT from http://www.vandyke.com which work fine also in ANSI color mode).

5.1.8 Printer Output, Options

There are three different kinds for printer output:

a) redirect the screen output (text and graphics) to any available printer (parallel, serial, USB, network) in GUI mode, using the printer driver via PrintGui() function

b) passing text output directly to printer device via SET PRINTER TO ...

c) passing text output to spool file and print thereafter (default).

There are several ways to create the printer output and pass it to your printer:

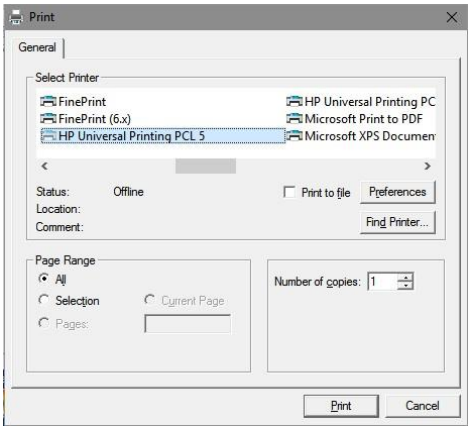
a. Redirection to printer driver

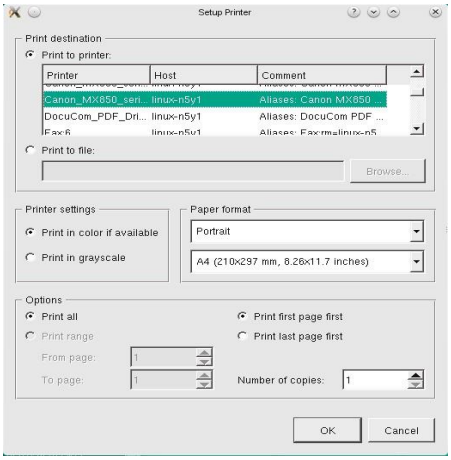
In GUI mode, you may redirect text and graphics (?, ??, @..SAY, @..DRAW) to any available printer (also GDI) device connected by parallel, serial, USB, LAN or WLAN interface, or shared over network. This is very similar to output from any other GUI application like OpenOffice, Word etc. via common CUPS in Linux or Winspool in Windows.

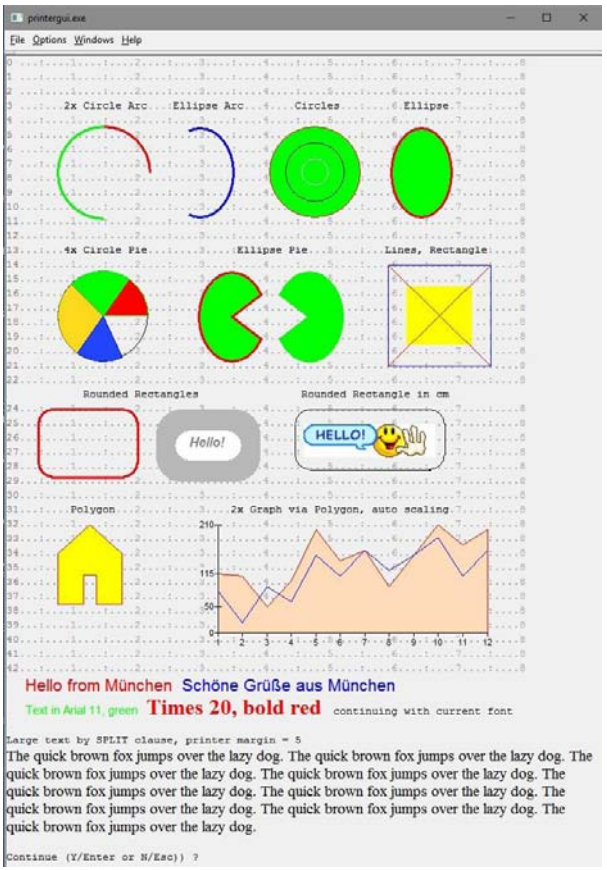
To print to GDI printer, simply invoke **PrintGui**(.T.) to start printer buffering, optionally parallel to screen output. With PrintGui() w/o parameter, you will start the printer output, for example

```
PrintGui(.T.)               // start buffering for GUI printer
@ 5,10 SAY "Text at row/col 5,10"
? "This is other text"
@ 7,5,12,9 SAY IMAGE FILE "mypicture.jpg" SCALE UNIT=CM
PrintGui(.F.)               // stop buffering to GUI printer
PrintGui()                  // print to selected printer
```

See further details in section FUN.PrintGui() and CMD.SET GUIPRINTER as well as examples in *<FlagShip_dir>/examples/printergui.prg*







b. Printing Using Spooler File

As mentioned above and described in CMD.SET PRINTER, FlagShip generates spooler-printer-file per default, when SET PRINTER is ON. The default name is <applic>.<process-id-num> located in the current directory. It name can be determined by FS_SET("print") function. You may set the print-directory by environment variable FSOUTPUT, e.g. SET FSOUTPUT= C:\my\path in Windows, or "export FSOUTPUT=/my/path" in Linux/Unix. If you wish to use another name, use SET PRINTER TO "my_output_name" command, where the output name is either newly created text file or available device name.

Every user-task gets unique printer-spooler-file name which don't interfere with other users or applications. The printer-spooler-file can be printed by several ways at the time of the application execution, or any time thereafter.

A typical command sequence for printing is

```
  SET CONSOLE OFF           // don't print to screen
  SET PRINTER ON            // redirect output to default printer (file)
  ? "first line"
  ? "second line"
  eject                     // print form feed (new page)
  SET PRINTER OFF           // end of printer redirection
  SET CONSOLE ON            // continue with output to screen
  ? "Output created into file " + FS_SET("print")  // opt. message to user
```

and/or

```
  SET DEVICE TO PRINT       // redirect @..SAY to default printer (file)
  @ 1,5 SAY "first line"
  @ 2,5 SAY "second line"
  SET DEVICE TO SCREEN      // end of @..SAY redirection
```

or by combined "?" and "@..SAY" commands.

With the FS_SET("prset") function, you may set special control sequences of your printer, e.g. CR+LF for new line, FF for page break, etc. If required, invoke this setup before above printing, since the given sequences will be included directly in the printer output (or spooler file).

c. Printing Directly To Port Or Device

If you don't need to bother with interfering by other users/applications, you may print directly to available device (port), e.g.

```
  #ifdef FS_WIN32    /* following sequence is compiled in Windows */
    SET PRINTER TO LPT1       // Windows: 1st parallel port
    SET PRINTER TO PRN        // Windows: default port
    SET PRINTER TO COM2       // Windows: 2nd serial port
  #else             /* following sequence is compiled in Linux/Unix */
    SET PRINTER TO /dev/lp0   // Linux/Unix: 1st parallel port
    SET PRINTER TO /dev/stty1 // Linux/Unix: 2nd serial port
  #endif
  ... print according to 5.1.8.a
  SET PRINTER TO              // end of printer redirection
```

d. Printing Via FlagShip's Printer Class

The easiest way to do print in GUI mode is to use PrintGui() function which communicates with FlagShip printer class, and supports nearly all screen oriented output also for printing. You alternatively may execute the _oPrinter:exec() or _oPrinter:execFormatted() methods, you may select the printer driver via the common printer pop-up dialog oPrinter:Setup(). It supports all available local and remote printers, also attached via network.

```
  ... print according to 5.1.8.b
* _oPrinter:Setup()         // optional, select printer driver
  _oPrinter:Exec()          // print the default spooler file
* _oPrinter:ExecFormatted() // formatted printout
```

e. Passing Spooler-File To Printer

As said above, the standard spool file is named <application>.<pid> and it real name can be determined by FS_SET("print") function. You also may specify any other file name by e.g.

```
SET PRINTER TO ("/my/path/MyFile.prn")   // output into user file
... print according to 5.1.8.b
SET PRINTER TO                           // end of printer redirection
```

Once the output file is created, you may pass it to the printer port. Since it is plain text file, you may alternatively use one of copy commands

```
#ifdef FS_WIN32    /* following sequence apply for Windows */
 COPY FILE FS_SET("print") TO ("LPT1:")      // copy to 1st parallel port
 RUN ("COPY " + FS_SET("print") + " LPT3:")  // copy to user port
 COPY FILE ("C:\my\path\MyFile.prn") TO ("LPT2:")    // user file
#else             /* following sequence apply for Linux/Unix */
 COPY FILE FS_SET("print") TO ("/dev/lp0")   // copy to 1st parallel port
 RUN ("cp " + FS_SET("print") + " /dev/lp1") // copy to 2nd parallel port
 COPY FILE ("/my/path/MyFile.prn") TO ("/dev/lp1")   // user file
#endif
```

You may pass the spooler file to your printer at any time by using operating system commands. You either may specify the name of spooler file, or display the automatically created file name by FS_SET("print"). Assume, the file is named "myapplic.prn", invoke in Windows

```
C:> COPY myapplic.prn LPT1        // print via 1st parallel port
C:> COPY myapplic.prn PRN         // print via default parallel port
C:> COPY myapplic.prn LPT3        // print via user defined port
```

or on Linux

```
lpr myapplic.prn                  // print to default printer
lpr -PmyPrinter myapplic.prn      // print to selected printer
cp myapplic.prn /dev/lp0          // print to 1st parallel port
```

f. Printing Via Redirected Port

In Windows, you may use NET USE to redirect an unused port to network (or local) driver:

```
C:> NET USE LPT3 \\ComputerName\PrinterSharedName
```

where

ComputerName is either IP address, or shared computer name, or local computer displayed by NET VIEW

PrinterSharedName is the shared name of the printer, see Start -> Printer and Faxes -> (Right click on printer) -> Properies -> Sharing -> Shared name

You also may do it from the application, e.g. by

```
RUN ("NET USE LPT3 \\Server\Printer2")
RUN ("NET USE LPT3 \\MYCOMPUTER\Printer5")
```

and then print by

```
SET PRINTER TO LPT3
... print according to 5.1.8.b
SET PRINTER TO
```

or simply copy the output to your printer

```
... print according to 5.1.8.a
COPY FILE FS_SET("print") TO ("LPT1:")       // copy to 1st parallel port
* RUN ("COPY " + FS_SET("print") + " LPT3:") // copy to user port
* COPY FILE ("C:\my\path\MyFile.prn") TO ("LPT2:")    // user file
```

This will work fine on printers supporting text mode (like matrix or line printers, or printers with PCL or postscript capability). It usually will not work on GDI printers, see below.

In Linux, the CUPS tool will manage also redirection to network printer or to USB printers. Once CUPS is set, simply issue

```
... print according to 5.1.8.b
RUN ("lpr -PmyPrinterName " + FS_SET("print"))
```

g. Printing On GDI Based Printers

A GDI based printer is also known as Windows-printer. It does not include hardware and software to manage text input, but requires rastering by CPU using special printer driver, see further details in FUN.PrintGui() or in *http://en.wikipedia.org/wiki/Graphics_Device_Interface#GDI_printers*

The easiest way in GUI mode is to use PrintGui() function which allows you to redirect screen output to any available printer, see 5.1.8.a above. You also may use the Printer class (see 5.1.8.d above). An example is in *<FlagShip_dir>/examples/printergui.prg*

In terminal i/o mode, you will need some software to create the GDI image (e.g. http://www.dos2usb.com). You may pass the output to another GUI application (e.g. to Notepad) to be printed there. In Linux you may use the "lpr" tool to print it. For example

```
... print according to 5.1.8.b
#ifdef FS_WIN32
  ? "Select your printer via 'Print' menu"
  RUN ("Notepad " + FS_SET("print"))            // for MS-Windows
#else
  RUN ("lpr -PmyUsbPrinter " + FS_SET("print"))  // for Linux
#endif
```

which is similar to method 5.1.8.c for direct print in GUI mode, but is usable also for terminal i/o based applications.

5.2 The Input System

Normally, user-program communication will be done via keyboard input. Using Unix pipes and redirection input from a file will also be accepted.

5.2.1 Keyboard Input

FlagShip stores the user keyboard entries in an internal type-ahead buffer. The buffer size is set by default to 80 characters, but may be resized using the SET TYPEAHEAD command, to any size from 2 bytes up to 2 Gigabytes. The characters stored in the internal buffer will be removed by the input commands and functions ACCEPT, ACHOICE(), DBEDIT(), INKEY(), INKEYTRAP(), INPUT, MEMOEDIT(), READ and WAIT.

Storage in the type-ahead buffer allows the required input to be input before the system has finished processing the last command. The characters are stored and removed according to the first-in, first-out principle. The next character available in the type-ahead buffer, if any, may be checked by the NEXTKEY() function. The last 10 keys removed from the buffer are available using LASTKEY().

