FlagShip

Object Oriented Database Development System

Cross-Compatible to Unix, Linux and MS-Windows

MULTISOFT

Release 8.1 **Section FSC**

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

multisoft Datentechnik, Germany

FlagShip

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

Section FSC

Manual release: 8.1

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

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

Copyright

Trademarks

Headquarter Address

multisoft Datentechnik
Schönaustr. 7
84036 Landshut
Germany

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

Phone: (+49) 0871-3300237

Web: http://www.fship.com

FSC: The FlagShip Compiler

1. The FlagShip Compiler

The **FlagShip** DBMS package consists of three main parts (refer also to LNG.1.1):

- The FlagShip Compiler, creates 32-bit or 64-bit objects and executables,
- The accompanying FlagShip Library,
- Additional tools and system files.

This section covers the operation and the handling of the Compiler and the executables produced. The additional FlagShip tools and utilities are also explained.

For detailed information of the FlagShip language, refer to section LNG. The preprocessor directives are summarized in the PRE section. All the standard commands, directives, functions and the Extend C system included in the FlagShip library are described in sections CMD, FUN and EXT.

1.1 Compiler Tasks

The FlagShip compiler consists of three parts (tasks) and performs several steps:

1. The compiler main control module, named "**FlagShip**" is usually installed in the <FlagShip_dir>/bin directory and in Unix/Linux with a symbolic link to /usr/bin/FlagShip and is therefore available in the standard PATH. Only this compiler module is directly invoked by the programmer.

   In MS-Windows, the main module is named **FlagShip.exe** and is installed below the installation directory, e.g. in C:\FlagShip8\bin or C:\Program Files\FlagShip8\bin sub-directory. Since the Setup does not modify your environment, there is a script associated to an icon on your desktop named "FlagShip console", which sets all required environments to reach the compiler and linker.

   Normally, this module activates the other FlagShip compiler tasks required, in addition to the final C compilation and linking of the executable. Depending on the switches and file names given, some of the following compiler steps may be skipped or executed separately.

   This main module reads the default settings stored in the **FS8config*** ASCII file (in the local, $FSCONFIG or /etc directory, see more in chapter FSC.1.4.2 and REL) and passes the selected information to steps 2 to 5.

2. The FlagShip Preprocessor named "**FlagShip_p**" is installed in the same directory as the main module, or in the directory specified by the FSPATH setting in the <FlagShip_dir>/etc/ FS8config* file. This task is called by the main FlagShip module for all ***.prg** and ***.fmt** sources.

It performs syntactical source checking and translates the preprocessor directives (see section PRE) and user defined commands to UDFs (user- defined-functions). The output is stored in files named ***.bp**, which are similar (but not identical) to Clipper's *.ppo files.

The preprocessor uses the standard include file **std.fh** by default (usually stored in .../usr/include), as well as other optional include files specified in the source code or by the -nI= and -i= switches.

If the switch -q is not used, a "*src-line Pass 1: file-name*" message displays information on the currently compiled source. If the switch -a is used, steps 3 to 5 are skipped.

3. The FlagShip Compiler named "**FlagShip_c**" (or FlagShip_c.exe resp.) is also installed in the same directory as the FlagShip_p module.

   This task is called by the main FlagShip module for all *.prg, *.fmt or *.bp sources as the second step in translating the preprocessed FlagShip (and also Clipper or xBASE) source code into C source. There are additional checks for syntax, semantics and plausibility.

   The input files are the *.bp from pass 1, the output ***.c**. If the switch -q is not used, a "*src-line Pass 2: file-name*" displays information on the currently compiled source. If the switch -b is used, steps 4 to 5 are skipped.

4. If the compilation in steps 2 and 3 is successful, or when giving *.c and/or *.o files only as input, the FlagShip main module includes the native C compiler (cc. bcc, ms-vc) to complete the translation into the machine language (native code), passing down all the required and user given parameters.

   In the absence of the -c switch, FlagShip also creates and compiles the start-up module **<name>_m.c**, where the <name> is the same as the first file specified on the command line, or the name given by -Mname.

   The produced output is ***.o** (or ***.obj** in Windows), i.e. native object file. The messages displayed depend on the C compiler used. If the switch -c is given, step 5 is skipped.

5. If no error is encountered in step 4, the FlagShip main module involves the Unix or Windows native linker (ld or LINK), passing down all the required and user given parameters.

   In this step the *.o and *.a, *.so (*.obj and *.lib, *.dll in Windows) object files and libraries are input. The dynamic or static FlagShip library, same as the required system libraries are passed automatically. All other user libraries must be specified with the library names at the invocation of FlagShip.

   Note: The FlagShip library is named libFlagShip_8rr_Xnn.a or libFlagShip_8rr_Xnn.so in Unix/Linux and FlagShip_8rr_Xnn.lib in Windows, whereby 8rr is the sub-release number and Xnn is either X32 or X64. They are stored in <FlagShip_dir>/lib (with a link to /usr/lib/* in Unix/Linux).

   The output produced is a self-contained executable, called a.out in Unix/Linux by default (or named <file>.exe in Windows where <file> is the first file in the compilation parameter). You may specify any other name by the -o switch.

When the compilation is finished (it takes usually only few seconds, the most time-consuming steps are 4 and 5), the executable produced can be invoked by giving its name in the command line, after the required and/or optional environment variables are checked or set.

All the above steps may be issued semi-automatically compiling all, the changed, the "newer" or any files by only one FlagShip invocation. Optionally, you may also use the Unix "make" or Windows "make" or "nmake" utility. Refer to chapter FSC.2 for further details.

1.2 Invoking the FlagShip Compiler

The FlagShip compiler, an executable named "**FlagShip**" (refer to chapter 1.1), is invoked in the Unix console command-line (or from the "FlagShip Console" CMD window in MS-Windows), alternatively via script or .bat file by the general syntax

FlagShip [<sources>] [@<cmdfile>] [<objects>] [<libs>] [<options>]

Sources are plain ASCII text files containing a collection of computer instructions in FlagShip or C computer language. The source code is transformed by the FlagShip compiler into low-level machine code. FlagShip will accept source files with extension ".prg" (or ".fmt") for the FlagShip language (Clipper, Foxbase, FoxPro, dBase are accepted as well), and ".c" for sources written in standard C language. Source files created by FlagShip compiler from .prg sources (.c, .bp, .bi) are accepted and passed automatically. If the source is in other than current directory, prefix the source with a path (absolute or relative). If the path or file name contains space(s), enclose the entry in quotas "...".

@cmdFile is optional "command-line-file", a simple text file containing names of source files and/or compiler switches. See details below.

Objects are low-level machine code files with the extension ".obj" or ".o", containing the translation of source code. The machine code depends on the used C compiler and architecture (32/64bit). Objects created from FlagShip invocation for .prg and .c source files are passed automatically.

Libs are libraries, i.e. collection of object files, with the extension ".lib" for MS-Windows and ".a" in Linux. The FlagShip standard library named FlagShip_8*.lib in MS-Windows or libFlagShip_8*.{a,so} in Unix/Linux contains all standard commands and functions, and is included automatically according to the used 32/64bit architecture. You may use additional libs created by FlagShip or by C, as well as third party libs. These libraries need to meet the same used architecture, libs for different architecture are incompatible to each other (this also apply for Clipper 16bit libraries which cannot be used).

Options are modifiers controlling the compilation process, described in details bellow, see chapter 1.4 below.

Wildcards (files with * and ? marks) are accepted for source files and objects, also for files in the command-line-file. These simple wildcards are accepted for files but not for directories, so `"../mysource/xy*.prg"` is ok, but `"../mys*/xy*.prg"` is not. Regex wildcards in Linux with {..}, [..] etc are resolved when entered in command-line with FlagShip invocation.

Invocation To get brief **on-line help**, invoke the compiler without options or parameters, or with the "-h", "/h" or "--help" by entering

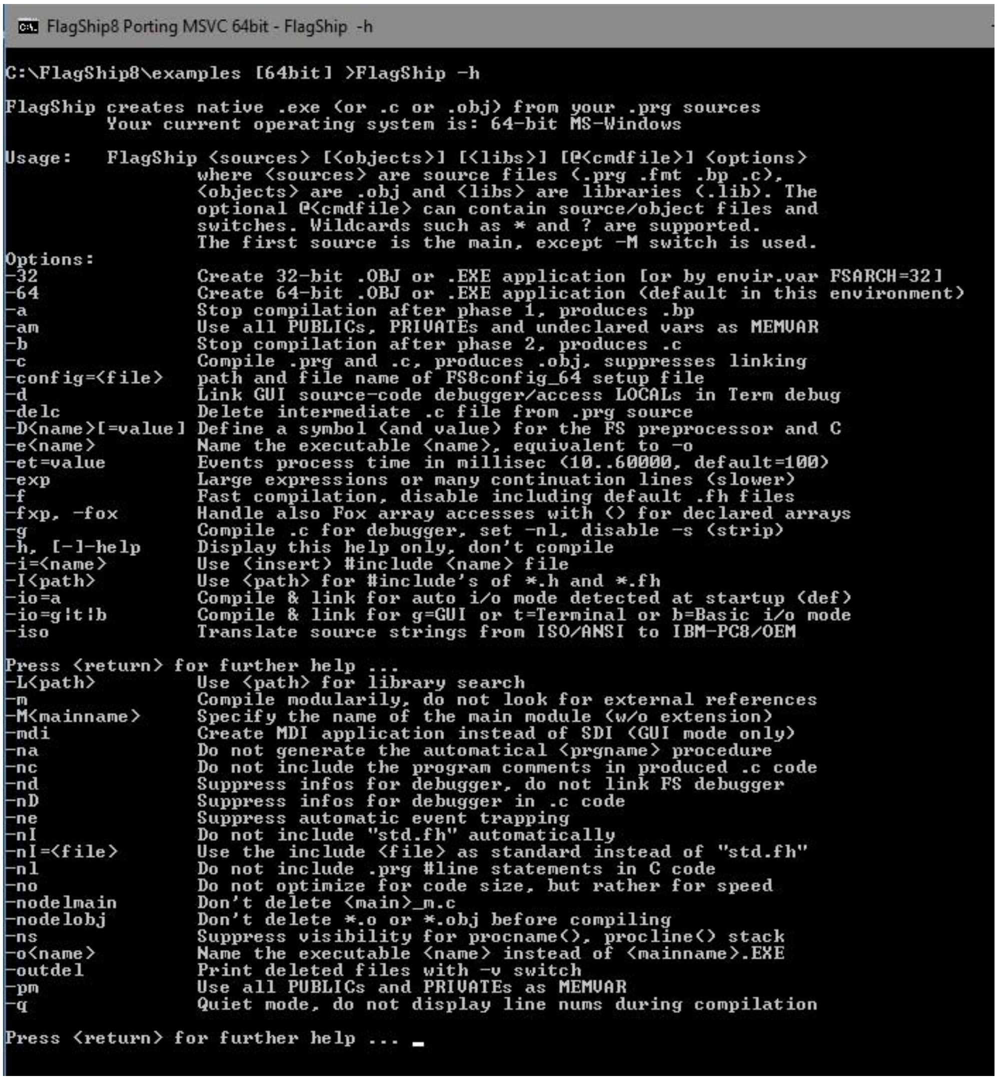
```
$ FlagShip
$ FlagShip -h
$ FlagShip /h
$ FlagShip --help
```

Do not enter the $ sign, since it here denotes the shell prompt only. Note also the upper/lower case significance of Unix/Linux file names.