To simulate a user input, characters may be pushed into the type-ahead buffer using the KEYBOARD command. Also special characters, like the RETURN or function keys, will be accepted.

> **Note:** key codes, which produce a negative INKEY() code (see appendix), will be stored in the type-ahead buffer as two characters, to ensure that they will be properly read.

The type-ahead buffer will be cleared by the CLEAR TYPEAHEAD command or by removing all awaiting characters with e.g.:

```
DO WHILE INKEY() # 0
ENDDO
? "last key pressed:", LASTKEY()
```

5.2.2 Keyboard Redefinition

Using the SET KEY or ON KEY command, any key may be redefined to execute the specified user defined procedure (UDP) instead of processing the depressed key.

The command SET FUNCTION assigns a string to a function key. When pressing such a function key in input mode, the string will be pushed into the type-ahead buffer and removed by the input command or function (see LNG.5.2.1).

If the same key is redefined by SET KEY and SET FUNCTION, the SET KEY has precedence, and the SET FUNCTION assignment only becomes active again, after the SET KEY redefinition is disabled.

Up to 48 keys may be redefined at the same time. On program start, the F1 key is automatically redefined to the HELP procedure (just as if the user had SET KEY 28 TO HELP). If such a UDP exists, pressing the F1 key will call this procedure, mostly used to execute a context specific help.

> **Note:** on some Unix systems (like SUN with X/Open) or terminals, the F1 key is pre-configured or sometimes hard-wired to system help. In such cases, if the configuration may not be changed, use another FN key for the context help purposes, e.g. the F2 key: SET KEY 28 TO ; SET KEY -1 TO HELP

To redefine the system keys Ctrl-O (activate the debugger) and Ctrl-K (abort the program), use the FlagShip functions FS_SET("debug") and FS_SET("break").

The availability of the ESCAPE, debug and BREAK key to the user is controlled by SET ESCAPE, ALTD() and SETCANCEL().

5.2.3 Full-screen Input

Similar to full-screen output, many FlagShip input commands or functions operate at specified screen position. These are:

Command / Function	Description
@ ... GET	} formatted in/output at row, column
READ	} for any data type with on-line valid.
READMODAL()	user modifiable READ (getsys.prg)
MEMOEDIT()	output/editing of memo fields or strings
DBEDIT()	display (modify) records from .dbf
GET class	low-level GET/READ system
TBROWSE class	display (modify) .dbf or array
TBCOLUMN class	set columns for the TBROWSE class

These operations display (formatted) data at a specified screen position and wait for a user input. The navigation keys (cursor up/down/left/right, End, Home, PgUp, PgDn etc.) support the full-screen handling.

Many operations support on-line validation of the user data input, e.g. using the RANGE and VALID clause for @...GET, defining a UDF in MEMOEDIT or DBEDIT etc. The GET/READ system also supports a pre-validation, using the WHEN clause.

Rerouting the full-screen-input to a printer or a file is pointless and is therefore not supported.

5.2.4 Menu System

To perform menu oriented choices on the screen, two menu systems are available in FlagShip: @..PROMPT / MENU and ACHOICE().

Command / Function	Description
@ ... PROMPT	} output menu items,
MENU TO	} get the user choice,
SET MESSAGE	} specify additional help text
ACHOICE()	pop up menus, with UDF control
ALERT()	pop up menu for messages etc.

Database display using DBEDIT() and the array/database TBrowse system are also very useful in executing the required user choices. The behavior of DBEDIT() is fully modifiable, since included in source code in the <FlagShip_dir>/system/dbedit.prg file.

In all menu systems, the user can move a light bar specifying the actually selected item (up and down or left and right), or press the first character of the menu text to perform fast selection.

The menu systems ACHOICE(), DBEDIT() and TBrowse support scrolling to invisible items. MENU TO has wrapping to the first/last item and additional help messages too. All of the menu systems may be nested to any level.

5.2.5 Input Mapping

To support the keyboard input on any terminal, FlagShip uses in Terminal i/o mode the standard Unix curses interface library and the terminal description terminfo, see chapter 5.1.4.

Many Unix keyboard/terminals do not support the whole PC-8 character set, function keys or support an ISO set only. FlagShip allows the incoming keyboard characters to to be mapped to other PC-8 characters. The default input-mapping for the predefined terminals (see sect. REL) is available in the ASCII file <FlagShip_dir>/terminfo/FSkeymap.def

> Note: On starting up, FlagShip searches for the character mapping file in the actual directory and in /usr/lib/terminfo. The search path may also be given using the environment variable SCRMAP. If the file is not found, or if the actual TERM setting is not included there, no input mapping will be done. During program execution, the mapping file and terminal name may be redefined using FS_SET ("inmap").
>
> If the terminal description specified by the TERM environment variable is not found at all, the further program execution is aborted with the message "...cannot handle the terminal ..."

Proper terminal setting is essential for the availability of function, cursor or special keys and for the correct interpretation of incoming characters. Example:

```
[ $ TERMINFO=/usr/home/myterm; export TERMINFO ]
[ $ SCRMAP=/usr/home/myterm; export SCRMAP     ]
[ $ mapkey /usr/lib/keyboard/FSkeys.us         ]
  $ TERM=FSansi; export TERM
  $ a.out
```

If the terminal is set up incorrectly, special keys may be misinterpreted. Make sure the additional Unix keyboard mapping or character set (mapkey, ttymap, xset, vidi, aixterm, hpterm etc.) is properly set or disabled. For more information see section SYS and the system-specific information in section REL.

For your convenience, we have added three scripts which sets your environment automatically. They may slightly differ according to the Operating System used (see details in sect. REL), but the general invocation is

```
$ newfscons a.out            being on console
$ newfswin  a.out            being in Xwindows
$ newfsterm a.out            from remote terminal
```

Because special keys (cursor, FN keys etc.) mostly produce a key-escape- sequences, setting the serial communication line too high or too low (like 36 Kbaud and above or below 1200 baud) may cause a misinterpretation of some incoming special keys, e.g. the cursor key as the ESCAPE key only.

> Note: For portable programs, omit the [Alt]+[key] combinations, since they are not available on most Unix terminals. For some terminals, FlagShip maps the Alt-FN keys to Shift-Ctrl-FN key instead. See section REL and the file FStinfo.src.

5.3 Difference between Terminal and GUI

In section LNG.1.2 are described the three different modes of operation which FlagShip support:

GUI : graphical oriented i/o, requires X11 on Linux, or the MS-Windows (both 32/64bit)
Terminal: text/curses oriented i/o e.g. for console or remote terminals, same behavior as FlagShip 4.48.
Basic : basic/stream i/o e.g. for Web, CGI, background processing etc. The screen oriented i/o is roughly simulated for source compatibility purposes.

Please refer there for additional description. We will focus here the difference from the programmer/developer view.

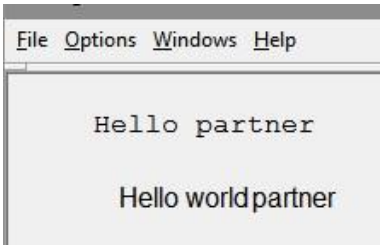
Most probably, the only significant difference in GUI to Terminal based application will be visible when using **proportional fonts** (which is the default of most window managers) and overwriting partition of screen text via @...SAY or setpos(), devpos(). Usually, you will not see any difference with the common xBase statements

```
@ 2,5 SAY "Hello world"
?? ", that's me"
```

which displays the text "Hello world, that's me" as expected, starting at row 2, column 5 (i.e. at y=45, x=30 in pixels). A visible difference occurs, when not considering that the length of proportional text may differ significantly to the size of fixed font text. So, the sequence

```
SET FONT "Courier", 12
cText := "Hello "
@ 1,5 SAY cText + "world"
@ 1,5 + len(cText) SAY "partner"

SET FONT "Arial", 12
cText := "Hello "
@ 3,5 SAY cText + "world"
@ 3,5 + len(cText) SAY "partner"
```



will display "Hello partner" with fixed fonts and in terminal based application, but "Hello world partner" with proportional fonts in GUI. Why? The text of "Hello world" is 52 pixel long (Helvetica 12) and ends at pixel 82, but the column 11 of @..SAY is calculated as x=99 in pixels, check by e.g. Col2pixel(len("Hello world")) or Strlen2col(cText).

So most possibly, only such programming constructs will need your attention and small adaption of the available source code. BTW, there are several ways to fix/program such proportional font behavior:

```
@ 3,5 SAY "Hello "
r := col()              // get the current column (here x = 7.89)
@ 3,r say "world"       // and use the retrieved column
@ 3,r say "partner"     // also later on
```

or

```
@ 3,5 SAY "Hello world"
@ 3,5+Strlen2col("Hello ") SAY "partner"  // use real text width
```

or

```
SET FONT TO "courier" SIZE 12       // set fixed font,
// SET FONT TO "adobe-courier",13   // (often better alternative)
@ 3,5 SAY "Hello world"             // the "old" code
@ 3,5+6 SAY "partner"               // remain unchanged
```

and so forth, there are other examples in the @...SAY description.

The same apply, when you try to overwrite, or clear previously displayed text by spaces using proportional font. The sequence

```
set font "Arial", 12    // proportional font
@ 3,5 say "XXXXXXXXXX"  // Col() = 175 pixel
@ 3,5 say "xxxxxxxxxx"  // Col() = 135 pixel
@ 3,5 say "          "  // Col() = 105 pixel
```

will work fine with fixed font, but will not produce the result you are expecting with proportional character set; you will not clear this area by spaces, but will see " xxxXXX". Why? All three text lines are ten characters long, but the proportional text occupy 40 to 110 pixels (see also LNG.5.3.2 below and the Strlen2pix() function). So, in this example, the second output overwrites first 70 pixels of the first, and the third output clears first 40 pixels only. To clear the first output, best to use @ 3,5 CLEAR TO 3,14 which automatically calculates the correct column size, or determine the required amount of spaces by @ 3,5 SAY SPACE(Strlen2space ("XXXXXXXXXX")) To clear screen region, use e.g. @ 3,5 CLEAR TO 10,MaxCol() To clear whole screen, use CLS or CLEAR SCREEN.

Overlayed widgets/controls: In GUI mode, the Get, Memoedit, Tbrowse and other objects are drawn as widgets (or controls in MS terminology) and hence displays as overlayed layers in "third dimension". To ensure the backward compatibility to FS44 and Clipper, these widgets remains visible also after the READ, Get, Memoedit etc. process finishes. The widget is cleared at the end of the variable visibility scope or via @.. CLEAR TO .. or CLS command. When the @..GET object overlays another object (like Tbrowse), you may want to clear the Get widget after READ finish via READ CLEAR command or via oGet:Destroy() method, which removes it from the topmost visibility layer. You may see these GUI widgets (controls) as small, modal sub-windows.

In GUI mode, FlagShip uses different "layers" for the display. At the lowest layer (display) resides the application screen layer and plain display by @..SAY, ?, ??, @..BOX, save/ restscreen etc. In layer above are standard widgets (controls) like @..GET, @..PROMPT, Pushbutton, Listbox, Achoice, Memoedit, Dbedit, Browse, Tbowse etc. In topmost layer resides overlayed widgets like dialogs, MDIopen or Wopen sub-windows.

You therefore cannot write "over" the widget in GUI mode (as opposite to the flat "two-dimensional" Textual i/o) using the common ?, ??, @..SAY.., @..BOX etc. statements which writes directly to the background, or to lowest layer in GUI.

Colors and lines drawing are per default disabled in GUI mode to provide proper GUI look & feel. You may enable the color support in GUI mode via SET GUICOLOR ON or Set(_SET_GUICOLORS,.T.). To draw semi-graphic ASCII characters 179..218 in GUI mode, use SET GUITRANSL TEXT ON or Set(_SET_GUIDRAWTEXT,.T.) for an automatic translation of text strings to graphic ASCII chars. To draw lines and boxes via @..TO.. and @..BOX in GUI mode too, use SET GUITRANSL LINES ON and/or SET GUITRANSL BOX ON or the corresponding Set(_SET_GUI*) function, see details in the section CMD and FUN. To disable the color or drawing support, set it OFF. You may draw lines also by the @..DRAW command. All these commands and functions may remain global in the source code, they will be ignored when the application run in Terminal or Basic mode.

5.3.1 Coordinates

In Terminal and Basic i/o mode, the current cursor or display position is reported in rows and columns. The coordinate system start at 0,0 in the upper left edge, the max. display size is reported by MaxCol() and MaxRow() for the bottom right edge (i.e. is 79,24 for 80x25 screens).

In GUI mode, any output is pixel oriented. For your convenience and to achieve cross compatibility to textual based applications, FlagShip supports also coordinates in common row/column values. It then internally re-calculates the given rows by using Row2pixel() and columns by using Col2pixel() functions. The character and line spacing is affected by the currently used font, see details in LNG.5.3.2 below.

Note: one **pixel** is a "dot on the screen", i.e. smallest single component of a digital image. The character size in pixel depends on the used font (see CMD.SET FONT and 5.3.2 below) and can be determined by Row2pixel(), Col2pixel() and Strlen2pix(). For example, with SET FONT "Arial",12 is the width of letter "X" 11 pixel, but "i" occupy only 4 pixel; the row height is 21 pixels, and column stepping is 13 pixels = width of "M". In contrast to, fixed fonts like SET FONT "Courier",12 always have the same size, here is the character and column width 11 pixel, and the row height 20 pixels. These data depends on the screen resolution, here for WUXGA desktop monitor with resolution of 1920x1200 pixel (check by oApplic:DesktopWidth and oApplic:DesktopHeight, see section OBJ.Application).

The GUI coordinate system is same as in Terminal i/o and is in range 0,0 to MaxRow(), MaxCol(). Nevertheless, you may use pixels directly, either globally by SET PIXEL ON, or by using the corresponding PIXEL clause or parameter in i/o commands and functions. In addition to, you also may use coordinates in mm, cm or inch (see SET COORD UNIT in section CMD), which are then internally re-calculated to pixel (or to lpi for printer output).

5.3.2 Fonts

In GUI mode, FlagShip uses typographic fonts. A "font" is the combination of typeface and other qualities such as size, pitch and spacing. For example, Helvetica is a typeface that defines the shape of each character. Within the Helvetica typeface, there are many fonts to choose from, i.e. with different size, bold and italic style and so on.

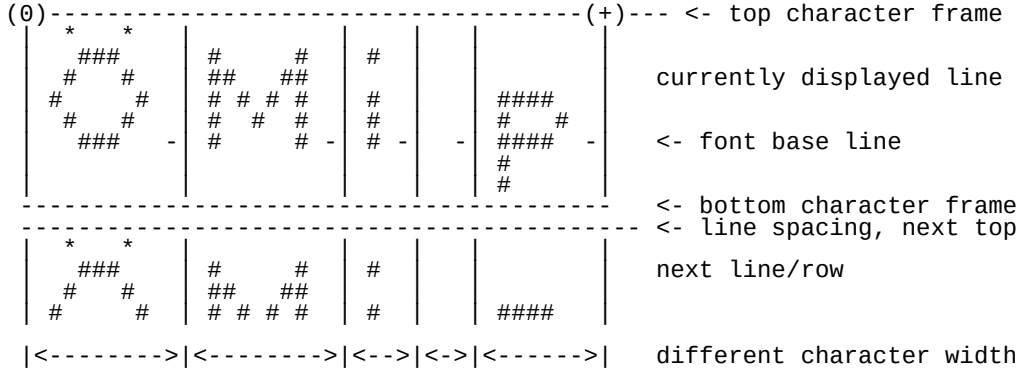
In Terminal i/o mode, FlagShip uses the standard terminal font. In MS-Windows, you may assign the required font by a right mouse click + Properties of the console window. In Unix/ Linux, the font can be assigned to the color_xterm or xterm system command (see the man pages or e.g. the <FlagShip_dir>/bin/newfswin script and section REL for details).

In Basic i/o mode, the default console font is used.

At the program start, FlagShip takes it default font from your X11 Window manager (e.g. KDE, Gnome etc) or from setup of MS-Windows. You may then at run-time assign and use any other available font by the SET FONT command, or by using corresponding FONT clause in many i/o commands and functions, or by using the `Font{}` class.

For our programming purposes, we need to distinguish between two kinds of fonts: having proportional or fixed pitch. The fixed pitch (monospaced) font (like "Courier" or "Terminal") has the same character width for all characters; we don't need to worry about the character spacing and can handle it by the same way as in textual application.

In proportional fonts (like "Helvetica", "Arial", "Times"), the character width vary, i.e. the "X" or "m" character is wider (needs more pixels) than the "i" or space character. When you display the string chr(153) + "Mi p" on the screen (e.g. by using ?, ??, @... i/o commands), you will see:



The current x/y or col/row display position starts at position marked by (0) above, the next position (after display) is marked by (+) and is reported by the ROW() and COL() function.

For a cross compatibility, and to start the output at 0,0 coordinates (see LNG.5.3.1), the default x/y font alignment is on the top left character frame as marked in the above picture. You may change this alignment by SET ROWALIGN TO...

You may retrieve the font characteristics by using Font class properties (see section OBJ), e.g. m->oApplic:Font:FontFamily reports the used font name/family, oApplic:Font:Size it size in points etc. You may determine the width of a string by using Strlen2pix(chr(153) + "Mi p") for above example, or the similar StrLen2col() or by oApplic:Font:WidthChar()

The font height (same as other font characteristics) is fix and corresponds to the used typeface (or font family) and it size. You only may vary the line spacing displacement by assigning the required amount of pixels (positive or negative) to the global variable `_aGlobSetting[GSET_GN_ROW_SPACING]`, see the source code in *<FlagShip_dir>/system/initio.prg* for details.

For additional information about font handling, see

- LNG.5.3 (difference between terminal and GUI i/o),
- LNG.5.4 (national character support),
- LNG.5.4.2 (national, special and Ansi/Oem chars),
- LNG.5.4.3 (output conversion, PC8 translation)
- OBJ.FONT (font properties),
- SET FONT (font setting and searching for)
- SET GUITRANSL (drawing semi-graphic characters, lines, boxes)
- SET PIXEL, Col(), Row(), Col2pixel(), Row2pixel(), SET GUIALIGN, SET ROWALIGN, SET ROWADAPT, StrLen2col(), StrLen2pix().

A good font introduction is given in *http://www.nwalsh.com/comp.fonts/FAQ/*, many additional information is available on the Web when searching for "typography", for example in *http://www.microsoft.com/*

You may list the available fonts on Unix by xlsfonts command or use any font manager otherwise.

Hint: For minimal porting effort from/to textual i/o, best to use fixed fonts, e.g. simply add

```
SET FONT "Courier", 12
//or: SET FONT "adobe-courier",10
oApplic:Resize(25, 80, , .T.)      // resize screen according to FONT
```

at the begin of your application (with #include "fspreset.fh" according to LNG.9.5) and use the standard row/col coordinates. The application behaves then same as in textual i/o mode. With proportional fonts, you will often get more pleasant look, but will need to consider the font characteristics by adapting the row/col output coordinates in your source. In Linux, best to use the “newfswin” or “newfsterm” scripts for Terminal i/o, see section REL.

5.4 National Character Support

You surely know the confusion: when opening a document (or source code) containing national character set or special characters in different editors (like MS-WinWord, DOS-Word, edlin, Notepad, UltraEdit, vi, Emacs, Nedit, Jedit, Kedit and so on), you may see **different** text. Instead of semi-graphic horizontal line you see A-umlaut, or instead of e-accent you see greek Theta and so on. This is caused by different keyboard and output mapping and/or the used/default editor's **character set**.

For you, as a software developer, it is nothing new that every character in the text document is represented by a binary bit combination. The most text documents and all source codes are byte oriented, i.e. each character in the text is represented by one byte. But the range of 255 possible byte combinations is not sufficient to cover all the human languages, alphabets and additionally also store some special characters like semi-graphic. To allow this, at least two bytes per character are required, such coding is known as Unicode or wide-character set. On the other hand, these two-byte (or sometimes up to six-byte) characters are not so easy to handle as the byte-by-byte text storage and requires special coding/decoding mechanism, slowing the string handling significantly.

> **Note:** We cannot cover here all the aspects and details about the different character sets, but will short explain only the main differences between them. For further information, please consult the literature or Internet. A good overview is for example available on *http://czyborra.com/charsets/codepages.html*, *http://czyborra.com/charsets/iso8859.html*, *http://czyborra.com/utf/* and the links there, as well as on *http://www.microsoft.com/globaldev/reference/default.mspx*, *http://www.microsoft.com/globaldev/reference/cphome.mspx* and so forth.

5.4.1. Different Character Sets

- 1 byte (8bit) character set is known as **ASCII** and is also named OEM, or IBM PC-8 character set. There are different variants of, known as "code pages", for example CP-437 or CP-850 for western chars, CP-852 or eastern Europe, CP-855 or CP-866 for Cyrillic chars etc. The common characters in all ASCII code pages are in the range 1..127 covering the standard Latin alphabet, numbers and some special characters. This character set is used in DOS, Unix and CMD console, remote terminal processing and in the most cases for source-code programming. Bytes 128..255 differs for these different character sets.
- Another 1 byte (8bit) character set is **ISO 8859** and contains a full series of standardized multilingual single-byte coded graphic character sets. It is also known as ANSI char set. The most popular are ISO-8559-1 and ISO-8559-15 (or CP-1252) for western character set, ISO-8559-2 (or CP-1250) for eastern Europe characters and so forth. Only the characters in range 1..127 are compatible to ASCII character set and to other ISO pages, characters 128..255 differs for each character set. This character set is used mostly in graphical environment like X11, MS-Windows, HTML browsers and on some Unix consoles.

- A multi-byte character set containing 2 to 6 bytes per character is known as **Unicode** or UTF-8, UTF-16. FlagShip use this character set internally for GUI and can handle it also for input/output, see further details in section 5.4.5 **Linux note**: the internal GUI translation with environment set LANG=xx_XX.UTF-8 (check by "echo $LANG") does not work for ASCII mode. Use "export LANG=C" or "export LANG=en_EN:ISO-8859-1" for proper translation.

5.4.2. Programming and Use Of National & Special Characters

When editing a text or program source, the most editors behaves similarly for all characters in the range 32 (space) to 126 (tilde). With all other characters outside of this range, you sometimes get anything else than you expect, depending on the editor setting. With other words: WYSIWYG is not always true what-you-see-is-what-you-get or -what-you-expect-to-get :-)

For example, the u-umlaut ("u" with two dots) is represented by chr(129) in ASCII charset = octal 201 = 0x81, but with chr(252) = octal 374 = 0xFC in ISO-8559-1 (Latin-1) character set. So when your keyboard has the "u- umlaut" key and you type "M(u-umlaut)enchen", your text may contain either chr(77,129,110,99,104,101,110) or chr(77,252,110,99,104,101,110) depending on the editor setting and/or the used environment.

Similarly, when displaying the string "M(u-umlaut)enchen" coded in source chr(77,129,110,99, 104,101,110), you may see either "München" or "M• nchen", depending on the used editor or editor encoding.

This apply also for special characters like the semi-graphic IBM-PC8 characters, so coding chr(195,196,197,196,180) which is roughly "|-+-|" can be displayed as "├─┼─┤" or as "ÃÄÅÄ´", i.e. (A-tilde), (A-umlaut), (A-circle), (A-umlaut), (apostrophe) depending on the console/X11/ Windows environment setting.

In FlagShip, you may influence the GUI output by SET SOURCE ASCII for considering strings written in ASCII / PC8 / OEM character set, or by SET SOURCE ISO when the source strings are coded in native ISO/ANSI charset. Additional commands for language support are SET GUITRANSL, SET ANSI, SET DBREAD, SET DBWRITE, SET KEYTRANS, FS_SET("ansi2oem" | "guikeys"), Ansi2oem(), Oem2ansi().

The default i/o setting is based on standard ASCII character set, including the IBM-PC8 extended characters. This assumes the .prg source contain ASCII / PC8 / OEM text and which provides full backward compatibility to DOS based applications (sources and databases) and makes porting easy.