In MS-Windows, the invocation is similar (but case insensitive), e.g.

```
C:> FlagShip
C:> flagship /h
D:> FlagShip --help
```

end so on.



```
C:\FlagShip8\examples [64bit] >FlagShip -h

FlagShip creates native .exe (or .c or .obj) from your .prg sources
         Your current operating system is: 64-bit MS-Windows

Usage:   FlagShip <sources> [<objects>] [<libs>] [@<cmdfile>] <options>
                 where <sources> are source files (.prg .fmt .bp .c),
                 <objects> are .obj and <libs> are libraries (.lib). The
                 optional @<cmdfile> can contain source/object files and
                 switches. Wildcards such as * and ? are supported.
                 The first source is the main, except -M switch is used.
Options:
-32              Create 32-bit .OBJ or .EXE application [or by envir.var FSARCH=32]
-64              Create 64-bit .OBJ or .EXE application (default in this environment)
-a               Stop compilation after phase 1, produces .bp
-am              Use all PUBLICs, PRIVATEs and undeclared vars as MEMVAR
-b               Stop compilation after phase 2, produces .c
-c               Compile .prg and .c, produces .obj, suppresses linking
-config=<file>   path and file name of FS8config_64 setup file
-d               Link GUI source-code debugger/access LOCALs in Term debug
-delc            Delete intermediate .c file from .prg source
-D<name>[=value] Define a symbol (and value) for the FS preprocessor and C
-e<name>         Name the executable <name>, equivalent to -o
-et=value        Events process time in millisec (10..60000, default=100)
-exp             Large expressions or many continuation lines (slower)
-f               Fast compilation, disable including default .fh files
-fxp, -fox       Handle also Fox array accesses with () for declared arrays
-g               Compile .c for debugger, set -nl, disable -s (strip)
-h, [-]-help     Display this help only, don't compile
-i=<name>        Use (insert) #include <name> file
-I<path>         Use <path> for #include's of *.h and *.fh
-io=a            Compile & link for auto i/o mode detected at startup (def)
-io=g|t|b        Compile & link for g=GUI or t=Terminal or b=Basic i/o mode
-iso             Translate source strings from ISO/ANSI to IBM-PC8/OEM

Press <return> for further help ...
-L<path>         Use <path> for library search
-m               Compile modularily, do not look for external references
-M<mainname>     Specify the name of the main module (w/o extension)
-mdi             Create MDI application instead of SDI (GUI mode only)
-na              Do not generate the automatical <prgname> procedure
-nc              Do not include the program comments in produced .c code
-nd              Suppress infos for debugger, do not link FS debugger
-nD              Suppress infos for debugger in .c code
-ne              Suppress automatic event trapping
-nI              Do not include "std.fh" automatically
-nI=<file>       Use the include <file> as standard instead of "std.fh"
-nl              Do not include .prg #line statements in C code
-no              Do not optimize for code size, but rather for speed
-nodelmain       Don't delete <main>_m.c
-nodelobj        Don't delete *.o or *.obj before compiling
-ns              Suppress visibility for procname(), procline() stack
-o<name>         Name the executable <name> instead of <mainname>.EXE
-outdel          Print deleted files with -v switch
-pm              Use all PUBLICs and PRIVATEs as MEMVAR
-q               Quiet mode, do not display line nums during compilation

Press <return> for further help ... _
```

If the executable "FlagShip" was not found, you should check the environment variable PATH (using e.g. "env" or "printenv" or "echo $PATH" in Unix or “echo %PATH%” in Windows). The directory where the FlagShip compiler modules are stored (usually /usr/bin or <FlagShip_dir>/bin) must be included there. The **<FlagShip_dir>** is the installation directory, usually C:\FlagShip8 in MS-Windows or /usr/local/FlagShip8 in Linux, see the previous chapter.

With FlagShip for MS-Windows, you need to use the "FlagShip-Console" window (an icon on your desktop) for the compiling stage, which sets the required environment and paths automatically to reach the "FlagShip" executable.

The simplest way to create a Unix executable is by invoking the compiler using only the main module name, for example:

```
   $ FlagShip address.prg
or $ FlagShip address.prg -o address
```

If the application consists of several source files, all the files referred to (e.g. by SET PROCEDURE TO <fileName>.prg), will also be compiled automatically, see below. The created Linux executable a.out (or the one given by -o switch) is then executed by simply entering

```
   $ ./a.out
or $ ./address
```

In MS-Windows the linker creates the executable named same as main source file (here ADDRESS.EXE), which is invoked by its name (or the name given by -o switch), i.e.

```
   C:> FlagShip address.prg
or C:> FlagShip mysource.prg -o address.exe
   C:> address.exe
```

Compiling by FlagShip this way is very similar to using the DOS compiler and linker, e.g.:

```
   C:> CLIPPER address
   C:> RTLINK|PLINK86|BLINKER file address
   C:> address.exe
```

but FlagShip do the compile and link steps automatically, except you use the -c switch (compile only).

If your source consist of more than a single .prg file, you need to compile (or link) all used sources, since the executable must contain all in your application used procedures and functions. Files referred by SET PROCEDURE TO <fileName>.prg are processed automatically (except you use the -m switch). All used standard FlagShip functions are taken automatically from the FlagShip library. You may simply add all required source files in the command- line, e.g.

```
    $ FlagShip mymain.prg part1.prg part2.prg other.prg -o myapplic
or  $ FlagShip mymain.prg part[1-2].prg ot*.prg -o myapplic
or  $ FlagShip *.prg -Mmymain -o myapplic

    $ ./myapplic
```

or in Windows

```
    C:> FlagShip mymain.prg part1.prg part2.prg other.prg -o myapplic.exe
or  C:> FlagShip mymain.prg part*.prg o*.prg -o myapplic.exe
or  C:> FlagShip *.prg -Mmymain -o myapplic.exe

    C:> myapplic.exe
```

which compiles all above sources, links them together with FlagShip library and creates an executable named "myapplic" or "myapplic.exe" resp. The first source file should be the main program, or use the "`-Mprocname`" switch to specify the start procedure (see "Compiler Switches" below).

For large projects, you may compile .prg sources to objects and subsequently link them together. The advantage is, that only changed sources needs to be recompiled, not the whole project, e.g.

```
   $ FlagShip -c *.prg -Mmymain
   $ FlagShip mymain.prg *.o -Mmymain -o myapplic
or $ FlagShip changed.prg *.o -Mmymain -o myapplic
```

or in Windows

```
   C:> FlagShip -c *.prg -Mmymain
   C:> FlagShip mymain.prg *.obj -Mmymain -o myapplic.exe
or C:> FlagShip changed.prg *.obj -Mmymain -o myapplic.exe
```

You also may use "**command-line-file**" which is a simple text file containing names of source files and/or compiler switches, separated by space or new line. Empty lines are ignored, in-line comments starts with hash (#) or two slashes (//). Pass this file to FlagShip by prefixing the name with at-sign "@". The combination of compiler options (if any) and entries in the command-line-file is the resulting invocation of FlagShip compiler.

For example, the text file named "myApp.cmd" contains:

```
   # my command-line-file for myapplic[.exe]
   other*.prg  "../my src/some*.prg"   // used source files
   util*.c -Mmymain -na                # other files and switches
   -delc
```

Invoking

```
   $ FlagShip myapp.prg @myApp.cmd -d -o myapplic
```

or

```
   C:> FlagShip myapp.prg @myApp.cmd -d -o myapplic.exe
```

is then equivalent to

```
   FlagShip myapp.prg other*.prg "../my src/some*.prg" util*.c \
            -Mmymain -na -delc -d -o myapplic[.exe]
```

or with resolved wildcars

```
   FlagShip myapp.prg other1.prg other2.prg "../my src/some_a.prg"    \
            "../my src/some_b.prg" "../my src/some other.prg" util.c  \
            utilx.c utily.c -Mmymain -na -delc -d -o myapplic[.exe]
```

Even more comfortable is to use the "make" or "nmake" utility (see FSC.2), which recompiles changed sources automatically. FlagShip provides tool named "**fsmake**" (see section FSC.6.9) which creates the by make used template "Makefile" nearly automatically.

1.3 Compiler Options and Switches

By invoking the FlagShip compiler, a list of options may be specified to control its behavior. Each option is introduced by the "-" (minus, dash) mark, immediately followed by the option character(s) and optionally an additional argument.

The general syntax is

FlagShip [<sources>] [@<cmdfile>] [<objects>] [<libs>] <options>

The **options** fall in three basic categories:

- FlagShip preprocessor switches
- FlagShip compiler switches
- C compiler and linker (cc and ld) switches

Options may be specified in any order and must be separated by at least one space or TAB character. All the options are case sensitive. Some of the options take an additional argument, which has to be given immediately (without spaces) behind the option.

The general syntax used here for the argument is <argument>. Replace the argument by the required entry, but do not enter the < > metacharacters themselves.

-32	Create 32-bit objects/executable. Default for BCC32 and 32bit OS by using <FlagShip_dir>/etc/FS8config_32 configuration. May also be set by environment variable FSARCH=32 Note for Linux: when the current environment is 64bit, you will need also 32bit libraries to be able to link
-64	Create 64-bit objects/executable. Default on 64bit operating systems by using <FlagShip_dir>/etc/FS8config_64 configuration. May also be set by environment variable FSARCH=64 Note for Linux: when the current environment is 32bit, you will need also 64bit libraries to be able to link
-a	Performs the preprocessor phase for *.prg files only. Skips the FlagShip C compiler and link phases. File(s) with **.bp** extension will be produced as a result. These are similar, but not identical to the .ppo files of Clipper.
-am	The FlagShip compiler assumes all ambiguous (undeclared and unaliased) variables are references to PRIVATE, PUBLIC or auto-PRIVATE memory variables. This has the same effect as using the MEMVAR-> or M-> aliases for any ambiguous reference. By omitting this option, all ambiguous references are assumed to be FIELD variables, if such exist. Otherwise they are assumed to be dynamically scoped variables.

-b	Performs the preprocessor and the FlagShip compiler phase for all specified *.prg or *.bp files only. Skips the C compiler and linker phase. File(s) with **.c** extension will be produced as a result. If the option -c is omitted, the main C module `<name>_m.c` is also produced.
-c	Performs the preprocessor and the FlagShip compiler phase for all specified *.prg or *.bp files and additionally invokes the Unix or Windows C compiler (cc) to compile the *.c files. Does not produce the main .c module `<name>_m.c`. Skips the linker phase. Object files with **.o** (or **.obj** in Windows) extension will be produced as a result.
-C	Causes the application to create a "core dump" file when it receives an "abort" signal or if an internal error occurs.
-config=<file>	Use configuration <file> instead of the default FS8config*
-d	Creates additional information for the FlagShip debugger to access LOCAL and STATIC variables. The -nl, -nD and -nd switches are ignored. When specified during the link phase, the executable stops in the debugger at the first executable statement of the main module. In Terminal i/o mode, you may then continue by entering 'Q' or set breakpoints etc, see FSC.5.2. In GUI mode, you must specify this switch to be able to use the **source code debugger**, see also FSC.5.1 for details. This switch disables -ne and -et=value switches for GUI, if set.
-D<name>	Defines an empty identifier for the FlagShip preprocessor or the C compiler, e.g. for conditional compiling. The <name> passed is case-sensitive, but is usually used in uppercase. Note that the empty identifier FlagShip is defined automatically by the FlagShip preprocessor. To define an identifier for C only, use -Wc,-D<name>.
-D<name>=<value>	Defines and/or initializes an identifier for the FlagShip preprocessor or the C compiler, e.g. for conditional compiling. The <name> and <value> passed are case-sensitive, usually in uppercase.
-delc	Deletes intermediate .c files created from .prg after the object file or the executable was successfully created. Does not apply when the -b switch is used.
-dyn	}
-dynamic	} forces dynamic linking, see <FlagShip_dir>/etc/FS8config*
-et=value	Set event process time in millisec (10..60000, default=100) Ignored when -d switch is given, or Set(_SET_EVENT_DURATION,n) or SetEvent(,<n>) is set.
-exp	Use large buffers for source with many continuation lines or large expressions. This switch slow-down the compilation significantly, so use it only if required, e.g. on stack overflow, when the compiler pass 1 exits w/o any error message but reports ** Warning: C compiler not

	invoked (1). Note: in Linux, you alternatively may increase the stack size by e.g. "ulimit -s 20000"
-f	Fast compilation, do not include default .fh files except std.fh, set.fh and inkey.fh. Equivalent to passing -DNOSTDCLASS and -DNOSTD-FUNCT, see *<FlagShip_dir>/include/std.fh* for consequences: the .bp and .c are significantly smaller and hence the compilation is faster, but it disables prototyping of standard functions and including of getclass.fh, tbrclass.fh and errclass.fh with early binding and checking of these objects, so you need to explicitly #include them when required. With -f switch, the compiler do not check passed parameters to standard functions which may result in runtime errors. So use this option with care.
-fxp	Handles FoxPro array compatibility and "." alias syntax
-fox	Same as -fxp, sets additionally -DFOXPRO -DFOXARRAYS for std.fh and uses additionally stdfoxpro.fh preprocessor file.
-g	Compile .c for C debugger, sets -nl, disables -s (no strip) In Linux, the "-g" switch is passed to C execution, in MS-Windows the "-Od -Z7" (or "-Od -v" with BCC32) switches are used instead (modifiable in the FS8config* file by CCDEBUG).
-h /h /?	}
-help	} displays short help with all available switches
--help	}
-i=<name>	Uses an include file <name>. Up to 10 `-i=<name>` options may be specified. Equivalent to adding the statement #include "<name>" at the end of the std.fh (or at the end of the file specified by the -nI=<file> switch), or at the beginning of each compiled .prg source.
-I<path>	Searches for the #include files *.fh (FlagShip) and *.h (C) in the specified <path> directory if they are not located in the current directory, but before searching the default /usr/include path. If more than one search directory is required, use multiple `-I<path>` options. To define a search path of *.h files for C only, use `-Wc,-I<path>`. Note: MS-Windows compiler accepts both "/" and "\" for path delimiters, Unix/Linux only "/".
-io=a|b|g|t	Compile for [a] *all* modes=hybrid, or for [b] *basic*, [t] *terminal*, [g] *GUI* mode only. Default is -io=a, creates hybrid application, determines the i/o mode at run-time, re-definable by command-line switch of the executable. You need to use the same switch for all application modules, also in the link-only phase if such is appropriate. The -io=g switch also avoids display the subsequent console window in GUI mode when running via link on desktop or from explorer, see details in FSC.1.3.2 and FSC.3.1
-iso	Translates strings in .prg from ISO → PC8 in preprocessor phase. Useful when extended character set = chr(128..255) is required in

strings (such as umlauts, accents, semi-graphics, etc.) and your source code editor supports only ISO/Windows character set. Note: see the charset difference in *<FlagShip_dir>/manual/charset.pdf* or in appendix (APP) of the printed or pdf manual, and read also section LNG.5.4 for further details about national characters.