In addition to the default setting, FlagShip supports any special needs by using external, user modifiable translation tables. These are:

a. Screen/terminal output mapping table for Terminal i/o mode is read automatically at start-up of a terminal based application. The default table is named FSchrmap.def and is located in the terminfo directory, see details in SYS.2.4. The table depends on the current

TERM or FSTERM environment variable. User modifiable tables can be specified and loaded via FS_SET("outmap") function.

b. Keyboard mapping table for Terminal i/o mode is read automatically at start-up of a terminal based application. The default table is named FSkeymap.def and is located in the terminfo directory, see details in SYS.2.5. The table depends on the current TERM or FSTERM environment variable. User modifiable tables can be specified and loaded via FS_SET("inmap") function, some tables are already predefined in *<FlagShip_dir>/terminfo/FSkeymap.**

c. Keyboard mapping table for GUI mode is pre-defined in the library and corresponds to the FSguikeys.def file located in *<FlagShip_dir>/terminfo* directory. User modifiable tables can be specified and loaded via environment variable FSGUIKEYS=<filename> at start-up and/or any time later via the FS_SET("guikeys") function. **Linux note**: the internal GUI translation with environment set LANG=xx_XX.UTF-8 (check by "echo $LANG") does not work. Use "export LANG=C" or "export LANG=en_EN:ISO-8859-1" etc. for proper translation.

d. OEM -> Ansi and Ansi -> OEM translation and GUI output table is used for Ansi2oem(), Oem2Ansi() and for output translation in GUI mode, when SET SOURCE ASCII (which is the default when using #include "fspreset.fh", see LNG.9.5). These tables translate the OEM (ASCII, PC8) string to ANSI (ISO, Latin) and vice versa. The default table is pre-defined in the library and corresponds to the FSansi2oem.def file located in *<FlagShip_dir>/terminfo* directory specifying the ISO-8859-1 = Latin1 character set. User modifiable tables can be specified and loaded via FS_SET("ansi2oem") function.

e. User specific sorting table, upper/lower translation and messages is read automatically at start-up of a GUI or terminal based application. The default table is named FSsortab.def and is located in the terminfo directory, see details in SYS.2.6. User modifiable tables can be specified and loaded via FS_SET("loadlang"/"setlang") function, some tables are already predefined in *<FlagShip_dir>/terminfo/FSsortab.**

See also examples western.prg, arabic.prg, slavic.prg, greek.prg and so on, located in <FlagShip_dir>/examples/

5.4.3. String Output Conversion

In Terminal i/o mode, which use Curses with 8bit coding, the output conversion is relative simple:

a) the environment variable FSTERM or TERM specifies the used screen character set
b) the passed string is converted according the used FSchrmap.* table and the environment variable FSTERM/TERM to Curses character
c) the Curses subsystem displays these characters on the screen

In GUI mode, the string conversion for screen output is performed in these more complex steps:

a) when SET SOURCE is ASCII or SET GUITRANS ASCII is ON, the passed string is converted from ASCII to ISO set (still 8bit) via Oem2Ansi() according to the current FSansi2oem.* table
b) when SET GUITRANSL TEXT is ON, the semi-graphical PC-8 characters or theirs mapping are considered according to the FSansi2oem.* table
c) the ASCII or ISO string is converted to Unicode (16+ bit) in that order:
   - when a specific font is assigned to a widget, or widget's font: CharSet() or font:CharSetName() was set, use this character set
   - when a specific font is assigned to the oApplic:Font property, or `m->oAplic:Font:CharSet()` or `m->oApplic:Font:CharSetName()` was set, use this character set. For MessageBoxes, the `m->oApplic:FontWindow` object is used, instead of `m->oApplic:Font` for the default output
   - when the environment variable LANG contain a <dot><charset> specification, use this <charset>. For example "export LANG=`pl_PL.ISO-8859-2`" uses the ISO-8859-2 table for the Unicode conversion.
   - otherwise use Latin1 = ISO-8859-1 (or ISO-8859-15 containing the Euro € sign)
d) When the current or default font in GUI mode is FONT_UNICODE, the Unicode string or glyph is displayed natively, w/o above conversion.
e) display the text by the GUI i/o subsystem (using internally Unicode)

In Basic i/o mode, none input/output conversion is done, the passed string is displayed "as is" via the standard i/o system.

5.4.4. Character Input Conversion

The keyboard input is in Terminal and GUI mode asynchronous, which means the user key press is stored in the keyboard or event buffer and can be retrieved by the application later via INKEY() and associated commands or functions like InkeyTrap(), ACCEPT, INPUT, WAIT etc. The same is valid for mouse movement or button press in GUI mode, whereby many of the mouse (and keyboard) actions may be handled by the GUI subsystem automatically. In Basic i/o mode, the input is handled synchronously by the standard i/o system at the time of invoking Inkey().

A key press triggers following actions:

a) The keyboard sends a scan code to the system (Unix or Windows)
b) This scan code is translated to one byte or to sequence of multiple bytes by system internal tables according to the keyboard specification. On Unix, these tables are user modifiable, see e.g. "man 5 keymaps", "man ttymap", "man loadkeys", "man mapkey", "man mapchan", "man 8 getkeycodes", "man stty" and section SYS.2.1 in this manual
c) On X11 system, following tables are considered by the X system as well: /etc[/X11]/XF86Config, ~/.Xdefaults or ~/.Xresources, ~/.Xmodmap, see e.g. "man xmodmap", "man X11", "man XFree86", "man setxkbmap"
d) The system passes the byte or sequence of bytes to the application. Depending on the used i/o mode, FlagShip's run-time system converts this to Inkey-equivalence-number (see manual appendix and inkey.fh):

- In Terminal i/o, the Curses subsystem converts the byte or sequence of bytes (Esc-sequence) to mnemonic token or character according to the used FSTERM or TERM environment variable. This conversion can be changed by the *<FlagShip_dir>/terminfo/Fstinfo.src* file, see also section SYS.2.2. The incoming character can additionally be converted by user modifiable FSkeymap.* table, see details in LNG.5.4.2.b above and in SYS.2.5
- In GUI i/o mode, the from system receiving character or sequence is translated via the FSguikeys.* table to Inkey number, see also section LNG.5.4.2.c above. This table is user modifiable and loaded by FS_SET("guikeys",file). You may additionally use SET KEY-TRANSL to influence the conversion.
- In Basic i/o mode, none additional conversion is done, the Inkey number is taken from the system's input buffer.

5.4.5. Unicode

FlagShip supports also native Unicode in GUI mode.

Unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems. The Unicode Standard contains a repertoire of more than 110,000 characters covering 100 scripts. The standard is maintained by the Unicode Consortium and is defined in ISO/IEC 10646, you may download it from ISO via http://standards.iso.org/ittf/licence.html

Unicode can be implemented by different character encodings. The most commonly used encodings are UTF-8 and UTF-16. **UTF-8** uses one byte for any ASCII characters chr(1..127), which have the same code values in both UTF-8 and ASCII encoding, and up to four bytes for other characters (like glyphs). **UTF-16** (or UCS-2) uses a 16-bit code unit (two 8-bit bytes) for each character but cannot encode every character in the current Unicode standard. See further details e.g. in http://en.wikipedia.org/wiki/Unicode

In FlagShip, Unicode is used mainly to display and edit glyphs in text and database fields. A **Glyph** is a graphic symbol that provides the appearance or form for a character. A glyph can be an alphabetic or numeric font or some other symbol that pictures an encoded character. It is often used e.g. in Asian languages such as Japanese, Chinese, Hindi, Korean, etc.

To remain compatible to sources and databases created in ASCII (1 byte per character), FlagShip stores Unicode text in UTF-8 encoding, but accepts input in both UTF-8 and UTF-16 encodings.

To display Unicode text and glyphs, the current or default font needs to be set to Unicode, eg. by oApplic:Font:CharSet(FONT_UNICODE) or SET GUICHARSET FONT_UNICODE or the user's font object oMyFont:CharSet(FONT_UNICODE). It is supported by ?, ??, Qout(), @..SAY and

output of MemoEdit(), @..GET / READ. In Linux, you may need to set Unicode font, e.g. SET FONT "mincho" as well.

To edit Unicode, SET MULTIBYTE ON must be set as well. It is supported so in Inkey(), InkeyTrap() and any input based on Inkey(), such as @..GET/READ, MemoEdit() etc. When entering Unicode glyph w/o FONT_UNICODE and/or without SET MULTIBYTE ON, the UTF-8 sequence chr(128..255) may be converted to OEM or ASCII or ISO for national charset, or to semi-graphics by SET GUITRANSL or SET SOURCE or FS_SET() commands/functions, which of course will corrupt the glyph chr() sequence and hence may display/store garbage. When the SET MULTIBYTE ON is active or the MULTIBYTE clause in @..GET is set, all automatic conversions are disabled.

Predefined text needs to be stored in UTF-8 encoding (usually one byte for ASCII chr(1..127) and up to four bytes each chr(128..255) for Unicode). In the FlagShip library, there are helpers for handling Unicode, e.g. Utf16_Utf8() to convert UTF-16 number to UTF-8 string, Cp437_Utf8() for converting ASCII PC-8 CodePage 437 to UTF-8 etc, see section FUN for details and example in *<FlagShip_dir>/examples/unicode.prg*

There are some **disadvantages** when using Unicode: only low ASCII characters chr(1..127) are handled as such. National characters chr(128..255), e.g. Umlauts, PC-8 semi-graphics etc. cannot often be displayed directly (but via multi-byte sequence), since UTF-8 may use these chr(128..255) for multi-byte glyphs. Hence the upper/lower conversion will work only for the lower character set chr(1..127), searching for special character in string may fail, and the buffer or field for editing needs to be extended in the application to fit all required glyphs (e.g. 10 Japanese/Chinese glyphs requires buffer or database field of 30 characters). For that reason, use Unicode only when really required.

5.4.6. Case Studies

1. The .prg sources are coded in PC-8 (ASCII,OEM) character set, or contain characters 1..127 only, databases are DOS compatible and contains PC8 (ASCII) code, the default:
   a. Terminal i/o: the environment variable TERM or FSTERM should be set accordingly to your used environment (e.g. FSansi, fsansi, fslinux etc.), see details in section REL. For a proper display of special chars, best to use the "newfscons", "newfswin" or "newfswin" scripts which sets the environment accordingly.
   b. GUI i/o: to proper display the ASCII strings in ISO mode, use SET SOURCE ASCII (or #include "fsprest.fh"), which is similar to Oem2Ansi() sting translation done manually. You may disable this translation by SET SOURCE ISO or ANSI. To draw PC-8 semi-graphic characters, lines and boxes, use SET GUITRANSL TEXT/BOX/LINES ON and/or explicitly by @..DRAW or GuiDrawLine().
2. The .prg sources are coded in PC-8 (ASCII, OEM) character set and contain also national characters 128..255, coded by an explicit ASCII char() value like this: cString := "M"+chr(129)+"nchen" or "M\0201nchen" to display "München" = Munchen with u-umlaut:

a. Terminal i/o: same as (1a) above. If different than the default PC8 (CP437/850) character set is used, apply the corresponding input/ output char translation by FS_SET("outmap") and/or FS_SET("intmap")
b. GUI i/o: same as (1b) above. If different than the default PC8 (CP437/850) character set is used, set the conversion table via FS_SET("ansi2oem"). Compare e.g. FSansi2oem.def and FSansi2oem.*

3. The .prg sources are coded in native ISO (Ansi) character set and contain also national characters 128..255 coded inline, e.g. as "München" or "M• nchen". Note: the semi-graphic characters are usually not available in native ISO mode.
   a. Terminal i/o: similar as (1a) above, but set the ISO terminfo, e.g. FSsun, fslinux etc. by using the TERM or FSTERM.
   b. GUI i/o: use SET SOURCE ISO (after #include "fsprest.fh" if used). The SET GUITRANSL TEXT conversion has usually no effect, but SET GUITRANSL BOX/LINES and @..DRAW or GuiDrawLine() work fine.

4. When the .prg source strings are coded "inline", e.g. "München" or "M• nchen", the conversion depends on the byte representation of the special character - if the u-umlaut is chr(129) == ASCII, or chr(252) == ISO code. Simply follow suggestion 5.4.6.2 or 5.4.6.3 above.

5. When you are using ASCII program mode, the data are stored also in this backward compatible mode in the database. When you instead want to store ISO data in the database, and your current setting is ASCII, use SET ANSI ON or SET DBREAD/DBWRITE ANSI/ISO. When your current source setting is ISO/Ansi, and you want to keep the database backward compatible, you may translate the database content by SET ANSI OFF or SET DBREAD/DBWRITE ASCII/PC8.

See also examples western.prg, arabic.prg, slavic.prg, greek.prg and so on, located in *<FlagShip_dir>/examples/*

6. The GET System

The GET system of FlagShip allows the programmer, to design his own custom-built GET/READs. The system consists of the (low-level) GET class and the (high-level) **modifiable** access routines found in *<FlagShip_dir>/system/getsys.prg*.

Both the high and low level systems are already included in the FlagShip library, so if modifications are not necessary, the system is automatically available during the compiling and linking phase.

The GET system of FlagShip is functionally compatible to Clipper 5.x and, using commands @..GET and READ, also to all Clipper'87 or other xBASE programs.

Usually, the usage of the command @...GET and READ is the most common and comfortable way to access the whole system. The FlagShip preprocessor (see section FSC) will translate these commands automatically to the equivalent class definitions and function calls.