-l<name> (Unix/Linux only) Uses the library named lib<name>.a for the link phase to satisfy all unresolved externals. Equivalent to `-Wc,-l<name>`. Externals (e.g. the UDF or UDP names) included in the specified *.o files will be preferred by the linker. The FlagShip library (libFlagShip_8*.a or libFlagShip_8*.so) will be used automatically. If more than one library is required, use multiple `-l`<name> options.

In MS-Windows, simply specify the library name (e.g. myownlib.lib) same as giving names of object files.

-L<path> (Unix/Linux only) Searches for default or specified libraries in the specified <path> directory if they are not located in the current directory, and before searching the default /usr/lib path. If more than one search directory is required, use multiple -L<path> options. Equivalent to -Wc,-L<path>.

-libpath:<path> (Windows only) Searches for default or specified libraries in the specified <path> directory if they are not located in the current directory, and before searching the default Windows include path. If more than one search directory is required, use multiple -`libpath:<path>` options.

-m Performs modular compilation of the specified *.prg files only. Suppresses the automatic search and compilation of .prg files called from the current source module by the DO statement or the SET FORMAT, SET PROCEDURE commands. However, header files specified with the #include directive or the -i= switch are compiled.

-mdi Compiles GUI based application using MDI (multiple document interface) screen layout, instead of the default SDI (singe document interface). Additional windows can be opened by MdiOpen() or the corresponding Wopen() of FS2 Toolbox. You don't need to use this switch if you use Wopen() from FS2 Toolbox. Apply for application running in GUI i/o mode, ignored otherwise with developers warning at run-time.

-M<name> Specifies the main module name. Normally, program execution will start at the beginning of the .prg file first given in the command line. Should the execution be started anywhere else, specify the main UDP procedure or UDF function with the <name> argument here. Similar to the UDF or UDP invocation, the <name> specifies the main name of the file or the procedure name only, without extension. If this switch is not specified, the start procedure/function is determined at run-time in that order:

a) if FUNCT/PROC main() is available, start execution in main()
b) if FUNCT/PROC start() is available, start execution in start()
c) otherwise use the implicit name of the first source in the compiler/linker list.

If -M<name> was given, the execution always starts in the <name>() procedure or function. The current start function is displayed with the -FSversion command line switch of the application ("*a.out -FSversion*" in Linux or "*myapplic.exe -FSversion*" in Windows).

-na Suppresses the automatic generation of a procedure with the same name as the .prg file. This option **must** be used if filewide variable declarations are used in the .prg file, or when the the file name contains embedded spaces within the first 10 chars. Refer to sections LNG.2.3.1, LNG.2.6.3 and CMD (command PROCEDURE and STATIC) for more information.

-nc Suppresses the transfer of full-line comments into the produced C code. Equivalent to the #nocomments preprocessor directive. See section PRE.

-nd Suppresses debugging information. The produced object cannot be debugged any more. If this switch is active during the link process, the debugger is not linked at all; pressing the ^O key or invoking ALTD() is then ignored. Note: using this option during the link process may result in a notable reduction of the executable size, since the unused standard functions (but available for the debugger) are not linked.

-nD Suppresses the debugging information and the generation of event trapping in the produced C code. Automatically includes the `-nl` and `-nd` (compiling) switch. Used mostly for tested-out and released modules, to increase the execution speed and decrease the object size by approx. 5 to 10%. For a released application, specify also the -nd switch during the link phase to avoid linking the debugger.

-ne Suppress automatic event trapping. Warning: use this option only if you exactly know what are you doing. Otherwise the GUI application may hang.

-nI=<file> Uses the <file> for the default preprocessor translation, instead of the "std.fh" file. If the <file> is stored in any other than the current directory, add the relevant path.

-nL Suppresses the generation of C line number information in the .c code produced. Used for C debugging of #Cinline or the produced C code and to localize a C error message within the produced .c files.

`-nl` Suppresses the generation of the corresponding .prg line number information in the produced .c code. By specifying this option, the resulting objects are a few percent smaller, but the exact localization of run-time errors will be not possible, nor can the source line numbers

	be displayed when the debugger is invoked or if there is a run-time error.
-no	Does not optimize for code size, but rather for speed. Similar to the SCO -Oxp option of cc (mcc). Ignored by the most C compilers nowadays.
-nodelobj	Don't delete object files before starting FlagShip compiler. If not given, FlagShip deletes .o or .obj file of the same name as .prg before processing to avoid confusions and incorrect link with old objects when the compilation fails due of syntax errors.
-ns	Suppress visibility for procname(), procline() stack
-o<name>	Specifies the <name> of the executable produced and passes this option to the linker. Equivalent to the -Wc,-o<name> entry. If the option is not specified, a.out is created by default.
-outdel	Display deleting of intermediate files. Apply together with -v. If not specified, intermediate files are deleted silently. See also the -delc and -v switch.
-pm	Automatic declaration of PUBLIC, PRIVATE and PARAMETERS variables as MEMVAR. If no other aliasing is used there, the FlagShip compiler will preface all variables declared by PUBLIC, PRIVATE and PARAMETERS using the pseudo-alias MEMVAR->
-q	Compiles in the quiet mode, suppresses line number display. This option is highly recommended if the output is redirected to a file or if compiling in the background. When using slow terminal connections, it will also speed up compilation.
-r=<name>	Specifies the <name> (optionally including path) of the repository file (see 1.4.3). Used in conjunction with the options -rc and/or -ru .
-rc	Adds prototypes of CLASS declarations and METHOD declarators found in the currently compiled file(s) into the repository, see also chapter 1.4.2.
-ro	The produced repository file (according to the -rc and -ru switch) will overwrite the old one. If the option is not specified, the prototypes of all files being currently compiled are appended to the existing reposit.fh (or the -r=<name>) file.
-ru	Adds prototypes of typed UDFs (PROCEDURE or FUNCTION) found in the currently compiled file(s) into the repository file, see also chapter 1.4.2 for additional info.
-stat	}
-static	} forces static linking (partially or fully), see .../etc/FS8config*
-v	Compiles in verbose mode. Displays the FlagShip release used, the compiler and linker phases and their switches.

-version	} Only display serial number & release of the FlagShip compiler
--version	} as opposite to -v switch which also process the compilation
-w	Issues a warning on all ambiguous (undeclared and unaliased) variables. Equivalent to -w1.
-w<value>	Sets a warning type during the FlagShip preprocessor and compiler phase. The warning types may be combined together, e.g. -w1 -w4

w0 : Does not display any additional warnings, but errors only. Equivalent to not using of the -w option and disables all -w<n> options set.

w1 : Issues a warning on all ambiguous (undeclared and unaliased) variables, but the first occurrence per function only. Purpose: checks for missing LOCAL, STATIC, MEMVAR, FIELD, PRIVATE and PUBLIC declarations.

w2 : Displays warnings for all untyped LOCAL and STATIC variables, but the first occurrence per function only. Purpose: checks missing AS <type> declarations.

w3 : Displays warnings for all unknown parameter and function return types. Purpose: checks missing PROTOTYPE <udf> declarations.

w4 : Displays warnings for all unknown classes, but on the first occurrence of the object per function only. Purpose: checks missing PROTOTYPE <class> declarations; all unknown class elements are late (run-time) evaluated. Refer to the section LNG.2.11.6 for detailed information. Hint: enable the #include "stdclass.fh" statement in the std.fh file or use the -i=stdclass.fh compiler option.

w5 : Report prototyped but not declared methods and instances.

w6 : Report automatic conversion of Fox array syntax.

-Wc,-<opt>	Passes the specified option <opt> to the C compiler and linker (cc). For example, -Wc,-omyexe is equivalent to -o myexe. Note: passing compiler or linker switches to MS-VC by this way does not work in all cases because of it position sensitivity. In doubt, include the switch directly in (a local copy of) FS8config file, see FSC.1.4.2.
-z	Suppresses shortcutting and optimizing logical operators .AND. and .OR. except in macro evaluation, which is always executed using the optimization. Refer to the section LNG.2.9 for detailed information.
-<option>	Passes the specified <option> to the C compiler and linker (cc). It is shorthand for -Wc,-<opt>, if the given <option> is none of the above valid FlagShip options. For example, -g will create additional C debugging information in the object file (used for adb, db, cv), and is the same as -Wc,-g. Note: using this "direct switch" depends on the used C compiler and linker.

Example of the option usage (note the allowed intermixing of file names and the compiler options):

```
$ FlagShip address.prg
$ FlagShip *.prg -Mmain -otest
$ FlagShip -na -nl -nL -nd -c -m -DTEST_ONLY *.prg
$ FlagShip -L../mylibs1 -Llibs2 [a-p]*.o      \
     -I/usr/john/myfh new.prg mylib1.a lib2.a \
     -Mmainudf -m -Wc,-otest
$ FlagShip -L../mylibs1 -Llibs2 [a-p]*.o      \
     -DSCOUNIX -lMylib libMylib2.a            \
     xyz*.prg -na -Mmyname  -urcc             \
     -U"-lFlagShip -ll -lcurses -lmalloc -lm -lPW"
$ FlagShip -umcc *.prg -Mmain -otest
```

For further examples, see following chapters.

1.3.1 Comparison between FlagShip and Clipper 5.x compiler options:

FlagShip:	Clipper 5.x:
-32	n/a (default = 16bit)
-64	n/a (default = 16bit)
-a	/P (/S)
-am	/V
-b	n/a, p-code only
-c	n/a, linker in DOS
-C	n/a
-d	/B
-D<name>[=<exp>]	/D<name>[=<exp>]
-delc	n/a
-dyn	n/a
-f	n/a (default)
-fox	n/a
-g	n/a, linker switches
-h	n/a
-I<path>	/I<path>
-i=<name>	n/a
-io=<mode>	n/a, console only, similar to -io=t
-iso	n/a, always PC-8 charset
-L<path>	/R<path>, linker in DOS
-l<file>	n/a, linker LIB <file> in DOS
-M<file|proc>	n/a, 1st module in DOS linker
-m	/M
-mdi	n/a, always SDI
-na	/N
-nc	n/a
-nd	not using /B

-nD	n/a
-nI=<file>	/U<file>
-nL	n/a, p-code only
-nl	/L
-nO	n/a, p-code only
-nodelobj	n/a
-ns	n/a
-o<name>	n/a, linker in DOS
-outdel	n/a
-pm	/A
-q	/Q
-r<opt>	n/a
-stat	n/a (default link mode)
-U<options>	n/a, linker options in DOS
-u<name>	n/a, linker name in DOS
-v	n/a
-version	n/a
-Wc,-<options>	n/a
-w	/W
-w<value>	n/a
-z	/Z
<file> <file> <file*>	@<clp file>

n/a = not available or not applicable.

1.3.2 Standard Define's

Depending on the FlagShip version and target system, there are different #define's specified in the <FlagShip_dir>/etc/FS8config* file.

`-DFlagShip`	always true in FlagShip
`-DFlagShip5`	true with FlagShip 5.x and newer
`-DFlagShip6`	true with FlagShip 6.x and newer
`-DFlagShip7`	true with FlagShip 7.x and newer
`-DFlagShip8`	true with FlagShip 8.x and newer
`-DFS_WIN32`	true with FlagShip for MS VisualStudio 32bit or 64bit
`-DFS_WIN64`	true with FlagShip for MS VisualStudio 64bit
`-DFS_BCC32`	true with FlagShip for Windows and BCC 32bit
`-DFS_LINUX`	true with FlagShip for Linux 32bit or 64bit
`-DFS_LINUX32`	true with FlagShip for Linux 32bit
`-DFS_LINUX64`	true with FlagShip for Linux 64bit
`-DVFS_FS2TOOLS`	true with available FS2 Toolbox

You may use `#ifdef .. [#else ...] #endif` directives in your .prg or .fh source to compile specific parts for different target platforms:

```
#ifdef FlagShip
     ... FlagShip specific statements ...
  #ifdef FlagShip8
     ... Visual FlagShip (FlagShip8.x and newer) specific statements ...
  #else
     ... FlagShip older than FlagShip8 specific statements ...
  #endif
#else
     ... Clipper, Fox etc. specific statements ...
#endif

#ifdef FS_BCC32
     ... MS-Windows with BCC33 specific statements ...
#endif
#ifdef MS_WIN32
     ... MS-Windows with VisualStudio 32 or 64bit specific statements ...
  #ifdef MS_WIN64
     ... MS-Windows with VisualStudio 64bit specific statements ...
  #else
     ... MS-Windows with VisualStudio 32 specific statements ...
  #endif
#endif
#ifdef FS_LINUX
     ... Linux 32 or 64bit specific statements ...
  #ifdef FS_LINUX64
      ... Linux 64bit specific statements ...
  #else
      ... Linux 32bit specific statements ...
  #endif
#endif

#ifdef VFS_FS2TOOLS
   ... FS2 Toolbox is available ...
#endif
```

You of course may set also you own #define's by the -D<name> compiler switch. Check the current compiler switches by "FlagShip -v ..." and watch output for -D... values.