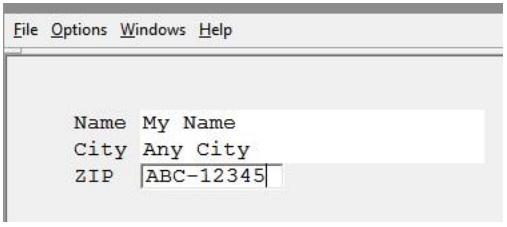
This chapter covers the basic information of the GET system. For a further description and additional options, see commands @..GET and READ in section CMD. The experienced programmer who wishes to tune the input system to his own needs, will find in-depth information in section OBJ and in the interface program *<FlagShip_dir>/system/getsys.prg*

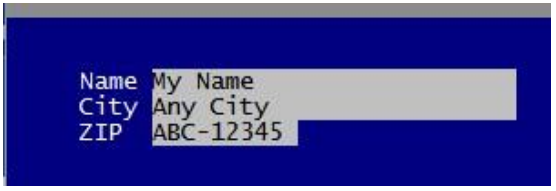
The @...GET Command

To activate full-screen data input (see also chapter 5), the variable (or database field) storing the user entry, and the position and size of the input field has to be defined:

```
SET FONT "Courier", 12
LOCAL name := space(25), city := space(25), zipcode := space(10)
SET COLOR TO "W+/B,N/W"
CLS
@ 2,5 SAY "Name" GET name
@ 3,5 SAY "City" GET city  VALID LEN(TRIM(city)) > 2
@ 4,5 SAY "ZIP " GET zipcode PICTURE "!!!!!!!!!!" VALID !EMPTY(zipcode)
READ
```

The above example defines three entry fields. Two of them are validated for the correct entry, the entry of the zip code is automatically converted to uppercase.





The @..SAY..GET command outputs these three fields, including the additional description in the specified colors. The description will be given intensive white on blue background, the entry fields are colored inverse white. Many additional features, including pre-validation, formatting etc. are available, see (CMD) @..GET.

The FlagShip preprocessor additionally creates for each of the @...GET commands a GET object and adds it into a global array GETLIST. Using a LOCAL (or PRIVATE, STATIC) `GETLIST := {}` declaration allows you to create GET/READs, nested to any level.

The READ Command

In the example above, the READ command enables the user input in the three input fields. The user may move from field to field using the cursor-up and cursor-down keys or edit the actual field with e.g. the cursor ← → keys, BACKSPACE, DELETE, INSERT and so on. More editing and navigation keys are available, see (CMD) READ. The correct data entry will be automatically checked by the defined VALID condition.

The FlagShip preprocessor translates the READ command to the READMODAL(getlist) UDF call. If a LOCAL array GETLIST was declared, a nested READ is executed.

The source code of the READMODAL() function available in getsys.prg (directory *<FlagShip_dir>/system*). The inner behavior of the READ command including the pre- and post-validation checking is thus freely modifiable.

An advanced programmer may also change the low-level GET/READ behavior by inheriting the GET class into his own subclass. See details in section LNG.2.11 and OBJ.2.

7. The TBrowse System

The FlagShip TBROWSE system allows the user to browse and/or modify tables, i.e. databases or arrays. It is similar to the (older) DBEDIT() function and based on the TBROWSE and TBCOLUMN object class.

An advanced programmer may also change the low-level TBROWSE and TBCOLUMN behavior by inheriting the class into his own subclass. See details in section LNG.2.11 and OBJ.3.

This chapter covers the basic usage of the TBrowse system. For detailed information, see section OBJ.3. It is also used in DBEDIT(), available in source code in the *<FlagShip_dir>/system/dbedit.prg* file, which may be a good source of a practical TBROWSE usage.

Creation and Usage of TBrowse Objects

By using one of the class-definition functions TBROWSENEW() or TBROWSEDB() or TBROWSEARR(), a new TBrowse object can be created. The **TBROWSEDB()** is specially designed for browsing databases, the former has more generic capabilities. The **TBROWSEARR()** is designed to browse arrays.

The columns of the Browse systems are described by TBColumn objects, defined with TBCOLUMNNEW(). At least one column needs to be assigned.

Horizontal and vertical movement in the browsed table is specified by code blocks assigned to the corresponding instance variables. At least the :SkipBlock must be assigned, assigning :GoTopBlock and :GoBottomBlock is recommended for speed.

```
mybrow := TBROWSENEW  (5,0, MAXROW()-1, MAXCOL())
mycol1 := TBCOLUMNNEW ("Cust.Name", {|| name})
mycol2 := TBCOLUMNNEW ("Address", {|| city + " " + zipcode})

mybrow:ADDCOLUMN (mycol1)
mybrow:ADDCOLUMN (mycol2)
mybrow:SKIPBLOCK := {|par| myskip(par)}
```

Additional settings, like different color specifications for any column, vertical column separators etc. are available.

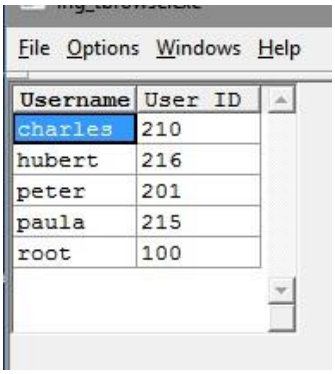
Stabilizing the System

The main advantage of the TBrowse system compared to e.g. ACHOICE() is the asynchronous movement in the table and the actual data being displayed. This allows the data to be prepared "in the background" and to display only the required part of it.

Every time user movement is requested, the system gets into an "unstable" condition. When the database (or array) movement and the data display is finished, the TBrowse system becomes "stable". If any user key is pressed in the meantime, the stabilize and/or output process may be interrupted for the next required action. Example of a small, simple TBrowse program:

```
LOCAL users := { { "charles", "210" }, ;    // the array to browse
                 { "hubert ", "216" }, ;
                 { "peter  ", "201" }, ;
                 { "paula  ", "215" }, ;
                 { "root   ", "100" }  }
LOCAL element := 1                          // index in the array
LOCAL key, mycol
LOCAL mybrow := TBROWSENEW (0,0, 7,20)      // create TBrowse object
mycol := TBCOLUMNNEW ("Username", {|| users[element][1]})
mybrow:ADDCOLUMN (mycol)
mycol := TBCOLUMNNEW ("User ID ", {|| users[element,2] })
mybrow:ADDCOLUMN (mycol)
mybrow:SKIPBLOCK := { |input, oBrowse, temp| temp := element, ;
                      element := MAX(1, MIN(LEN(users), ;
                      element + input)), element - temp }

mybrow:COLSEP  := " | " ; mybrow:HEADSEP := "─┼─"
CLS                                         // or: CLEAR SCREEN
WHILE (.T.)
   WHILE (!mybrow:STABLE)                   // (re)build screen,
      mybrow:STABILIZE()                    // wait for stabilizing
      IF NEXTKEY() != 0                     // optional:
   EXIT                                     // manage async.input
      ENDIF
   ENDDO
   key := INKEY(0)                          // get key pressed
   DO CASE
   CASE key =  19                           // left
      mybrow:LEFT()
   CASE key =  4                            // right
      mybrow:RIGHT()
   CASE key =  5                            // up
      mybrow:UP()
   CASE key = 24                            // down
      mybrow:DOWN()
   OTHERWISE
**    RETURN (element)                      // other key termina-
      QUIT                                  // tes the browsing
   ENDCASE
ENDDO                                       // system is unstable now
```



8. The Open C System

The open architecture of FlagShip use 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,
- Included C inlines within the regular .prg code, using the #Cinline directive,
- Invoking the standard FlagShip functions from any C program.
- Modification of the intermediate C code produced by the FlagShip compiler.

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

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

Modification of the produced C code is not very common, but possible by very experienced C programmers. Such modified code will loose the high level of compatibility, but porting to other Unix (or Windows) systems can be done on the C FlagShip level.

! **Warning:** Programming in the C language allows you nearly unlimited access to the whole Unix or MS-Windows 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.

This chapter covers the basic usage of the FlagShip's Open C System. For detailed information, see section EXT.

8.1 The Extend C System

The 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 or MS-Windows 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.

Compared to stand-alone C or Clipper's Extend C program, very few modifications and rules are necessary:

- Include the FlagShip's extend file FSextend.h
- Use the macro FlagShip(fn_name) instead of the common function C declaration by fn_name() or CLIPPER fn_name().
- After internal variable definitions, first call the function FSinit() to receive the parameters from your FlagShip application.
- For parameter passing, use the Extend System functions _parxxx(). Preferably, check the incoming parameters for type and validity.
- Pass the return values to FlagShip using _retxxx() functions, pass other values to .prg using _storxxx().
- Return from the C program and pass control to FlagShip using FSreturn.

Example of a small Extend C program to rotate a string from left to right:

```
/*** File strrot.c                                                ***/
/*** Call it from FlagShip program: new = str_rotat (old)         ***/

#include <FSextend.h>
#include <string.h>
FlagShip (str_rotat)                              /* declare name */
{
   int  lng, left, right;                         /* internal C   */
   char *str, temp;                               /* variables    */
   FSinit () ;                                    /* init params  */
   if (PCOUNT != 1 || _parinfo (1) != CHARACTER) { /* check param1 */
      _retc ("");                                 /* return error */
      FSreturn;                                   /* and exit     */
   }
   str = _parc(1);                                /* get string-  */
   lng = strlen (str);                            /* ptr passed   */
   if (lng > 1) {
      for (left=0, right=lng-1; left < right; left++, right-- ) {
         temp = str[left];
         str[left]  = str[right];                 /* swap chars   */
         str[right] = temp;
      }
   }
   _retc (str);                                   /* push to FS   */
   FSreturn;                                      /* exit C to FS */
}
```

Note: The program works on the internal string copy passed from FlagShip, expanding the string length is therefore NOT allowed. For more examples see in section EXT.

You may compile this program together with your application entering:

```
$ FlagShip test.prg strrot.c
$ a.out
```

8.2 Open C API

This interface is designed for any C programmer. It allows you to access nearly the whole system, including the local and dynamically scoped FlagShip variables, or standard FlagShip and C functions trough any C program. By using such a interface, access from other programming languages (Fortran, Cobol, Pascal) is also possible. If you wish, you may create the whole application in C; the only pre-requirement is, that the main module must be a FlagShip procedure or function.

The Open C API is commonly used for programming of RDDs (if you don't like to program it into .prg), other system drivers, or for very often called functions to increase theirs execution speed. In fact, the most of FlagShip standard functions are programmed by using Open C API. See details in the section EXT.4.4, as well as examples in the files *<FlagShip_dir>/system/CB4rdd.tar.Z* or .../ascirdd.tar.Z

8.3 The Included C Code (Inline-C)

Using C code between the directives `#Cinline` and `#endCinline` allows you to include it directly into FlagShip's .prg source file. During the compiler phase, all statements between these two directives will be passed directly into the intermediate C code. You may directly access all the TYPED FlagShip variables, or the untyped ones using connecting functions.

The typical usage of included C:

```
** File test.prg
LOCAL angle
LOCAL_DOUBLE radian, sinus, cosine
INPUT "Enter angle 0...360 degree: " TO angle
radian := 2.0 * 3.1415926535 * angle / 360.0

#Cinline
   sinus   = sin(radian);              /* use std. C math library */
   cosinus = cos(radian);              /* use std. C math library */
#endCinline

? "sin(" + ltrim(str(angle)) + ")=", sinus, ;
  "cos(" + ltrim(str(angle)) + ")=", cosinus
```

The previous string rotation example from 8.1 can be also defined as inline C code. Because of the additional variable declaration within the C part, and the include's for the C part, the whole C routine has to be enclosed in curly brackets {...}:

```
** File test.prg
LOCAL myStr as character
myStr := "abcdefgh"
? "original string:", myStr
#Cinline
{
#include "FSopenc.h"                          /* VAR_* macros */
   int  left = 0, right;                      /* internal C   */
```

```
   char *str, temp;                                  /*   variables  */
   str = VAR_CHR( VAR_NAME_LOCAL(mystr) );           /* ptr to LOCAL */
   for (right = strlen(str)-1; left < right; left++, right-- ) {
      temp = str[left];
      str[left]  = str[right];                       /* swap chars   */
      str[right] = temp;
   }
}
#endCinline
? "swapped string:", myStr
```

Note the different string indices in C (0...strlen(str)-1) compared to the FlagShip/xBASE convention (1...LEN(str)). Also note that the accessed FS variables in the C part via VAR_NAME_LOCAL() must be given in lower case, and need to be assigned to C type via VAR_*() before using. More examples and a full description of using in-line C code are available in sections EXT.3 and EXT.4.