Note: the #define names and values are **case sensitive**, i.e. the #ifdef FlagShip is **not** equivalent to #ifdef FLAGSHIP

1.3.3 Mode of operation

Extract from section LNG.1.2:

The FS8 compiler and library handles **three different i/o modes**:

GUI :	graphical oriented i/o, requires X11 or MS-Windows/32 or /64
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/Win64, Terminal on console, Basic otherwise. The detected environment can be overwritten by the command-line switch -io=g or -io=t or -io=b when invoking 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. These functions are invoked at start-up, before user INIT FUNCTION and user main UDF is executed.

When you create, or run the application in Terminal or Basic or hybrid mode directly from X11 or MS-Windows environment (i.e. not executing it from an open console window), FlagShip creates new temporary console window, which will be closed (after a short delay, see CMD.QUIT) when the application ends. You may avoid this by using -io=g compiler switch.

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.

Please refer also to section LNG.5.3 "Difference between Terminal and GUI" for additional programming hints.

1.4 Files Used by the FlagShip Compiler

1.4.1 Input files

When invoking the compiler, one or more source and/or object files of the application may be specified:

```
   $ FlagShip [options] <file names> ...
```

The FlagShip compiler accepts three categories of input <file names>:

- Source program code written in the FlagShip and other xBASE languages.
- Source code written in the C language (using the Extend or Open C interface system), and/or the .prg files, already translated into C.
- Object files created by the C compiler. Optionally, user or third party libraries may also be used for the current Unix or Windows system, but only when they are 100% compatible to your used C and linker.

The file extensions (which must always be specified) used for the input files of the FlagShip compiler are:

*.prg	is the default file extension for source files written in the xBASE language. See section LNG. Inline C code may also be included in the *.prg file.
*.fmt	is the extension of format files, used by the SET FORMAT command. FlagShip compiles them in the same way as the *.prg files. See section LNG and CMD.
*.bp	are already pre-processed .prg files, using the "std.fh" translation rules. Output of the -a compiler option.
*.c	are C source files, created by the programmer or by the FlagShip compiler. Also, output of the -b compiler option.
*.o	are Linux object files, containing native machine code, created by the C compiler (cc). These files are created by FlagShip, if the -a or -b compiler option is not given, or if -c is specified. They are either 32bit or 64bit, depending on used architecture. Check by "file *.o".
*.obj	are object files in MS-Windows, 32 or 64bit
*.a	are object libraries in Linux. A library, created by ar, is a collection of object files (32 or 64bit). On Unix, the library <name> is formed as "lib<name>.a" FlagShip accepts both the short naming convention (using `-l`<name>) and the full file name given as lib<name>.a
*.lib	are libraries in MS-Windows, depending on used architecture. Note that 16bit (DOS, Clipper etc) libs are incompatible to 32bit/ 64bit Windows, you need to create them (by FlagShip) for the used 32/64bit architecture.
*.so	are Linux dynamic object libraries, see chapter FSC.1.7 for additional information.
*.dll	are dynamic object libraries in MS-Windows

Additionally, #include files with the following extensions may be specified in the .prg or .c source code:

*.fh	which are FlagShip #include files used in .prg sources, containing all the translation rules for the standard and user- defined commands as well as definitions of #define identifiers and their values. Other extensions may also be used in the source code. The default std.fh file is used automatically, if the `-nI` compiler option is not given.
*.h	are C #include files, containing variable structures and other definitions. Used by the Open C system.

All input files may be preceded by a path specification. The other binary and text files used by the application are described in the section LNG.3.

1.4.2 Configuration file FS8config

To allow you the maximal amount of flexibility, the standard flags, libraries and options are automatically read from a configuration file named FS8config*. There are usually two different config files for each used architecture: FS8config_32 and FS8config_64 and selected by FlagShip compiler according to the -32 or -64 switch or FSARCH environment variable.

The config file is searched (in that order)

- in the current directory, then
- in the path given by the FS8CONFIG environment variable (if available),
- in the path given by the FSCONFIG environment variable (if available),
- in the /etc directory
- in the <FlagShip_dir>/etc directory

by the main module of the FlagShip compiler.

The FS8config* file is a usual ASCII file, modifiable with any text editor (but not by MS-Office, LibreOffice etc). To create and use locally modified copy of the FS8config* file, simply issue

```
$ cp <FlagShip_dir>/etc/FS8config_32 .
$ FSCONFIG=`pwd` ; export FSCONFIG     (not required for curr dir)
```

or on MS-Windows system

```
D:> copy <FlagShip_dir>/etc/FS8config_64
D:> set FSCONFIG=D:\<this_dir>
```

Structure of the FS8config* file: plain ASCII file, containing keywords followed by white space(s), colon (:) and an optional string setting. The keyword and the options are case sensitive, the keyword must start the line. Any valid keyword may occur only once, another same named must be commented out. Lines are terminated by NEWLINE and may not exceed 512 characters. Any line not beginning with a valid keyword is treated as comment, in-line comments are not supported.

Syntax: **<keyword><whitespaces>:<setting>**

<keyword>	identifier of FlagShip and/or CC options, see below
<whitesp>	any number of blanks or tabs
<colon>	separator between the keyword and settings
<setting>	everything, including white space, which follows the colon (:) and up to the new line is taken as is. If no setting is required, press <enter> followed the colon sign, or comment the keyword out (e.g. by prefacing the keyword with the # or * character).
# comment	comment lines (see also above)

Valid **keywords**:

FSDIR	Macro-holder definition for the <FlagShip_dir> path. The macro $(FSDIR) in following keyword-settings will be replaced by the content of FSDIR. If the content is empty, FlagShip will use automatically determined <FlagShip_dir> path from the location of FlagShip (or FlagShip.exe) compiler, i.e. the '/usr/local/ FlagShip8' or 'C:\FlagShip8' or 'C:\Program Files\FlagShip8' path. You may so specify e.g. the include or library settings. Note: when your <FlagShip_dir> path include blanks, you need to enclose the whole replaced command in double quotas.
FSPATH	Directory including the FlagShip* executables. If not specified, the path of the FlagShip command line invocation, and as last resort, the environment PATH is used.
CCPATH	Directory, including the CC executable (C compiler) or script. If not specified, the environment PATH is used.
CCNAME	Name of the C compiler, usually "cc" or "CL" or "BCC32".
CCDEBUG	Switch(es) to activate cc compilation in debug mode. If not specified, "-g" is used in Unix/Linux and "-Od -Zi" in Windows.
WINSYS_G	switch used for MS-Windows linker (in the POST* settings) when compiled with the -io=g switch. Default is -subsystem:WINDOWS
WINSYS_ATB	switch used for MS-Windows linker (in the POST* settings) when not compiled with -io=g switch. Default is -subsystem:CONSOLE
MACRO1..MACRO9	are nine user-defined macros. If specified, the $(MACRO1), $(MACRO2) ... $(MACRO9) tag defined below within the FS8config file will be replaced by the content of the macro definition. When the macro name is declared but it content is empty, the $(MACROx) tag is removed. Some of these macros may already be pre-defined for your convenience during setup, e.g. for C include paths, add-on libraries etc.

FSOPTIONS	Up to 32 FlagShip compiler options (see chapter FSC.1.3) separated by white space. Will precede the options given at the FlagShip command line.
PREOPTIONS	Options passed to the <CCNAME> compiler, preceding the file name. Usually define's, includes, optimization and debugger infos. Used only if neither -stat nor -dyn switch was given.
POSTOPTIONS	Options passed to cc following the file name. Usually linker options, libraries etc. Used only if neither -stat nor -dyn switch was specified.
PREDYNAMIC	Similar to PREOPTIONS but used with the -dyn compiler switch
POSTDYNAMIC	Similar to POSTOPTIONS but used with the -dyn compiler switch
PRESTATIC	Similar to PREOPTIONS but used with the -stat compiler switch
POSTSTATIC	Similar to POSTOPTIONS but used with the -stat compiler switch

Note: On some systems, there are two different options for static compilation and linking: either

- fully static (the default), where all libraries (lib*.a) are linked statically to your application, or
- partially static, where only FlagShip (and FS2 if available) libs are linked statically, all other dynamically.

The full static linking has the advantage that the application is fully independent of the target system (where the executable run), but it usually produces huge files and requires installation of C and X11/Windows static libs (lib*.a or *.lib respectively) on the developer's system.

The partial static linking use current dynamic system libs (lib*.so or *.dll in Windows) same as with dynamic linking, which may cause problems where the target system differs heavy from your current developer's system version, but the executable may be smaller than with the fully static linking.

The main difference between dynamic linking and partial static linking is, that with the second you don't need to distribute the libFlagShip_8*.so (or FlagShip_8*.dll) with your executable, see also SYS.1.2 for further distribution hints. So decide by yourself according to your needs.

To enable "partial static linking": edit the <FlagShip_dir>/etc/FS8config* file (or better a local copy of), comment-out (by # in front of) the default POSTSTATIC line (usually at line 40), and un-comment the second, by default commented-out POSTSTATIC line (usually at line 44).

Example of a modified FS8config file (some lines splitted here):

```
# My configuration file for the FlagShip compiler
# ------------------------------------------------
FSPATH     :/usr/local/FlagShip8
CCPATH     :/usr/bin
FSOPTIONS  :-q -w1 -w4 -DFlagShip8 -DFlagShip -I/usr/myinclude
CCNAME     :cc
PREOPTIONS :-DFS_LINUX64 -Oxp -I/usr/myHinclude
POSTOPTIONS:-L/usr/mylibs -L. -lMyLib -lFlagShip_8102_x64 -lmalloc -lm
```

```
PREDYNAMIC :-I/usr/local/FlagShip8/include -CSON -Oxp
POSTDYNAMIC:/usr/local/FlagShip8/lib/dynFlagShip_8102_x64.o \
   -lFlagShip_8102_x64 -lm
PRESTATIC  :-DFGSLINUX -DFGSLINUXELF -I. -I/usr/local/FlagShip8/include ...
   -L/usr/local/FlagShip8/lib -L/usr/X11/lib -fwritable-strings -fPIC
POSTSTATIC :-Wl,-Bstatic -lFS2_8102_x64 -lFlagShip_8102_x64 \
   -Wl,-Bdynamic -lm ... -lX11 -lXext -lXft -lSM -lICE -ljpeg -ldl -s
# eof
```

You may verify the passed switches by using the -v compiler option.

1.4.3 Output files

The output file of the FlagShip compiler is usually an executable

.exe	MS-Windows executable, named same as the main module or set by the -o or -e switch. You may rename it to any other *.exe name. The name is case insensitive.
a.out	Default name of executable in Unix/Linux. You may set any name with and without extension by using the -o or -e switch, or simply rename the file a.out to anything else (file must have "rx" permission). Note: if the current path or a dot is not in your PATH environment variable, you may need to invoke it as "./a.out". The file name is case sensitive.

Depending on the compiler option used, the .bp, .c and .o (or .obj) files as described in chapter 1.4.1 will also be created.

Additionally, prototypes of typed functions and classes are summarized in the **repository file**, named reposit.fh (or specified by -r=<name>), when the -rc and/or -ru switch is set (which is the default in the FS8config* file, see 1.4.2). Every time, the FlagShip compiler detects a declaration of a typed procedure or function, and/or the CLASS, ACCESS, ASSIGN or METHOD declarator, it adds the prototype into this global repository file. This file is for your convenience, to be able to extract the appropriate PROTOTYPES into your own #include file, or to directly #include "reposit.fh" in your source. Refer also to section FSC.1.3 and LNG.2.11.

1.4.4 Directories, Paths and Access Rights

FlagShip assumes that all the source and object files to be compiled and linked are available in the current directory. Otherwise, the path should precede the file name given. If the path or file name contains spaces, the whole entry must be enclosed in double quotas "...".

The #include files in .prg sources will be searched by the FlagShip compiler using the following algorithm:

1. Look for the file name as given in #include "File.Ext”
   a. the current directory
   b. the path given by the -I compiler switch (if any)
   c. in the `-I`... include directory specified in FS8config*, FSOPTIONS
   d. the /usr/include directory
2. Repeat step 1.a to 1.d with "file.ext" (in lower cases)
3. Repeat step 1.a to 1.d with "FILE.EXT" (in capital letters)

Note: if the *.ch include files of Clipper (except of std.ch) are used in unmodified ported applications, there are accepted too, since there are symbolic links in <FlagShip_dir>/ include/*.fh to corresponding *.ch. But do **not** use Clipper’s std.fh

During the linking process, the libraries *.a (or .so, *.lib, *.dll) will be searched by the linker using the following algorithm:

1. Look for the file in the current directory.
2. Use the -L path(s) to search, if one is specified.
3. Use the -L path(s) specified in FS8config*, keyword FSOPTIONS or PRE*
4. Use the default /usr/lib directory.

The output produced is always stored in the current directory, except when explicitly specified by the -o option.

During the compiler phase, you must have at least the following user, group or "others" access rights or common Windows rights (refer to LNG.3.3):

rwx	for the current directory (./),
r-x	for the /usr/bin, /usr/lib and /usr/include directory,
r--	for all the *.prg, *.fmt, *.fh and *.a files,
rw-	for all the *.bp, *.c, and *.o files,
rwx	for the executable file produced.

During the FlagShip and cc compiler or linker phase, temporary files are created in the /tmp directory. If an exported environment variable **TMPDIR** is specified, this path is used instead of /tmp. Make sure, you have full access rights (rwx) to the /tmp or $TMPDIR directory, otherwise a file creation error will occur.

1.5 Automatic Compilation

As described in the chapter 1.2, the simplest way to invoke the FlagShip compiler is to specify the main module only, for example:

```
$ FlagShip -32 address.prg -o adr.exe
```

which will compile all the program modules used in the whole application starting with the module address.prg, and produce an 32bit executable named adr.exe . The dependent modules are determined from the source statements

```
DO filename
EXTERN filename
SET PROCEDURE TO filename
SET FORMAT TO filename
```

Note: If automatic file detection is not possible, or not required, you may invoke the compiler specifying the desired file names, as described below in section 1.6.

This simple method is best suited to the initial compilation or to small applications. Unlike some DOS compilers, FlagShip will compile all the .prg modules called modularly, producing the equivalent *.o or *.obj (and the intermediate *.c) files for any .prg file used, and will perform the linking.

When changes of one or more of the used source files are necessary, a full recompilation may be time-consuming. In such a case, modular compilation of the changed source file is better and faster.

1.6 Modular Compilation

Suppose the application consists of the main module named address.prg, sub-modules named addr1.prg to addr5.prg and a format file named addr6.fmt. You may compile them all, using

```
      $ FlagShip ad*.prg *.fmt -m -Maddress -oadr.out
```
or:
```
      $ FlagShip address.prg ad??[1-5].prg   \
         addr?.fmt -m -Maddress -oadr.out
```
or:
```
      $ FlagShip -m -oadr.out address.prg    \
         addr1.prg addr2.prg addr3.prg       \
         addr4.prg addr5.prg addr6.fmt
```

In the first two examples, the file names specified by the wildcards *, ? or [...] are first expanded by the Unix shell which then passes all the names at once to the FlagShip compiler. MS-Windows does not support regular expressions via regex but only the * and ? wildcards. **Note:** Do not enter the trailing “\” above, but enter all in one line.

Now, when the module addr3.prg is changed, only this one module needs to be recompiled. The other modules can be linked at the object level only, using

```
    $ FlagShip addr3.prg a*.o -m -Maddress -oadr.exe
```

Such modular compilation may significantly speed up the creation of the executable. Of course, if two or more modules have to be recompiled, they should be specified in the same way. When using the "make" (or "nmake" in Windows) tool, this is done automatically, see section FSC.2

Using different compiler options

Where different program modules of the application must use different compiler switches, e.g. the -na option when file-wide STATICs or UDFs with the same name as the file name are used (for example the files addr3.prg and addr5.prg), compile them separately into objects, and use them for the link phase. Example for compiling all the above files using two different compiler options:

```
  FlagShip add3.prg addr5.prg -m -na -c       # 1st option
  FlagShip address.prg a*[1-2].prg \
      a*4.prg *.fmt -m -c                     # 2nd option
  FlagShip a*.o -Maddress -oadr.out           # link only
```

in the same way, you may also add C source files, created using the Open C System:

```
  FlagShip myc1.c myc2.c -c
  FlagShip a*3.prg *.o -m -Maddress -oadr.out
```

Or recompiling everything at once:

```
  FlagShip a*.prg *.frm m*.c -m -Maddress -oadr.out
```

Using a command-line-file

You may specify all required files and/or switches in a ASCII text file and pass the file name preceding by at-sign (@) to FlagShip command line, e.g.:

```
  # addr.cmd = command-line-file for address.exe
  address.prg                  // main file
  addr*.prg addr6.frm          // subsequent sources
  myc1.c myc2.c myobj.obj      #  other used files
  -m -Maddress -oadress.exe    # used switches
```

and invoke the FlagShip compiler with

```
  FlagShip @addr.cmd              -or-
  FlagShip @addr.cmd @other.cmd   -or-
  FlagShip @addr.cmd -d -v        -etc.
```

See also previous chapter FSC.1.2 for additional details.

Using a description file

Sometimes, if your application consists of many files, it is more comfortable to type the required file names only once and use an ASCII file as a description file for FlagShip again and again. You may use your text editor (e.g. vi) typing:

```
   $ vi myscript.txt
   <a>
       address.prg
       addr1.prg addr2.prg addr3.prg
       addr4.prg addr5.prg addr6.frm
       myc1.c   myc2.c
       myobj.o  mylib.a
       -m -Maddress -oadr.out
       -Wc,-lMylib  -L/usr/john/Mylib2 -I../myfh
   <esc>:wq
```

and invoke the FlagShip compiler with

```
  $ FlagShip `cat myscript.txt`             -or-
  $ FlagShip -nl `cat myscript.txt`         -or-
  $ FlagShip xyz.o `cat myscript.txt`       -etc.
```

Note the reverse apostrophe (`) in the entry. Also note that the length of the command line passed is usually restricted to 4 KBytes (sometimes only 2 KBytes), depending on the Unix system and shell type used.

In MS-Windows, you may use .bat file instead, including all the required compilation data.

Using a user-defined library

With many .prg programs used, it may sometimes occur that Windows or Unix shell cannot expand the whole command line, since it has limited size (usually 8 KBytes in Windows, 250 KB or more in Linux). Should a "shell buffer overflow" message occur, the best way to process an unlimited number of modules is to build a user library. Libraries are also often used to concentrate all object files used. Note: the "fsmake" tool (see FSC.6.9) will create and update user library semi-automatically.

a. Compile all sources modular into *.o objects, e.g.:

```
$ FlagShip [a-m]*.prg -m -c
$ FlagShip [n-z]*.prg -m -c
```

b. Build a user library (here named Mylib) by

```
$ ar rv libMylib.a [a-m]*.o                      ## Unix/Linux
$ ar rv libMylib.a [n-z]*.o                      ## Unix/Linux

C> LIB /out:Mylib.lib abc.obj bcd.obj other.obj  ## MS-
VisualStudio -or-
C> TLIB Mylib.lib +abc.obj +bcd.obj +other.obj   ## Windows with
BCC
```

c. Link (or compile and link) the application using

```
$  FlagShip -L. -lMylib -Maddress -oadr.out          -or-
$  FlagShip a*3.prg libMylib.a -Maddress -
oadr.out   ## Unix/Linux
C> FlagShip a*.prg Mylib.lib -Maddress -oadr.exe     ## MS-
Windows
```

d. If some modules need to be changed later, recompile them modularly, replace them in the library and link the application using:

```
$ FlagShip -m -c adr4.prg                    ## compile
$ FlagShip -m -c adr3.prg -na                ## separate modules

$ ar rv libMylib.a addr3.o addr4.o           ## Unix/Linux
$ FlagShip addr.prg -L. -lMylib -oaddr       ## Unix/Linux

C> LIB /out:Mylib.lib  adr3.obj adr4.obj     ## MS-VisualStudio -
or-
C> TLIB /u Mylib.lib adr3.obj adr4.obj       ## Windows with BCC
C> FlagShip addr.prg Mylib.lib -oadr.exe     ## MS-Windows
```

e. Refer also to chapter 1.7 for additional information.

Using a "make" utility

On a large application, the "make" utility of Unix or Borland, or Microsoft "nmake" will perform all the needed recompilation and linking automatically for all files changed. This is the most comfortable way, especially during program development. Refer to the chapter FSC.2 for more information. The tool "fsmake" (see FSC.6.9) helps you to create the Makefile.

Re-routing the Compiler Output

All screen output from the FlagShip compiler is sent to the stderr device. When a large application is compiled, the screen output may be redirected to any file:

```
$   FlagShip a*.prg -q -m -v -Maddress 2>err.log      ## Unix/Linux
C:> FlagShip a*.prg -q -m -v -Maddress 2>err.log      ## MS-Windows
```

Note the -q option, which suppresses the repetitive line number output and hence makes the output file better readable.

You also may (preferably) use the "script" shell command which echoes all the stdin and stderr output to specified text/log file (Unix/Linux only):

```
$ script err.log
$ FlagShip a*.prg *.fmt -q -c
$ FlagShip a*.o d.prg -q -m -Maddress -o myExe
$ exit  ## the script, creates err.log file
```

Compiling in the Background

To take advantage of the Unix and Windows multitasking facility, full or partial FlagShip compilation (where the most time-consuming task is the C compiler and linker) may be issued in background. Usually, the screen output is also redirected to a file to avoid disturbing other processes being performed in the meantime:

```
$   FlagShip a*.prg -q -m -Maddress 2>>err.log &    ## Unix/Linux
C:> FlagShip a*.prg -q -m -Maddress 2>>err.log      ## Windows
```

1.7 Libraries

The libraries libFlagShip_8*.[a,so] (or FlagShip_8*.lib on MS-Windows) are part of the FlagShip package, specially ported to the target Unix or Windows system. They includes all internal functions needed to run your application, as well as to perform various evaluations at run time, e.g. macros, dynamic variable access, objects, debugger, i/o and database system and so on.

By default, the FlagShip library will be installed in the <FlagShip_dir>/lib directory with a symbolic link to /usr/lib (in Unix). See section GEN.3. If the installation is done in another directory, you must tell the FlagShip compiler the path where to search for it, e.g.

```
   $ FlagShip -L/my/usr/lib myprog.prg
```

or in MS-Windows

```
   C:> FlagShip -libpath:c:\my\data\libs myprog.prg
   or C:> FlagShip myprog.prg c:\my\data\FlagShip_8106_x64.lib
```

Of course, you may preferably specify it in the local copy of the FS8config* file instead.

If additional libraries are used, they must be ported to the current OS and stored in the object library format (using "ar" or LIB respectively). Therefore, 16bit DOS object libraries cannot be used on 32bit (or 64bit) systems at all.

Static vs. Dynamic Libraries

For some systems, the FlagShip package is distributed with two libraries:

- the default static library named /usr/lib/**libFlagShip_8*.a**
- and a dynamic library named /usr/lib/**libFlagShip_8*.so** plus an object file named /usr/lib/**dynFlagShip_8*.o**
- or similarly FlagShip_8*.lib, FlagShip_8*.dll and dynFlagShip_8*.obj for MS-Windows systems.

The difference in the usage of static vs. dynamic libraries or DLL is:

- The required modules from the static library are fully integrated into the executable (a.out). You have to link "static" (often the default setting in <FlagShip_dir>/etc/FS8config*). It is then sufficient to distribute the executable (a.out) only, see SYS.1.2. The executable may generally be executed on a wide range of (sub)releases of the same operating system.

- When the dynamic library libFlagShip_8*.so (or the FlagShip_8*.dll) is available, you may alternatively link the application "dynamically" (system dependent, see Release Notes), e.g.:

  ```
  $ FlagShip -dyn myapp*.prg
  ```

or you may "burn-in" the path where the libFlagShip_8*.so resides by

```
$ FlagShip -dyn myapp*.o /usr/lib/dynFlagShip_8*.o -R/usr/mylibs
```

Same as in the statically linked application, Unix loads the same code only once and uses it for concurrently used executables. But, as opposed to static libs, when you run different FlagShip applications at the same time, the system will also share common object modules (standard functions) available in the dynamic library (e.g. the GET, i/o or database system, etc.)

The advantage are small executable files, whereby the application loads modules from the dynamic libraries (libFlagShip_8*.so and system's *.so) at run-time when they are required. Usually, there is no significant difference in the RAM requirements or usage, since these modules also have to be loaded in RAM.

The disadvantage of dynamically linked applications is, that you have to distribute both the executable (**a.out**) and the **libFlagShip_8*.so** library (but do **not** distribute the dynFlagShip_8*.o or libFlagShip_8*.a files). It may also sometimes be dangerous, if your customer has FlagShip applications from different vendors, to install the supplied libFlagShip_8*.so for common use, when the other vendor uses a different FlagShip release... Moreover, the dynamically linked applications are usually not so widely compatible to other (sub)releases of the same operating system, as static applications are.

So the decision to use the static or dynamic libraries is left to you depending on your needs and requirements. Generally, the best is to use dynamically linked applications in-house (or for well known customer's system) and to distribute the statically linked ones.

Note that the library name contains the VFS release and OS version (e.g. FlagShip_8101_x64.lib, libFlagShip_8101_x32.a, libFlagShip_8101_x64.so). Do not intermix them with other versions or OS version, unpredictable results or linker error may occur.

System Libraries

For linking, the standard C and Unix/Windows functions will be taken from the default malloc, curses, l, m, PW, kernel32, user32, shell etc. (system dependent) libraries. The default search path is /usr/lib in Unix or the one specified by the -L or -libpath: option. To change these defaults, or their order, specify it in a local copy of the FS8config* file.

User Libraries

If needed, you may also include your own (see chapter 1.6) or third party libraries by using

```
$   FlagShip myprog.prg /xyz/libMylib.a           ## Unix/Linux
C:> FlagShip myprog.prg "D:\my path\Mylib.lib"    ## MS-Windows
```

or as linker options

```
   $ FlagShip -Wc,-lMylib -Wc,-L/xyz myprog.prg      ## Unix/Linux
```

with the same results. See below for dynamic libraries (*.so or *.DLL). Any user library will be searched for externals before FlagShip and standard libraries. Of course, you may preferably specify it in the local copy of the FS8config* file instead.