You may also use the CALL command, which translates FlagShip variables to the C equivalent automatically:

```
** File test.prg
#Cinline
#include <string.h>                                  /* std. Unix    */

void RotateStr (str)                                 /* in/output,   */
unsigned char *str;                                  /* see CALL cmd */
{
   int  left = 0, right;                             /* internal C   */
   char temp;                                        /* variables    */
   for (right = strlen(str)-1; left < right; left++, right-- ) {
      temp = str[left];
      str[left]  = str[right];                       /* swap chars   */
      str[right] = temp;
   }
}
#endCinline

FUNCTION main ()
LOCAL mystr
mystr := "abcdefgh"
CALL RotateStr WITH mystr
? "swapped string:", mystr

** Compile: $ FlagShip test.prg -na -Mmain
```

Note: C functions must be declared at the beginning of the .prg file. The required C include files should be put there as well.

For a comparison, the same string rotation program, fully coded in the FlagShip language:

```
** File test.prg
LOCAL mystr, temp
LOCAL_INT left := 1, right         // typed vars increases the speed
mystr := "abcdefgh"
right := LEN (mystr)
```

```
DO WHILE left < right
   temp  := ASC(SUBSTR(mystr,left,1))
   STRPOKE (@mystr, left++, ASC(SUBSTR(mystr,right,1)))
   STRPOKE (@mystr, right--, temp)
ENDDO
? "swapped string:", mystr
```

8.4 Modifying the intermediate C Code

Because the modification of the produced C code from the FlagShip compiler is not very common but may be interesting for very experienced C programmer, the description will be given in detail in the section EXT.

9. Program and Data Compatibility

This chapter describes possible differences to compatible database systems for MS-DOS (such as Clipper or dBASE). A full description for porting your application will be given later in the section SYS.

9.1 Program compatibility

The FlagShip language is in semantics, syntax and operation highly compatible to programs written in dBASE, Foxbase, FoxPro or in the Clipper language. Therefore, porting such applications from MS-DOS to Unix or MS-Windows is extremely easy.

> **Note:** please keep in mind, FlagShip is an **independent** programming language in its own right, and neither the Clipper nor the dBASE or Fox system (all of which are often not fully compatible to their own previous releases, work other than the documentation says and have several documented or undocumented anomalies). FlagShip supports very different operating systems from the MS-DOS. Our goal is to maximize the portability and minimize your expense as much as possible. The remaining, very small differences in FlagShip are system dependent, and can be handled in your program (see below).

The programmer should only consider the specific differences between the two operating systems, for more detailed information see chapters LNG.1.3 LNG.9.3 and section QRF, SYS.

9.2 Data compatibility

FlagShip fully supports the .dbf and .dbt (or .fpt) database structure without any modification or conversion. Indices from MS-DOS xBASE languages (like dBASE .ndx. .mdx or Clipper .ntx or Foxpro .cdx, .idx) are not supported by the default DBFIDX driver, FlagShip uses optimized B-tree indexes in **.idx** files instead which supports automatic integrity checking, files >> 2GB and more. For the first time, you need to reindex by INDEX ON..TO.. Nevertheless, other replaceable database and index drivers are already available (see section RDD), additional drivers may follow in the next release or by Third Party vendors.

The .mem files are compatible to Clipper (5.x and '87) and most of the other xBASE systems per default. FlagShip reads (RESTORE FROM) both the older (rel 3.x) and the new *.mem format and produces (SAVE TO) always in the new format. The *.mem files are now portable to/from DOS or to/from other Unix or MS-Windows systems, just like *.dbf and *.dbt or *.fpt files. Because FlagShip can store arrays (Clipper doesn't), it is up to you to enable this option using FS_SET("memcomp", .F.); the full .mem file portability to DOS may be lost thereafter. Screen variables are not compatible to Clipper or xBASE at all.

You may use the same data files (like databases, memo files etc.) on Unix and MS-DOS. When transferring these binary files from other system, you must use binary transfer or protocol,

instead of text transfer. You cannot use index files from DOS - however FlagShip .idx indices are cross compatible for different platforms (Windows, Linux, Unix). For copying the .prg and other source files, you may use text or binary protocol.

9.3 Differences to Clipper and other xBASE

The way FlagShip operates has some similarity to Clipper (CA, formerly Nantucket), which was designed for the 16bit MS-DOS system only. FlagShip is still more flexible, designed mainly for a multi-user purposes, and for 32/64bit systems. Because of the translation into the C language, no "hidden p-code" is produced, and this C source may by modified by yourself if required, see section EXT.

FlagShip has no problems or limits with "not enough memory". Both Unix and 32/64bit MS-Windows uses virtual memory management, so your programs may be as large as 2-4 Gigabytes (this is 6000 times 640 Kbytes). Because of this it is no longer necessary to use complicated overlays to run a large application. Also, any other program may be called directly by using the RUN statement. Note: the real size limit of your application is restricted only by the "swapspace" area of your disk, the size of which is often fix set during Unix/Windows installation, see also section SYS.

To ensure forward and backward compatibility between FlagShip and Clipper, the Extend System of Clipper is included into FlagShip as well. Only very slight modifications are necessary, see LNG.8 and section EXT. It is very easy to embed C code written to your special needs directly into your FlagShip (or Clipper) program, rather than modifying the generated C code. Nevertheless, FlagShip's Open C System gives you the option of doing this.

Notable differences to dBASE and FoxBase are listed in the Appendix. An additional compatibility to FoxPro is available by using the -fox compiler switch.

Notable extensions and differences to Clipper and MS-DOS:

- Unix file names are case sensitive, but FlagShip will optionally convert them automatically to the Unix convention (see FS_SET() and example below in chapter 9.4). You may convert old files in upper/mixed case to lowercase by files2lower script, see section FSC.6.3
- Clippers .NTX indices cannot be used. FlagShip's .idx indices must be created on the target system using INDEX ON...TO. See LNG.2.1, LNG.4.5 and (CMD) INDEX
- MS-DOS (Clipper's) object files and libraries cannot be used for Unix, nor MS-Windows based FlagShip, the .prg or .c sources must be recompiled on the target system. Third party libraries are usable only if they are ported to Unix or 32/64 MS-Windows by FlagShip, or compiled by the same C compiler, or if available in source code. See section FSC.
- Be careful of differences in system commands, if used in RUN (e.g. RUN ls -la * in Linux instead of RUN DIR *.*). See (CMD) RUN and the "man" pages of your Unix system.

- FlagShip produces spooled printer output by default, but it may be fully deactivated on special request according to SYS.2.7. To use direct output to the device driver, use e.g. SET PRINTER TO /dev/lp0. See LNG.3.4 and (CMD) SET PRINTER. In GUI mode, you may select the driver via common printer dialog and pass the output directly to it by using PRINTGUI() function.
- Screen variables of type "S" are used in FlagShip for SAVESCREEN() variables, instead of the var type "C" in Clipper. Converting functions, e.g. SCRDOS2UNIX and SCREEN2CHR are available (for Terminal i/o) to be able to use DOS created screens or to store the screen content into dbf or dbt fields. See LNG.2.6.4 and FUN.
- Binary 0 (represented by CHR(0)) normally terminates the string in the C language. FlagShip supports embedded chr(0) automatically or on request for most string operations. See LNG.2.6.5.
- File attributes (used e.g. in the ADIR() or DIRECTORY() function) of Unix (drwxrwxrwx) differ to the MS-DOS ones (a,d,s,h,v). See LNG.3.1 and FUN.
- For portability avoid using the [Alt]+[Fkey] combinations, since they are often hard wired in Unix or Windows system GUI. See LNG.5. 2 and section SYS.
- Clipper's .clp and .lnk files are not supported, since FlagShip's command line entry is more powerful, see section FSC. The usage "make" utility is supported, the "fsmake" tool will create the Makefile semi- automatically for you. See section FSC.
- The program interruption key is ^K, the debugger is activated with ^O. Both keys are redefinable using FS_SET(). See sections FSC and FUN.
- Additional settings using FS_SET(...) are available, e.g. to run in developer mode with more warnings, automatic file conversions etc. See sections QRF, CMD and FUN.
- Many additional FlagShip commands and functions, enhanced the language, are available.
- Warning: avoid to copy Clipper's own .ch header files into your working directory, especially when the same file name exist in FlagShip's include directory. FlagShip's .fh header files (and it .ch copy) provides extended functionality. For special definitions, best to use own .ch or .fh files.
- To minimize porting effort, add these statements at the begin of your main module:

```
#ifdef FlagShip
   #include "fspreset.fh"
   SET FONT "courier"
#endif
```

- Possible differences in syntax or functionality are noted and explained in the reference part, sections CMD, FUN, OBJ, PRE and EXT.

9.4 FlagShip Extensions

To support the full program portability between DOS and Unix/Windows, we have added into the FlagShip language the following extensions:

1. Public FlagShip: your application may decide automatically and online, on which operating system it is actually running, if you use the reserved PUBLIC variables CLIPPER and FLAGSHIP:

   If the application is compiled by FlagShip, the public variable FLAGSHIP will be automatically set to "true", the variable CLIPPER is unchanged as "false". If you recompile it with Clipper, FLAGSHIP variable remains "false" and CLIPPER variable will be set by the compiler to "true". Running the same program with other xBASE dialects sets both public variables to .F.

	Unix/Windows	DOS
public FlagShip	.T.	.F.
public Clipper	.F.	.T.

2. **#ifdef FlagShip**: is the preferable, more comfortable way to keep compatibility with all Clipper 5.x programs. The FlagShip preprocessor defines automatically #define FlagShip as true, so the usage of

```
#ifdef FlagShip           /* "FlagShip" is case sensitive */
   RUN ls -la *.* | less  // or other FlagShip statements
   FS_SET ("lower", .T.)
#else
   RUN DIR *.*            // or other Clipper statements
#endif
```

   is very comfortable to code different system options (see example in getsys.prg). The decision which code segment to compile is done by the preprocessor, the other part (like the Clipper's one in a FlagShip program) will be not included in the translated code at all. See also section PRE. To determine difference between FlagShip for MS-Windows and Unix/Linux, use

```
#ifdef FS_WIN32
   RUN dir *.exe        // MS-Windows statements
#else
   RUN ls -la           // Unix/Linux statements
#endif
```

3. **FS_SET() functions**: the Unix system and FlagShip offer you more possibilities than in MS-DOS. You can set some additional switches, compared to standard SET ... commands. To insure the compatibility to Clipper, all of these are included in the FlagShip FS_SET function. For the DOS program, you only include a dummy empty function (or link the compiled fstodos.prg program) to satisfy the linker. See more in section FUN.

4. 8-bit support: FlagShip accepts the complete IBM-PC character set in the source code, during the run time process as well. The only requirement is the support of the 8-bit-set from your Unix system and terminal, see more in section SYS and newfs* scripts in section FSC.6.3 There are no limitation in Windows 32/64bit.

5. Multiuser/Multitasking: additional to DOS networking, FlagShip supports also the Unix and Windows specific multiuser and multitasking environment. For your convenience, FlagShip uses the same statements for this as Clipper/dBASE does for the network support. In addition to, FlagShip also supports **fully automatic** locking, see more in chapter LNG.4.8.

6. Data security: FlagShip automatically supports all of Unix access rights and checks the index integrity. See chapters LNG.3.1 and LNG.4.5.

7. Error System: FlagShip offers you two different error systems, including an extended, almost Clipper 5 compatible, object oriented error system, see sections FSC and OBJ.

8. Open C System: the programmer has five options to include his C programs into a FlagShip application (see chapter 8 and section EXT):

   - using the nearly Clipper compatible Extend System,
   - coding C programs directly in the .prg program, using the #Cinline directive,
   - using the CALL command as interface to C,
   - programming in the C language with access to the FlagShip library functions,
   - directly merging or modifying the intermediate C code produced from the FlagShip compiler.

9. FlagShip supports, in addition to all default Clipper and xBASE variable types, **typed variables** (in the CA/VO syntax) as well as C-like typed, which significantly increase the program stability, the execution speed and allow you direct data exchange between the .prg and inline C program part. See LNG.2.6.1 and section CMD.

10. As opposed to Clipper 5, FlagShip allows exiting from a DO WHILE or FOR loop using BEGIN SEQUENCE...BREAK...END in the same manner as Clipper'87. See LNG.2.5.3 and CMD.BEGIN.

11. As opposed to Clipper 5, FlagShip supports **user-defined objects** and classes (OOP), compatible to the VO syntax. You may also modify the behavior of any standard class (e.g. Get, Tbrowse, Error, DbServer, DataServer, DbfIdx) by inheriting it into your own sublass. See LNG.2.11.

12. Nested GET/READ: the GET/READ system may be nested to any level using LOCAL GETLIST := {} within a UDF. The GET system is fully user modifiable, using the file *<FlagShip_dir>/system/getsys.prg*. See LNG.6 and CMD.READ.

13. Character Mapping: FlagShip supports special handling with different character sets using external mapping tables for the screen output and/or keyboard input. See chapters LNG.5.1.4, LNG.5.2.5, section SYS and functions FS_SET ("outmap" and "inmap").

14. Individual Sorting: to support different human languages, commands like ASORT() etc., are on request controlled by external sorting tables from ASCII files. See chapter SYS and FS_SET ("loadlang").

15. Individual Index Searching using the SEEK EVAL command.
16. Extended Color Manipulation using the SETSTANDARD, SETENHANCED and SETUNSELECTED commands.
17. Printer output is by default done to external file and using the Unix and Windows **spool** system, so there are no collisions in multiuser / multitasking environment. The printout may be activated directly or outside the application. Additional redirections and pipes are supported. See chapter LNG.3.4.
18. On special request, you may in Linux deactivate the Curses initialization and the creation/use of the default spooler file. See details in SYS.2.7.
19. File system: In Unix, there is no equivalent to the MS-DOS drive selector (like C:) in the path specification, but FlagShip can substitute it automatically to an Unix directory using the environment variable x_FSDRIVE (see LNG.3.2, LNG.9.5 and section FSC). The conversion of "\" to "/" within a path will be done automatically. You may enable the character letter case translation of file names and/or path names using FS_SET() functions.
20. Many additional FlagShip commands and functions, which enhance the language, are available. See section QRF, CMD, and FUN.

9.5 Keeping compatibility with DOS programs

Using the extended options of FlagShip, it is easy to keep compatibility between the DOS and the Unix and 32/64bit Windows application. It is important for maintenance purposes to be able to maintain **one** common set of source code for both DOS and Windows/Unix.

Example of a fully DOS/Unix/Windows compatible application, normally no other modification necessary:

1. Include the following statements into the main .prg module:

```
*** main module, remains fully compatible to Clipper 5.x ***
*
#ifdef FlagShip
   #include "fspreset.fh"            // the preferred method
#endif
```

- or - insert your preferred definitions as needed:

```
#ifdef FlagShip                      // automatically defined in FS
   FS_SET ("lower", .T.)             // convert files to lower case
   FS_SET ("pathlower", .T.)         // paths and drives to lower case
   FS_SET ("translext", "ntx", "idx") // search for .idx instead of .NTX

   SET SOURCE ASCII                  // use PC8/OEM/ASCII character set
   SET FONT "courier", 10            // use fixed fonts in GUI
```

```
   oApplic:Resize(25,80,,.T.)        // resize GUI window accord.to font
   SET GUITRANSL LINES ON            // allow draw lines in GUI
   SET GUITRANSL TEXT ON             // allow draw semi-graphic in GUI

   IF GETENV ("C_FSDRIVE") == ""     // only if drive letter C: is used
      ? "set C_FSDRIVE env.variable first"
      QUIT
   ENDIF
#endif
```

That's all there is to it! The rest of the applicat. (all .prg modules) remain unchanged, e.g.

```
SET PATH TO C:\test\Data;..\Xyz\ABC
SET DEFAULT TO ("D:\other\Data")
IF .not. FILE("XYZ.NTX") ...
```

2. Set Unix environment variable(s) to substitute the **DOS drive** letters prior to the execution of the a.out, if drive letters are used:

```
$ C_FSDRIVE=/usr/data1 ; export C_FSDRIVE   (if C: or c: letter used)
$ D_FSDRIVE=/usr/data2 ; export D_FSDRIVE   (if D: or d: letter used)
```

9.6 Porting to Unix/Linux step-by-step

See section LNG.9.7 below for porting hints for Windows.

1. Copy (binary!) your sources (*.prg, *.fmt) and data (*.dbf, *.dbt, *.lbl, *.frm) from DOS to Linux. If the file names are in uppercase or mixed case now, you may convert them to the usual Unix/Linux lowercase by "files2lower", see also fsman LNG.9 for details

2. At the begin of your main module, add the statements

```
   #include "fspreset.fh"
   SET FONT "courier"          // use fixed fonts
   //or often better: SET FONT "adobe-courier",10
   oApplic:Resize(25,80,,.T.)  // resize GUI window accord.to font
   // SET GUICOLOR ON          // optional, use Terminal colors in GUI
```

   for an automatic upper/lowercase support and to use fix fonts to minimize porting effort, see fsman LNG.9 and LNG.5.2, LNG.5.3. If you wish to see PC8 lines and boxes also in GUI mode, add also

```
   SET GUITRANSL TEXT  ON
   SET GUITRANSL LINES ON
   SET GUITRANSL BOX   ON
```

   as described in section LNG.5.3 and LNG.5.4.2. See also example in *<FlagShip_dir>/examples/umlauts.prg* and *pc8lines.prg* where the *<FlagShip_dir>* is your installation path, e.g. /usr/local/FlagShip8

3. Internalization: in dependence of the used source-code editor, you may need to set

```
   SET SOURCE ASCII   // already set in fspreset.fh
or SET SOURCE ISO     // or: SET SOURCE ANSI
```

for an automatic translation of your national character set used in strings. You may test it with *<FlagShip_dir>/examples/umlauts.prg* or *western.prg* For database internalization, you may use SET ANSI ON/OFF or Oem2Ansi() and Ansi2oem() functions, in dependence on the used run-time mode and character set.

4. If your program already tests for index availability, skip to point 5. Otherwise add test for indices before opening them

```
if !file("myindex" + indexext())
   use mydatabase excl
   if !used()
      alert("could not open 'mydatabase.dbf' exclusively")
      quit
   endif
   index on ... to myindex
   use
endif
use mydatabase index myindex shared
```

   This method also creates FlagShip indices at the first invocation.

5. Compile your sources by FlagShip, e.g.

```
FlagShip myapp*.prg -Mmain -o myapplic
```

   see fsman section FSC for further details. Watch for compiler or linker errors. If any error occurs, read the self-explaining error message and check fsman section FSC.1.8, fix and recompile. For larger applications, you may preferably use 'make' (see FSC.2) which replaces DOS/Clipper compiling and linking with @... or rmake files.

6. Run your application by "`./myapplic`" in GUI mode, or by using the newfswin (or newfscons, newfsterm) script in Terminal i/o mode, e.g. "`newfswin ./myapplic -io=t`", see section FSC.3 and REL for details.

9.7 Porting to MS-Windows step-by-step

1. You may use your sources (*.prg, *.fmt) and data (*.dbf, *.dbt, *.lbl, *.frm) from DOS "as is" except the indices, which needs to be created anew, see LNG.9.3 and point 4 below.

2. At the begin of your main module, add the statements

```
// #include "fspreset.fh"   // optional
SET FONT "courier",10       // use fixed fonts
oApplic:Resize(25,80,,.T.)  // resize GUI window accord.to font
// SET GUICOLOR ON          // optional, use Terminal colors in GUI
```

   for an automatic upper/lowercase support (optional, only to keep multi- platform compatibility to Unix/Linux) and to use fix fonts to minimize porting effort, see fsman LNG.9 and LNG.5.3, LNG.5.4. If you wish to see PC8 lines and boxes also in GUI mode, add also

```
SET GUITRANSL TEXT  ON
SET GUITRANSL LINES ON
SET GUITRANSL BOX   ON
```

as described in section LNG.5.3 and LNG.5.4.2. See also example in <*FlagShip_dir*>*examples**umlauts.prg* and *pc8lines.prg*, where the <*FlagShip_dir*> is your installation path, e.g. C:\FlagShip8

3. Internalization: in dependence of the used source-code editor, you may need to set

```
   SET SOURCE ASCII   // already set in fspreset.fh
or SET SOURCE ISO     // or: SET SOURCE ANSI
```

for an automatic translation of your national character set used in strings. You may test it with <FlagShip_dir>\examples\umlauts.prg For database internalization, see SET ANSI ON/OFF or Oem2Ansi() and Ansi2oem() functions, used in dependence on the current run-time mode and character set, see details in section CMD and FUN.

4. If your program already tests for index availability, skip to point 5. Otherwise add test for indices before opening them

```
if !file("myindex" + indexext())
   use mydatabase excl
   if !used()
      alert("could not open 'mydatabase.dbf' exclusively")
      quit
   endif
   index on ... to myindex
   use
endif
use mydatabase index myindex shared
```

This method also creates FlagShip indices at the first invocation.

5. Compile your sources by FlagShip, e.g.

```
FlagShip mymain.prg myapp*.prg
```

see fsman section FSC for further details. Watch for compiler or linker errors. If any error occurs, read the self-explaining error message and check fsman section FSC.1.8, fix and recompile. For larger applications, you may preferably use make (named 'nmake' in MS-VC++) which replaces DOS/Clipper compiling and linking with @... or rmake files, see FSC.2 for details.

6. Run your application by "mymain" in GUI mode, or in Terminal or Basic i/o mode by using the -io=t or -io=b switch respectively. You may invoke the executable either from command-line (console window or start->run) or by click on the .EXE file in Explorer. To avoid CMD subwindow in GUI mode, compile with -io=g switch. See section FSC.3 and REL for further details.

10. Programming Examples

For a overview how to use FlagShip, we give you here short commented program examples. For these purposes, different programming techniques are demonstrated. Please consult also many other examples in the manual or the attached *.prg sources.

Example 1: create a new database and fill it with data. See also example 5 for a generalized program.

```
*** file: example1.prg

#ifdef FlagShip
# include "fspreset.fh"                       // w/o regard lower/uppercase
  SET FONT "courier",10                       // use fixed fonts
  oApplic:Resize(25,80,,.T.)                  // resize GUI window
#endif
#define CRLF chr(13)+chr(10)                  /* pseudo-constants */

LOCAL aStruct := {{"Name",    "C", 25, 0}, ; // declare multidimensional
                  {"First",   "C", 20, 0}, ; // array containing the
                  {"Address", "C", 50, 0}, ; // required database
                  {"Phone",   "N", 20, 0}, ; // structure
                  {"Note",    "M", 10, 0} }
LOCAL ii

if !file("address.dbf")                       // is the database available?
   dbcreate ("address", aStruct)              // no, create it now
endif
use Address                                   // open the database
if !used()                                    // successful ?
   ? "Database address.dbf is in " + ;
     "exclusive use by others"                // no, display message and
   quit                                       // terminate the application
endif
if LastRec() == 0                             // is it empty ?
   ? "adding dummy addresses "                // yes, fill first 10 records
   for ii := 1 to 10
      ?? "."                                  // display progress
      append blank                            // add new record
      replace Name  with "Any Name" + str(ii,3)
      replace First with "foo", Address with "dummy data" , ;
              Phone with ii * 10000 + ii
      if (ii % 2) == 0
         replace Note with "Note of record " + ltrim(str(recno())) + ;
            CRLF + "-- line 2" + CRLF + CRLF + "-- line 4"
      endif
   next
endif
quit                                          // program end
*** eof

Compile: FlagShip example1.prg -oexample1
Execute: example1   -or-   ./example1
```

Example 2: display some fields the database "address.dbf"

```
*** file: example2.prg

PROCEDURE Main (cmd1, cmd2)                // main entry point

? "Command-line parameters: "
if empty(cmd1)
   ?? "-none-"
else
   ?? cmd1, if (empty(cmd2), "", cmd2)
endif
if .not. file("address.dbf")
   ? "Sorry, database 'address.dbf' is not available."
   QUIT
endif
USE address                                // open database, exclusive
if .not. used()                            // success ?
   ? "Sorry, cannot open address.dbf - used by others ?"
   QUIT
endif
? padl("recno",7), "", padr("Name",len(name),"."), ;
  padr("First",len(First),"."), padc("Phone",fieldlen("Phone"),".")
LIST name, First, Phone                    // or: ... TO FILE /dev/lp0
USE                                        // close database
return
*** eof

Compile: FlagShip exam*2.prg -Mmymain -na
Execute: a.out  -or-  ./a.out  -or-  newfscons ./a.out
```

Example 3: browse thought the database "address.dbf".

```
*** file: example3.prg
#ifdef FlagShip
# include "fspreset.fh"                    // convert files to lowercase
#endif
if ! file("address.dbf")
   ? "Sorry, database 'address.dbf' is not available."
   quit
endif
USE address SHARED                         // open database, multiuser
if !used()                                 // success ?
   ? "Sorry, cannot open address.dbf - access rights ?"
   quit
endif
DbEdit (0,0, 15, MaxCol())                 // browse at row 0..15
use                                        // close database
return 22                                  // exit with return code
*** eof

Compile: FlagShip ex*3.prg -oexample3
Execute: example3   -or-   ./example3
```