Note: During the FlagShip and cc compiler or linker phase, as well as for the 'ar' librarian, temporary files are created in the /tmp directory. If an exported environment variable TMPDIR is specified, this path is used instead of /tmp. Make sure, you have full access rights (rwx) to the /tmp or $TMPDIR directory, otherwise a file creation error will occur.

To create a user library, use the default Unix 'ar' librarian or 'LIB' or 'TLIB' in Windows. Example to create the library 'Mylib' (fully named libMylib.a):

Modularly compile all the sources into *.o objects, e.g.:

```
   $ FlagShip *.prg -m -na -c
```

Build a user library (here named Mylib) with

```
   $ ar rv libMylib.a *.o
```

In MS-Windows, the LIB or TLIB utility is used instead of "ar" in Unix:

```
   C:> FlagShip abc.prg bcd.prg -m -na -c
   C:> LIB /out:Mylib.lib abc.obj bcd.obj       (VFS for MS-VC6)
   C:> TLIB /a  Mylib.lib +abc.obj +bcd.obj     (VFS for BCC32)
```

You may modify any library modules later, if changes in the source are re- quired, by recompiling and replacing that module e.g. 'abc.prg'

```
   $ FlagShip abc.prg -m -c -na
   $ ar rv libMylib.a abc.o                     (VFS for Linux, Unix)
   C:> LIB /out:Mylib.lib abc.obj               (VFS for MS-VC6)
   C:> TLIB /u  Mylib.lib -+abc.obj             (VFS for BCC32)
```

Note: the "fsmake" tool (see FSC.6.9) creates nearly automatically the Makefile template to create and/or update your user library.

When you replace library module by an object file linked in manually, the object module (based on .prg or .c file) must contain all public functions available in the library's module, otherwise linker error displays (see FSC.1.8 Linker Messages).

On some Unix systems (like SUN-OS), the 'runlib' utility must be run after the using the 'ar' librarian in the default library directory, see Release Notes REL.

Note: the order of the libraries specified is important, since most linkers search library symbols "forward" only. When using several user defined libraries, where the modules have mutual dependencies (e.g. one module required from Mylib1 calls a UDF in Mylib2 and another module required from Mylib2 invokes a UDF from Mylib1), one of the libraries must be specified twice:

```
$ FlagShip address.prg -m -lMylib1 -lMylib2 -lMylib1 -L../mylibs
```

For building a dynamic library, refer to the man pages of your linker (man ld in Linux/Unix). Usually, you build it from common object files via

```
$ ld *.o  -G  -o libMyLib.so
```

and then link your executable with the "-B dynamic" or "-dy" or "--dyna- mic" switch, e.g.

```
$ FlagShip address.prg -m -B\ dynamic -L. -lMyLib
$ export LD_RUN_PATH=$LD_RUN_PATH:./
$ ./a.out
```

On some systems (i.e. IBM AIX), the linker uses dynamic libraries per default (if the dynamic library is available), but you may force the static link by the "-bnso" linker switch (or the -Wc,-bnso and -Wc,-bI:/lib/syscalls.exp FlagShip switches). See details in the system dependant Release Notes (sect REL of your FlagShip manual) and the predefined settings in the <FlagShip_dir>/etc/FS8config* file.

You may use available **dynamic libraries** (.so in Linux or .DLL in Windows), either own (see above) or system libs or from third parties:

In Linux, simply compile/link with the library (e.g. libMylib.so in current dir, used in the other.c source):

```
FlagShip -c other.c
FlagShip myapplic.prg other.o -L. -lMylib
```

The .so library must match the used 32bit or 64bit architecture and the gcc and be located in /lib, /usr/lib, /usr/lib32, /usr/lib64 or in path specified by evironment var $LD_RUN_PATH.

In Windows, you need to link with .LIB file of the same name as .DLL, where the .LIB is a subset of .DLL containing these externals and addresses. The .LIB is automatically generated by the compiler when compiling the DLL.

```
FlagShip -c other.c
FlagShip myapplic.prg other.obj Mylib.LIB
```

If not available, you can create the .LIB from .DLL by

```
implib -c -a Mylib.LIB Mylib.DLL
```

in VFS/BCC, or for VFS/MSVC according to how-to description in *http://support.microsoft.com/?scid=kb%3Ben-us%3B131313&x=1&y=15* or in *http://adrianhenke.wordpress.com/2008/12/05/create-lib-file-from-dll*

Generally: the .LIB is used at compile/link time, the .DLL at run-time, where the .DLL must be accessible via PATH, or be located in current dir, or in the Windows system directory. Both the .LIB and .DLL library must match the used 32bit or 64bit architecture, which is set accordingly by using the "FlagShip8-32/64 Console" (icon on the desktop).

1.8 Compiler Messages

Since the FlagShip compiler performs several tasks (see FSC.1.1), different messages from these tasks may appear. All the messages are written to stderr and may be rerouted to a file, e.g.

```
   FlagShip -v my*.prg 2>myerr.log
```

Refer also to chapter FSC.1.6.

FlagShip Main Module Messages

If only "FlagShip" or "FlagShip -h" is entered, an on-line help which includes the possible options is displayed, for example:

```
   FlagShip: no filename given.
   Usage   : FlagShip <files>
             [-32|-64] Create 32/64bit executable/objects...
             [-a|-b]   Stop compilation after phase 1 or 2 of FlagShip
             [-am]     Declare all PUBLICs, PRIVATEs and undeclared ...
             [-c]      Suppress linking and <main>_m.c generation
   ...etc.
```

If a source <file> is entered without an extension, supported file extensions are displayed, for example:

```
   FlagShip: Illegal file name extension
   Regular extensions are: prg - FlagShip source
                           fmt - FlagShip fmt source
                           c   - C source
   ...etc.
```

Otherwise, the used license is displayed:

```
   FlagShip PRO (unlimited users)
```

and the compilation begin.

If the -v or -version option is given, the FlagShip release is added:

```
   FlagShip PRO (unlimited users)
   (c) Copyright ... Release 8.01.15.., Serial# ...abcdef
   for Linux (or MS-Windows)...
```

where "8.01" is the current FlagShip release used (8.1 is the main release number, and "15" is a sub-release). The serial number "...abcdef" represents the rightmost part of the complete serial number according to the Activation Document.

Note: when asking <support@flagship.de> for support or help, please always include the complete release number, operating system and the displayed FS serial number.

FlagShip Preprocessor and Compiler Messages

If -q option is omitted, the currently processed source file and line numbers are displayed:

```
FS compiler phase:
12345 Pass 1: std.fh               all used #include files
12345 Pass 1: file1.prg            Preprocessor phase
12345 Pass 2: file1.prg            FlagShip compiler phase

12345 Pass 1: std.fh               all used #include files
12345 Pass 1: file2.prg            Preprocessor phase
12345 Pass 2: file2.prg            FlagShip compiler phase
```

The actual output depends on the compiler switches used. All phases are displayed in the same line. The line number (here 12345) reflects the line number of the currently processed source (or include) file.

If a syntax or semantic error is detected, the preprocessor or FlagShip compiler displays the source line and the error location there, for example:

```
file1.prg: Error in line 543, expression
.........v
a := a +
** Warning: C compiler not invoked (1)

file2.prg: Error in line 29, expression
....v
SET AXACT ON
** Warning: C compiler not invoked (1)
```

Using the -w option, additional warnings may occur:

```
file1.prg: Warning in line 234, undeclared variable name.
file2.prg: Warning in line 456, ambiguous reference city.
```

When an error occurs, the FlagShip compilation will continue, displaying all errors detected, but the output file (e.g. *.c) will not be valid, and all the subsequent compiler tasks will not be invoked. Additionally, a return code 1 is generated, which may be detected in a shell script or in make. On success, return code 0 is generated. When warnings only are displayed, subsequent processing will be performed.

If you instead get a message like

```
1234  Pass1: filename.prg
** Warning: C compiler not invoked (1)
```

without any other displayed errors, it may be caused by stack overflow. Check the displayed line number (here 1234) in the source file (here filename.prg) for large continuation or large expressions. If so, use the -exp compiler switch for this source to increase the buffer size. In Linux, you alternatively may increase the stack buffer by e.g. "ulimit -s 20000", in Windows increase the stack in FS8config* file.

On some OS (usually in MS-Windows) this stack overflow may raise pop-up window message

```
   FlagShip_p.exe has encountered a problem and needs to close.
   We are sorry for the inconvenience.
```

whereby you need to click "Abort/Don't Send" and use the -exp switch, or reduce the number of continuation lines by splitting this statement at (or before) displayed line number in the source.

C Compiler Messages

When the -v option is used, FlagShip displays the C compiler used and its options, e.g.:

```
   C compiler and linker phase:
   cc file1.c file2.c -DSCOUNIX -lFlagShip -lcurses ...
```

Further output from the C compiler and the warnings and errors displayed depend on the system and compiler used. Refer to the corresponding man pages.

The message 'file creation error' signals insufficient access rights to the current, or to /tmp or to by environment variable $TMP (%TMP% in Windows) or $TMPDIR specified directory, or insufficient disk space otherwise. See also FSC.1.4 and FSC.3.3.

If you instead get message like

```
   -- FS compiler phase:
     1234  Pass2: filename.prg
   -- C compiler and linker phase:
   filename.c
   filename.prg(321): error C2026: string too long, remaining chars removed
```

it signals limitation of the C compiler (here MS-VC++) in support of large string constants. You will need to reduce the string length in your source (here line 321 of filename.prg) e.g. by + concatenation in two statements.

In MS-Windows, the message "Cannot open include file windows.h" signals that you either have not use the "FlagShip Console" setting, or you have re-defined environment variable INCLUDE set by MS-VisualStudio.

Linker Messages

Since the linker is invoked by the cc or CL (see above), FlagShip does not produce an additional message. The linker will report all the unresolved externals for files not linked-in or libraries not found. For example:

```
Undefined symbol:       First referenced in file:         error type

_bb_myudf               file1.o   or: file1.obj               (1)
_bb_xyz                 file2.o   or: file2.obj               (2)
_bb_substr              file2.o   or: file2.obj               (3)
waddch                  myCudf.o  or: myCudf.obj              (4)
MyFunct                 file3.o   or: file3.obj               (5)
myvar                   file2.o   or: file2.obj               (6)
fgs_fsDEMO440           file5.o   or: file5.obj               (7)
fgs_fs432               file6.o   or: file6.obj               (8)

myown.lib: fatal error LNK1136: invalid or damaged file       (9a)
myown.lib: Access violation. Link terminated.                 (9b)
ld fatal: Symbol referencing error, no output written.        (9c)
```

The output is similar in MS-Windows, where the message may slightly differ, see also below.

1: The FUNCTION MYUDF or PROCEDURE MYUDF was not found in the *.o (*.obj) files and libraries used. Note: FlagShip prefaces all UDFs and UDPs by **_bb_ prefix** to avoid interference with same named standard and user C functions. Usually, the requested module was not compiled/linked into the executable: check for the UDF <fnName> reported as _bb_<fnName> and add the corresponding .prg file to the list of files. If a large list of FlagShip standard functions _bb_<fnName> is displayed, the FlagShip library (libFlagShip_8*.a or FlagShip_8*.lib) was neither found in the default path <FlagShip_dir>/lib nor /usr/lib nor in the directory specified by the -L switch; or the library structure is incompatible to the used operating system (cc, ld, link).

2: The ANNOUNCEd module "xyz" was not found.

3: The FlagShip library was not found.

4: No system library (here curses) or user-defined C function (invoked from the C source) was found.

5: The C function "MyFunct" (specified as Inline-C or stand-alone C) was invoked by the CALL command without regard to case-sensitivity.

6: The typed variable "myvar", referred to via GLOBAL_EXTERN, was not defined elsewhere using the GLOBAL declarator. The same applies for an unresolved C variable which was declared externally.

7: The "file5.prg" was compiled with the FlagShip DEMO version 4.40 but another library (Personal of Pro) or release is used during the linking process.

8: The "file6.prg" was compiled with the FlagShip version 4.32 but another incompatible library release is used during the linking process.

9: The "myown.lib" is not of the same architecture (DOS,Clipper, or is 32bit for 64bit executable etc), or is incompatible to LINK in MS_VC (9a) or to TLINK in BCC32 (9b). Linux linker reports additional messages (9c).

Code-blocks (_bb_cb_<refer>) errors: if you get linker message like

```
-- C compiler and linker phase:
/tmp/ccKabmto.o: In function `_bb_cb_1_160_1':
/tmp/ccKabmto.o(.text+0x9f): undefined reference to `_bb_foo'
collect2: ld returned 1 exit status
```

it says, that you are invoking an UDF named foo() in a codeblock at approx line# 160, and this foo() is not linked in. Often the problem is obvious and easy to fix. If not so, you will need to search for the source name by "grep _bb_cb_1_160_1 *.c" or (when the *.c was deleted) in the objects by "for i in *.o ; do echo "--$i--" ; nm $i | grep _bb_cb_1_160_1 ; done" since at this stage, the linker does not report source file automatically and the /tmp/xxx file is only internal, temporarily link file. Note that many commands creates code blocks automatically (see std.fh), so refer to your .prg source at the line number associated with the code block name (here approx line 160) for the use/invocation of the reported, undefined function - here foo(). If a typo, fix the UDF name there, otherwise link the corresponding function to your application, see also (1) above.

```
-- C compiler and linker phase:
Error E2238 xyz.c 3945: Multiple declaration for '_bb_cb_1_326_2'
Error E2344 xyz.c 3565: Earlier  declaration of  '_bb_cb_1_326_2'
```

The automatically generated code block is defined twice in the same source file. This usually happens by #include'ing of *.prg files, see details and correction hints in section PRE #include. You may check the occurence(s) by "grep -in include *.prg | grep -i prg".