Example 4: display or maintain any given database (optionally indexed by the giving index). See also BROWSE() and DBEDIT() and TBROWSEDB()

```
*** file: example4.prg
** procedure example4                          // created automatically

LOCAL executable, aColors
#ifdef FlagShip
# include "fspreset.fh"                        // w/o regard lower/uppercase
  executable := execname()
#else
  executable := "EXAMPLE4.EXE"
#endif

PARAMETERS DbfName, IndexName                  // retr. command-line params

if .not. CheckOpenIt (@DbfName, @IndexName, executable)
   ? "Start: " + executable + " dbf_name [index_name]"
   wait
   quit
endif
if iscolor()
   aColors := {"GB+/B,R+/BG", "GR+/B", "W+/N,N/W", ;
               "R+/B", "W+/N",  "G/N"  }
endif
@ 0,0 say "File : "
@ 0, col() say DbfName COLOR (if (iscolor(), "GR+/N", "W+/N"))

BROWSE (1, 0, maxrow()-1, maxcol(), aColors)         // does the work
return

*-------------------------------------------------------------------------
* CheckOpenIt() checks and corrects the parameters passed by reference,
*               and opens the database (with index, if given)
*
FUNCTION CheckOpenIt (DbfName, IndexName, execName)

if empty(DbfName) .or. left(dbfName,2) == "-h"
   ? "** The correct syntax is: " + execName + " dbfName [indexName]"
   return .F.
endif

if .not. (upper(right(dbfName,4)) == ".DBF")
   dbfName += ".dbf"                        // dbfName = dbfName + ".dbf"
endif
if .not. file(dbfName)
   ? "Sorry, file " + dbfName + " not available."
   quit
endif

Use (dbfName) SHARED
if !used()                                  // or: if .not. used()
   ? "Sorry, cannot open " + dbfName + " - access rights ?"
   return .F.
```

```
endif
if !empty(IndexName)
   IndexName += IndexName + if(indexext() $ IndexName, "", indexext())
   if file(IndexName)
      set index to (IndexName)
   else
      wait "Index " + IndexName + " not available, using " + ;
           dbfName + " w/o index. Any key..."
      indexName := ""
   endif
endif
return .T.

*** eof

Compile: FlagShip example4.prg -oshowdbf
Execute: showdbf   -or-   newfscons ./showdbf
```

Example 5: create a new database from a ASCII structure file, including an extensive validity check and command-line help. This demonstrates diverse array and file i/o handling. See similar, simplified Example 1. This file is available also in <FlagShip_dir>/system/creadb.prg.

```
*** File: creadb.prg
*-------------------------------------------------------------------------
* Main program
*
* FUNCTION Start (DbfName, AsciName)    // retr. command-line params

PARAMETERS DbfName, AsciName            // alternative syntax
LOCAL executable
LOCAL aDbStru := {}

#ifdef FlagShip                         // FlagShip settings
# include "fspreset.fh"                 // convert files to lowercase
# include "fileio.fh"                   // defines used here
  executable := execname()
#else                                   // Clipper 5.x settings
# include "fileio.ch"
  executable := "EXAMPLE5.EXE"
# define FS_START 0
#endif

if .not. CheckIt (@DbfName, AsciName, executable)
   quit
endif
aDbStru := readStruct(AsciName)          // fill array
if len(aDbStru) == 0
   ? "** sorry, no valid data in the file " + AsciName
   ? "   invoke '" + executable + " -h' for help"
   quit
endif

dbcreate (DbfName, aDbStru)             // create the database file
```

```
?
? "--> done, file " + lower(DbfName)
if ascan(aDbStru, {|x| x[2]=="M" }) > 0  // memo field(s) ?
   ?? " and " + lower(substr(DbfName,1,len(DbfName)-1)) + "t"
endif
?? " created."
? "--> Display structure (y/j/o|n) ? "
ii := 0
while !(upper(chr(ii)) $ "YJNON")
   ii := inkey(0)
enddo
if upper(chr(ii)) # "N"
   displStruct (DbfName)
endif
return 0

*------------------------------------------------------------------------
* CheckIt() checks and corrects the parameters passed by reference
*
FUNCTION CheckIt (DbfName, AsciName, execName)
LOCAL answer := ""

? "--> This '" + execName + "' creates a database according to " + ;
  "ASCII structure file"

if empty(DbfName) .or. empty(AsciName) .or. left(dbfName,2) == "-h"
   ? "--> The correct syntax is: " + execName + ;
     " <dbfName> <ascName>"
   ? "    where <dbfName> is the name of the dbf (optional with path)"
   ? "          <ascName> is the name of an ASCII file " + ;
       "describing the dbf structure"
   ? "    Structure of each line in the <ascName> file:"
   ? "    <FieldName> <FieldType> [<FieldLen>] [<FieldDeci>]"
   ? "      <FieldName> is the name of the field"
   ? "      <FieldType> is the field type (C,N,D,L,M)"
   ? "      <FieldLen>  is the total field length [or default]"
   ? "      <FieldDeci> is the number of decimal digits [or 0]"
   ? "    whereby the data are separated by space(s) or tab(s), e.g.:"
   ? "      |# comment lines or inline-comments are prefaced by "  + ;
         "the '#' sign"
   ? "      |                     # empty lines are ignored"
   ? "      |FamName C            # sets default field length to 10"
   ? "      |AddRess ch  50       # uppler/lower case is supported"
   ? "      |Salary  Num 10  3    # only 1st char of FieldType " + ;
       "is signific."
   ? "      |# eof"
   ?
   return .F.
else
   ? "--> enter '" + execName + " -h' for help"
   ? "-->"
   ? "--> Your entry is: " + execName, DbfName, AsciName
   ?
endif
if .not. file(AsciName)
   ? "*** Sorry, the ascii file " + asciName + " is not available."
```

```
   return .F.
endif
if .not. (upper(right(dbfName,4)) == ".DBF")
   dbfName += ".dbf"                        // make correction
endif
if file(dbfName)
   while !(upper(left(answer,1)) $ "NYJO")  // wait for correct key
      accept "Warning: database " + dbfName + " is AVAILABLE. " + ;
             "Overwrite (N|no/y|yes|j|o) ? " to answer
   enddo
   if upper(left(answer,1)) == "N"
      return .F.
   endif
endif
return .T.

*------------------------------------------------------------------------
* ReadStruct() reads the ascii file and stores data into array
*
STATIC FUNCTION ReadStruct (cFileName)
LOCAL  ii, handle, buff, nSize, line := 0, retArr := {}, arr

handle := FOPEN (cFileName, FO_READ)     // open input read-only
if handle < 0
   ? "*** i/o error on open " + cFileName + " (access rights ?)"
   return retArr
endif
nSize := FSEEK (handle, 0, FS_END)       // file size in bytes
FSEEK (handle, 0, FS_START)              // go to file begin
while FSEEK(handle, 0, FS_RELATIVE) < nSize
   buff := alltrim( FREADTXT (handle))   // read one asci line
   buff := strtran (buff, chr(9), " ")   // convert tabs to space
   ? str(++line,3) + ":" + buff
   if (ii := at("#", buff)) > 0
      buff := substr(buff, 1, ii-1)      // remove comments
   endif
   if empty(buff)                        // empty lines are ignored
      loop                               // read next line
   endif
   arr := tokenize (@buff, retArr)
   aadd (retArr, arr)
enddo
FCLOSE (handle)                          // close input
return retArr

*------------------------------------------------------------------------
* Tokenize() split input into tokens
*
STATIC FUNCTION Tokenize (buff, retArr)
LOCAL  ii, cc, nType := 0, arr := {}
STATIC aCheck := {{"C",0}, {"N",0}, {"M",10}, {"D",8}, {"L",1}}

while "  " $ buff
   buff := strtran (buff, "  ", " ")     // remove multiple spaces
enddo
buff := upper(buff) + " "                // convert to uppercase
```

```
while (ii := at(" ", buff)) > 1          // tokenize the input
   aadd (arr, left(buff, ii-1))          // into array elements
   buff := substr(buff, ii+1)
   if len(arr) == 4
      exit
   endif
enddo

do case
case len(arr) < 2                        // minimal requirements ?
   ? "*** error: at least <FieldName> and <FieldType> must be given"
   quit
case !isalpha(arr[1])                    // 1st char = Alpha
   ? "*** error: <FieldName> must start with Alpha char, here: " +;
     arr[1]
   quit
endcase
arr[1] := trim(left(arr[1], 10))         // name = max 10 chars

for ii := 1 to len(arr[1])               // check name validity
   cc := substr(arr[1],ii,1)
   if .not. (isAlpha(cc) .or. isDigit(cc) .or. cc == "_")
      ? "*** error: invalid character '" + cc + "' in " + arr[1]
      quit
   endif
next
if len(retArr) > 0
   if ascan (retArr, {|x| x[1] == arr[1]}) > 0  // check field name
      ? "*** error: field name " + arr[1] + " is multiple defined"
      quit
   endif
endif
for ii := 1 to len(aCheck)
   if aCheck[ii,1] == left(arr[2],1)     // check the field type
      nType := ii
      exit
   endif
next
if nType == 0
   ? "*** error: wrong <FieldType>, here: " + arr[2]
   quit
endif
arr[2] := left(arr[2],1)

if len(arr) < 3                          // check the field size
   aadd (arr, if(aCheck[nType,2] == 0, 10, aCheck[nType,2]))
   ? "--- FieldLength of " + arr[1] + " set to " + ltrim(str(arr[3]))
else
   arr[3] := val(arr[3])
   if arr[3] <= 0 .or.  ;
      (arr[3] # aCheck[nType,2] .and. aCheck[nType,2] # 0)
      ? "--- FieldLength of " + arr[1] + " CORRECTED " + ;
        "from " + ltrim(str(arr[3]))
      arr[3] := if(aCheck[nType,2]==0, 10, aCheck[nType,2])
      ?? " to " + ltrim(str(arr[3])) + " ... any key"
      inkey(0)
```

```
   endif
endif
if len(arr) < 4                          // check deci places
   aadd (arr, 0)
else
   arr[4] := val(arr[4])
   if arr[2]=="N"                        // but for numeric only
      if arr[4] < 0
         arr[4] := 0
      else
         if (arr[4] +2) > arr[3]
            ? "--- DeciPlaces of " + arr[1] + " CORRECTED " + ;
              "from " + ltrim(str(arr[4]))
            arr[4] := arr[3] -2
            ?? " to " + ltrim(str(arr[4])) + " ... any key"
            inkey(0)
         endif
      endif
   else
      arr[4] := 0
   endif
endif
return arr

*------------------------------------------------------------------------
* DisplStruct() display the database structure
*
STATIC FUNCTION displstruct(name)
local astru, ii, jj, m1 := 0, m2 := 0, m3 := 0, split := .F., dummy

use (name)
astru := dbstruct()
use
dummy := "    ------------- structure of " + name + " "
dummy += replicate("-", 60 - len(dummy) -1)
? dummy
ii := len(astru)
if ii > 5
   ii := round((ii / 2) + 0.1, 0)
   split := .T.
endif
for jj := 1 to len(astru)
   m1 := max(m1, len(astru[jj,1]))
   m2 := max(m2, len(ltrim(str(astru[jj,3]))))
   m3 := max(m3, len(ltrim(str(astru[jj,4]))))
next
for jj := 1 to ii
   ? space(4) + padr(astru[jj,1], m1), astru[jj,2], ;
     str(astru[jj,3],m2), str(astru[jj,4],m3)
   if split .and. ((jj + ii) <= len(astru))
      ?? space(10), padr(astru[jj+ii,1], m1), astru[jj+ii,2], ;
         str(astru[jj+ii,3],m2), str(astru[jj+ii,4],m3)
   endif
next
? "    " + replicate("-", 60 -5)
return NIL
```

```
*------------------------------------------------------------------------
* For DOS only: simulate FlagShip functions not available in Clipper
*
#ifndef FlagShip
  static FUNCTION FREADTXT(handle)
  local buff := space(500), out := ""
  local pos, ii, jj, cc
  pos := FSEEK (handle, 0, FS_RELATIVE)
  jj := FREAD (handle, @buff, 500)
  for ii := 1 to jj
     cc := substr(buff,ii,1)
     if asc(cc) != 10 .and. asc(cc) != 13
        out += cc
        pos++
     else
        while ii <= jj .and. (cc == chr(10) .or. cc == chr(13))
           pos++
           cc := substr(buff,++ii,1)
        enddo
        exit
     endif
  next
  FSEEK (handle, pos, FS_START)
  return out
#endif
*** eof

Compile: FlagShip creadb.prg -ocreadb
Execute: creadb   -or-   newfscons ./creadb
```

Index

D

G

H

I

K

L

M

N

O

P

R

S

T

U

v

w

x

Notes

multisoft Datentechnik
Schönaustr. 7
D-84036 Landshut

http://www.fship.com
sales@multisoft.de
support@flagship.de