Other common linker messages:

- undefined reference to _bb_myudf()
- error LNK2001: undefined external symbol __bb_myudf

  Same as 1, 2, 5 above, or missing add-on library containing MyUdf() or assumed auto-declared procedure (same as .prg name) but compiled with -na switch

- xyz.o(.text+0x4321): multiple definition of '_bb_myudf'
- abc.o(.text+0x1234): first defined here

  The function/procedure MyUdf is defined twice in different object files or libraries (here in xyz.prg and abc.prg). Check also for -na switch.

- Warning: size of symbol '_bb_myudf' changed from 1234 to 4567 in xyz.o

  When also "multiple definitions.." arise: you are replacing library's module by your own, but have not included all public UDFs (available in the lib object) into your source file, see also FSC.1.7. Otherwise multiply defined UDFs in different files which contains also other, different functions.

- xyz.obj: error LNK2005: __bb_myudf already defined in abc.obj
- myapp.exe: fatal error LNK1169: one or more multiply defined symbols

  The function/procedure MyUdf is defined twice in different object files or libraries (here in xyz.prg and abc.prg). Check also for -na switch.

- FlagShip_8*.lib(initiomenu.obj): error LNK2005: __bb_initiomenu already defined in mymenu.obj
- myapp.exe: fatal error LNK1169: one or more multiply defined symbols

  You are replacing standard library's module (here initiomenu.prg) by your own, but have not included all public UDFs (available in the lib object) into your source file (here in mymenu.prg); see also FSC.1.7.

- fatal error LNK1181: cannot open input file 'libucrt.lib' (Windows) says that you either have not use the "FlagShip Console" setting, or you have re-defined environment variable LIB set by MS-VisualStudio.

2. Using the Make Utility

The standard Unix "make" (or Windows “nmake”) utility is a powerful program development tool which enables files to be kept up-to-date at all times. Using make, only the changed modules (more specifically: source files more recent than their objects) need to be recompiled, other unchanged modules will be linked only. This may speed up the development time significantly.

Not only in Unix and Linux, make is available also in MS-Windows. In Borland's BCC32 distribution is same named "make" utility available. In MS-VC is a comparable utility "nmake" available (simply invoke it by "nmake" instead of "make" described below). Also Rmake from Clipper fall in this category, but has some limitations. You may find short description of make on Unix/Linux by "man make", or for Windows in help for BCC32 / MS-VC or on
http://edmulroy.portbridge.com/howto5.htm
http://www.opengroup.org/onlinepubs/009695399/utilities/make.html
and large manual on *http://www.gnu.org/software/make/manual/make.html*

In this chapter, only commented examples of the make usage are given, since the make utility itself is detailed described in above documents.

In general, you describe the program developed and its modules in a text file called the make file. If the file is named "Makefile", the make utility will invoke it automatically. Otherwise, the file name has to be specified using the -f switch. The basic elements of a make system are the "dependency rule" which explicitly defines how each file is built, and the "inference rule" to specify what to do (by using source/target extensions) if no dependency rule applies. Both rules consist of a dependency statement and establish the relationship between the target file and a series of dependent files or extensions. The dependence or inference statement is followed by one or more actions, specifying an invocation of an executable or an utility by the shell. Additionally, series of variables and/or macros are generally used.

Note: The make utility works correctly only if the proper syntax of the make script is used. Do not insert leading spaces or TABs on the lines specifying a macro, variable or the rule declaration, e.g. in the examples below "ManModule = ..." or ".prg.o:". Use TABs at the beginning of the lines where spaces are displayed here, e.g. at the beginning of the line with the statement " FlagShip ..." or " addr2.o \" in the following examples. Otherwise, the make parser will report an error.

Note for Makefile's in Windows using MS-VisualStudio: **do not** specify variables/macros named INCLUDE and LIB in the Makefile, since these environment variables are already defined by VisualStudio.

Normally, one or a combination of the different examples described below will be suitable for compiling any FlagShip application.

For your convenience, there are Makefile examples in <FlagShip_dir>/tools available, supporting also files in different directories. There is also tool named **fsmake** there, which creates Makefile semi-automatically.

Using Dependency Rules

The dependency rule is suitable for small applications or when many different compiler switches are used. Let us assume the application consists of the main program address.prg and two modules addr1.prg, addr2.prg, to be compiled using different switches:

```
# -------------- make1.mak ------------------------------------
# execute: $ make                   using the file "Makefile"   ****
# execute: $ make -f make1.mak      using the file "make1.mak"  ****
# execute: $ nmake -f make1.mak in Windows/VC using "make1.mak" ****

#for Linux:
EXE = address
OBJ = o
#or un-comment for MS-Windows:
# EXE = address.exe
# OBJ = obj

$(EXE): address.$(OBJ) adr1.$(OBJ) \
       adr2.$(OBJ)
   FlagShip addr*.$(OBJ) -Maddress -o $(EXE)

address.$(OBJ): address.prg
   FlagShip address.prg -c

addr1.$(OBJ): addr1.prg
   FlagShip -m -nl -c addr1.prg
addr2.$(OBJ): addr2.prg
   FlagShip -m -na -c addr2.prg
# ---eof make1.mak---------------------------------------------------
```

The dependency rule is clear and easy to specify. The disadvantage is the need to specify every file name several times. However, you may combine dependency rules with inference rules, as described later. Note the usage of EXE and OBJ macro, which allows you simply port to/from Linux/Windows by commenting-out the corresponding definition. You of course may specify the object extension and executable natively w/o macros as shown in make3.mak to make5.mak below.

Using Inference Rules

This rule is compact and allows make files to be constructed to compile all the modules, while specifying them only once. The following example is an extended version of the make1.mak file above, using variables and macros but compiling all sources using the same switches. The produced executable here is adr.out.

```
# -------------- make2.mak ------------------------------------------
# execute: $ make                  using the file "Makefile"   ****
# execute: $ make -f make2.mak      using the file "make2.mak"  ****
# execute: $ nmake -f make2.mak in Windows/VC using "make2.mak" ****

StdOpt = -q -m
MainModule = address
#--- for Unix/Linux
EXE_EXT=
OBJ_EXT=o
#--- or for MS-Windows
# EXE_EXT=.exe
# OBJ_EXT=obj
ExeName = adr$(EXE_EXT)

.SUFFIXES: .$(OBJ_EXT) .prg .fmt .c

THEOBJECTS= \
   $(MainModule).$(OBJ_EXT) \
   addr1.$(OBJ_EXT) \
   addr2.$(OBJ_EXT)

.prg.$(OBJ_EXT):
   echo "**** compiling .prg files using options $(StdOpt) ****"
   FlagShip $(StdOpt) -c $*.prg
   rm $*.c

.fmt.$(OBJ_EXT):
   echo "**** compiling .fmt files using options $(StdOpt) ****"
   FlagShip $(StdOpt) -c $*.fmt
   rm $*.c
$(ExeName): $(THEOBJECTS)
   echo "**** linking to $(ExeName) using main $(MainModule) ****"
   FlagShip $(StdOpt) -M$(MainModule) $(THEOBJECTS) -o$@
.SILENT:

# ---eof make2.mak---------------------------------------------------
```

The advantage of the inference rule is the general compilation scheme using extensions only. To add or remove modules of the application, only the THEOBJECTS macro block needs to be changed.

When different compiler switches are necessary, or when too many files are used, either a combination of dependency/inference rules, or a special type of inference may be used:

```
# -------------- make3.mak ------------------------------------------
# execute: $ make                  using the file "Makefile"   ****
# execute: $ make -f make3.mak      using the file "make3.mak"  ****

StdOpt1 = -q -m
StdOpt2 = -q -m -na
MainModule = mymain
ExeName = myprog

.SUFFIXES: .o .oo .prg .c
```

```
.prg.o:
   echo "**** compiling .prg files using option $(StdOpt1) ****"
   FlagShip $(StdOpt1) -c $*.prg
   rm $*.c

.prg.oo:
   echo "**** compiling .prg files using option $(StdOpt2) ****"
   FlagShip $(StdOpt2) -c $*.prg
   ln $*.o $*.oo
   rm $*.c

.SILENT:

.IGNORE:

THEOBJECTS= \
   $(MainModule).o \
   prog1.o \
   prog2.o \
   prog3.oo \
   prog4.oo

$(ExeName).out: $(THEOBJECTS)
   echo "**** linking to $(ExeName).out ****"
   FlagShip -M$(MainModule) `echo $(THEOBJECTS) | \
      sed 's/\.oo/.o/g'` -o$@.out

# ---eof make3.mak---------------------------------------------------
```

The trick used here is defining two different rules for two different object extensions. The rule defining the dependence *.prg to *.o uses default compiler switches. For programs which should be compiled with other switches, a special *.oo extension, instead of the default *.o, is used to determine the dependency.

Combined Dependency and Inference Rules

For most large applications, inference rules only, or their combination with dependency rules will be best suitable. In the following example, all .prg and .fmt files changed are compiled according to the inference rules, except for the files adr2.prg, adr3.prg and myCudf.c where the dependency rules apply.

```
# -------------- make4.mak ----------------------------------------
# execute: $ make                  using the file "Makefile"   ****
# execute: $ make -f make4.mak     using the file "make4.mak"  ****

StdOpt = -q -m
TestOpt = -DTEST_ONLY
MainModule = address
ExeName = adr

.SUFFIXES: .o .prg .fmt .c
```

```
$(ExeName).out: $(THEOBJECTS)
   echo "**** linking to $(ExeName).out using main $(MainModule) ****"
   FlagShip $(StdOpt) -M$(MainModule) $(THEOBJECTS) -o$@.out

myCudf.o: myCudf.c /usr/myinclude/myH.h /usr/myinclude/myheader.h
   echo "**** compiling myCudf using cc ****"
   FlagShip -c -I/usr/myinclude $(TestOpt) myCudf.c

addr2.o: addr2.prg /usr/myinclude/address.fh
   echo "**** compiling addr2.prg files using special options ****"
   FlagShip $(StdOpt) $(TestOpt) -I/usr/myinclude -na -c addr2.prg

addr3.o: addr3.prg
   echo "**** compiling addr3.prg files using special options ****"
   FlagShip $(StdOpt) -na -nd -c addr3.prg

.prg.o:
   echo "**** compiling .prg files using options $(StdOpt) ****"
   FlagShip $(StdOpt) $(TestOpt) -c $*.prg

.fmt.o:
   echo "**** compiling .fmt files using options $(StdOpt) ****"
   FlagShip $(StdOpt) -c $*.fmt

.SILENT:

THEOBJECTS= \
   $(MainModule).o \
   addr1.o  addr2.o \
   addr3.o  addr4.o \
   xyz.o   \
   myCudf.o

# ---eof make4.mak---------------------------------------------------
```

Refer also to ready-to-run **examples** in <FlagShip_dir>/examples/make*

Using a user defined Library

The make file may also be used to build a user defined library, according to chapters FSC.1.5 and 1.6.

```
# -------------- make5.mak ----------------------------------------
# execute: $ make                   using the file "Makefile"   ****
# execute: $ make -f make5.mak      using the file "make5.mak"  ****

StdOpt = -q -m
MainModule = address
LibPath = ../mylibs/

Objects1= \
   $(MainModule).o \
   addr1.o  addr2.o \
   addr3.o  addr4.o

Objects2= \
   xyz.o   \
   myCudf.o

.SUFFIXES: .o .prg .fmt .c .a

a.out: $(LibPath)libMylib.a
   echo "**** linking to a.out using main $(MainModule) ****"
   FlagShip -M$(MainModule) -L../mylibs -lMylib -oa.out

$(LibPath)libMylib.a: $(Objects1)
   ar rv $(LibPath)libMylib.a $(Objects1)

../mylibs/libMylib.a: $(Objects2)
   ar rv ../mylibs/libMylib.a $(Objects2)

myCudf.o: myCudf.c
   echo "**** compiling myCudf using cc ****"
   FlagShip -c myCudf.c
   ar rv $(LibPath)libMylib.a myCudf.o

addr2.o: addr2.prg
   echo "**** compiling addr2.prg files using special options ****"
   FlagShip $(StdOpt) -na -c addr2.prg

.prg.o:
   echo "**** compiling .prg files using options $(StdOpt) ****"
   FlagShip $(StdOpt) -c $*.prg

.fmt.o:
   echo "**** compiling .fmt files using options $(StdOpt) ****"
   FlagShip $(StdOpt) -c $*.fmt

.SILENT:

# ---eof make5.mak---------------------------------------------------
```

3. Executing the Application

The compiled and linked application is called an "executable". In Linux, it is similar to the DOS/Windows .EXE file, whilst Linux may use any name with an *optional* extension. If the -o compiler switch is not specified, the executable created by FlagShip for Unix/Linux is named "a.out" by default. In Windows, without -o switch, the first source is the name of the executable.

To invoke the executable, the user must have at least "rx" access rights to the file and "rx" access rights to the directory. For more information on access rights, refer to section LNG.3.3. In MS-Windows, the directory and the .exe file must not have "H" and "S" attribute set.

If the application is executed on a computer other than the developer's one, ensure that the same hardware, operating system and the same or higher OS release is used on the target. See also section SYS for more information on distribution and porting.

3.1 Invoking the Application

To start the application, enter the name of the executable, optionally preceded by a path and/or followed by user arguments, separated by at least one blank character (see CMD.PARAMETERS or FUN.PARAM() for retrieving the command-line-arguments).

3.1.1 Invoking the Application in Unix/Linux

You may invoke the executable in Linux by

```
$ a.out                                        # no arguments
$ a.out param1 "param 2 with blanks"           # 2 arguments
$ myaddress user=25 /mono "/dir=$HOME"         # 3 arguments
```

If the current search path (or dot .) is not included in the environment variable PATH, use the full file name including its path, e.g.

```
$ ./a.out
$ ./a.out param1 "param 2 with blanks"
$ /usr/home/john/myaddress user=25 /mono
```

The arguments given on the command line are passed to the PARAMETERS variables of the main module.

The FlagShip application may also be executed in the background (not for Windows). The console messages may be rerouted to a file (or another terminal), for example:

```
$ ./a.out param1 param2 >progr.log 2>errors.log &
$ myapplic param1 param2 >output.log 2>&1 &
```

You may suspend or examine the background task and continue it later (e.g. using the kernel shell, Unix/Linux only):

```
$ jobs              (list background jobs)
  [1] a.out
$ fg %1             (run in foreground)
$ ^Z                (stop the execution; see stty -a, susp)
$ bg                (continue run in background)
```

If the application was compiled in hybrid mode (with -io=a or without -io= switch), you may specify the execution mode by the "-io=g" or "-io=t" or "-io=b" switch at command-line, given in front of all other user parameters:

```
./a.out -io=t myswitch "hello world"   ## executes in Terminal i/o mode
newfswin  ./a.out -io=t                ## executes in Terminal i/o mode
newfscons ./a.out -io=t                ## executes in Terminal i/o mode
newfsterm ./a.out -io=t                ## executes in Terminal i/o mode
./a.out -io=b                          ## executes in Basic i/o mode
./a.out -io=g                          ## executes in GUI mode
```

Similary in MS-Windows (see more in next chapter 3.1.2)

```
C:> myapplic -io=t                  ## myapplic.exe in terminal i/o mode
C:> d:\mydir\myapplic -io=b mycmd   ## myapplic.exe in basic mode
```

or by searching the PATH environment variable or by click on the executable in file manager (like Konqueror, NC, Windows Explorer etc.) or by click on a link at Desktop. In such a case, you will need to use full qualified file names or set the "current directory" by CURDIR() or by SET DEFAULT to locate your databases and other files used.

If the -io=<mode> is not specified, the used mode is detected automatically from the currently used environment.

If you have compiled the application with -io=g or -io=b or -io=t switch, the executable automatically use this i/o mode.

When you create, or run the application in Terminal or Basic or hybrid mode directly from X11 or MS-Windows environment (i.e. not executing it from an open console window), FlagShip creates new temporary console window, which will be closed (after a short delay, see CMD.QUIT) when the application ends. To avoid displaying of this console window in GUI mode, compile with -io=g switch.

If the first argument is "-FSversion", the current FlagShip release will be displayed for approximately three seconds, for example:

```
$ ./a.out -FSversion param1

This program was developed with
FlagShip Release 8.23.1234, Serial# ...98ac5d from July 31, 2016
(c) Copyright 1989..2016 by multisoft Datentechnik, Germany
unlimited users, for HP Apollo 9000, HPUX 9.0+
```

where "8.23" is the FlagShip release used for the compilation (8.2 is the main release, 3 specifies the sub-release), and "1234" is additional information for support purposes. The serial number displayed represents the last digits of the full serial number according to the Activation Card.

To examine the above message for any time period, press Ctrl-S or the PAUSE key, then Ctrl-Q (stty dependent) to continue the program execution.

If the "Eval/Test drive" was used to compile the application, the following message is always displayed, regardless of whether the "-FSversion" argument was given:

```
$ ./a.out param1

This program was developed with the Eval version of
FlagShip database compiler, valid until March 27, 2016
(c) Copyright 1989..2016 by multisoft Datentechnik, Germany
```

and additionally, also the release, if the "-FSversion" was given:

```
Release 4.43.1234, Serial# ..12abc56, for SCO UNIX V/386 3.2+
```

Prior to the program execution, you should check the environment variables and the system for correct settings, see chapters FSC.3.1 and FSC.3.2. You may set them automatically by using the **newfscons** or newfswin or newfsterm script, see chapter FSC.6.7 and sect. REL.

For additional information on RAM requirements, swapping and kernel tuning, refer to the section SYS.

3.1.2 Invoking the Application in MS-Windows

In MS-Windows, you may start it by similar way and options as described above in 3.1.1. Invoke either from cmd.exe window by specifying the name of your executable plus optional parameters

```
D:> MyApplic
D:> C:\temp\myApplic param1 "param 2 with spaces"
D:> ..\myapplic -FSversion -io=t param1 param2
D:> "\my path with spaces\MyApplic" param1
```

or by searching the PATH environment variable or by click on the executable in Explorer or on the link on Desktop. In such a case, you will need to use full qualified file names or (better) set the current directory by CURDIR() or SET DEFAULT to locate your databases and other files used.

3.1.3 Common Problems at Startup

- I get system message like "file not found" or similarly

  Check if the executable was created correctly, i.e. without compiler and linker errors. If so, on Unix/Linux you may need to invoke the application (usually named "a.out" when the -o <myName> switch was not given) including the path, e.g. "./a.out", when your PATH environment does not contain current directory search (a dot), i.e. if it has not a dot in the PATH like "xxx:.:xxx".

  Note: you may add in ~/.bashrc or ~/.profile a statement "`PATH=$PATH:.`" (without quotas) to search the current directory (which is often disabled per default for security reason). MS-Windows searches the current directory first, before evaluating the environment variable PATH.

  When your path or file name contain spaces, you need to enclose it in quotas, see example above.

- The application compiles fine, but the executable finishes immediately w/o any message

  Check for the start or first procedure/function, if any. When the name is other than the .prg name and not MAIN() or START(), you need to use the **-M**<udfname> compiler switch to specify the application entry point. This is also required when compiling several sources at once using wildcards, or when compiling the main program with **-na** switch.

- The application compiles fine, but the executable finishes too quickly, flashing shortly the application message(s)

  Add a WAIT or e.g. SLEEP(3) or INKEY(3) statement before QUIT or RETURN from your main procedure

- A popup message "stack overflow" displays (MS-Windows only)

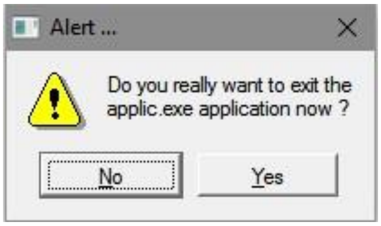
  MS-Windows does not use variable stack such Unix/Linux but of fix size, specified by the -Fnnnn compiler/linker switch. The default size is set in <FlagShip_dir>\etc\FS8config* to 4MB (-F4000000). If such message occurs for very large applications or very deep calling sequences, increase this switch e.g. to -F16000000. In other cases, it usually signals an infinite recursion in your application.

- Additional CMD/console windows pop-up at start of the GUI application

  Compile with -io=g switch, see also FSC.1.3

3.2 Aborting the Execution

In **GUI mode**, the application may be aborted by mouse click on the [X] button in the header of application window, or by selecting the File->Quit option.



Do you really want to exit the applic.exe application now ?

Alternatively, you may press the "Ctrl" and "K" key simultaneously (**^K**). Usually, a pop-up window displays asking you to confirm the termination

This behavior can however be freely re-defined by assigning your own UDF to public variable _MENU_QUIT via code block, e.g.

```
  PUBLIC _menu_quit := {|par1, par2| MyUdfAbort(par1, par2) }
```

On termination, the returned exit code may be set by global variables

```
  _aGlobSetting[GSET_N_RETURN_CTRL_K  ] := 1  // aborted via ^K
  _aGlobSetting[GSET_N_RETURN_CLOSE_EV] := 2  // aborted via event
  _aGlobSetting[GSET_N_RETURN_MENU    ] := 3  // aborted via menu
  _aGlobSetting[GSET_N_RETURN_ABORT   ] := 4  // other abort
  _aGlobSetting[GSET_N_RETURN_DEFAULT ] := 0  // standard exit
```

see also source of function InitIoQuit() in *<FlagShip_dir>/system/initiomenu.prg*

In **Terminal i/o mode**, the execution of your application may be aborted by pressing the "Ctrl" and "K" key simultaneously (**^K**). When the variable

```
  _aGlobSetting[GSET_T_L_ABORT_IOQUIT] := .T. // default setting
```

is set, the same behavior as in GUI mode apply, i.e. by using InitIoQuit() or re-directed user's UDF via _MENU_QUIT code block as described above. When you assign .F. to this variable, following message displays instead:

```
          +--------------------------------------+
          |    DO YOU REALLY WANT TO EXIT ??     |
          |                                      |
          |  Press Interrupt key again to exit.  |
          +--------------------------------------+
```

If you press ^K again, the program execution will be terminated, closing all open files and freeing the locks. If you press any other key, your application will continue running.

You may redefine the ^K break key by using the FS_SET("break") function. Program termination is possible only if it is not disabled by the SETCANCEL(.F.) function.

Terminating the foreground application by ^K will also terminate all child applications started from current in foreground or background.

You should **not** kill the application (like the "`kill -9`" or "`halt/shutdown/haltsys`" command in Linux), or Task-Manager in MS-Windows. Otherwise, lost changes and/or corrupted indexes may occur. See also LNG.4.8.6 and the QUIT command.

3.3 Environment Variables

Prior to program execution, the following shell environment variables should be checked and/or set:

3.3.1 Environment Variables for Unix/Linux

a. Environment variables used by the FlagShip compiler:

FSARCH=32	Optional, set the architecture for 32bit objects/executable. Same as the -32 compiler switch.
FSARCH=64	Optional, set the architecture for 64bit objects/executable. Same as the -64 compiler switch.
FS8CONFIG	Optional, path of the used FS8config_32 or FS8config_64 file. The default path is /usr/local/FlagShip8/etc
PATH	Standard Unix and Windows search path. Should include the directory of FlagShip executable.

b. Environment variables used at execution of the application:

FSERRORLOG Additional log protocol of RTE (run-time-errors) into ascii file. See section FSC.4 for details.

FSOUTPUT Optional, FlagShip specific. Path for the standard spool printer file (exename.nnnn), if the current directory is not to be used. Example:

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

or

```
$ export FSOUTPUT=$TMP    # if envir.var TMP is set
```

FSTERM Optional, FlagShip specific, Terminal i/o mode only. Equivalent to TERM, but may coexist with it. If specified, only FSTERM is used during the Curses initialization in the FlagShip application. Apply for Unix/Linux only. Example:

```
$ TERM=ansi ; export TERM
$ FSTERM=FSansi ; export FSTERM
```

FSTERM_SMALL Optional, FlagShip specific, Terminal i/o mode only. If set 1, accepts also terminal screen smaller than 56 cols x 15 rows, but the error message (RTE) and Alert() may loose some information. Use with care, since in special seldom cases segmentation fault may occur. If not set and the screen is smaller, corresponding message is displayed. Example:

```
$ export FSTERM_SMALL=1
```

FSTERMINFO

Optional, FlagShip specific, Terminal i/o mode only. Equivalent to TERMINFO, but may coexist with it. If specified, only FSTERMINFO is used during the Curses initialization in the FlagShip applic. Apply for Unix/Linux only. Example:

```
$ TERMINFO=/usr/lib/terminfo ; export TERM
$ FSTERMINFO=~/terminfo ; export FSTERMINFO
```

LANG

Optional, sometimes set to Unicode (UTF8). If difficulties with the point/comma decimal conversion (see fscheck.prg), or with semi-graphic in terminal i/o mode on X11, set this variable to:

```
$ LANG=english_us ; export LANG
```

or for X11 environment

```
$ LANG=en_EN.ISO-8859-1 ; export LANG
```

or

```
$ export LANG=C
```

Native Unicode is supported for input/output as well, see LNG.5.4.5

LINES, COLUMNS

Optional, Terminal i/o mode only. Overrides the terminfo specification of lines# and cols#, e.g.:

```
$ vidi e80x43                    (OS dependant)
$ TERM=FSansi ; export TERM
$ LINES=43 ;    export LINES
```

PATH

Standard Unix search path. For your convenience, it should also include the directory containing the current executable and/or "::" for the current directory search. This variable is also necessary for the correct execution of some commands like RUN. Example:

```
$ PATH=/usr/bin:/bin::$HOME ; export PATH
```

TERM

Terminal i/o mode only. Definition of the currently used terminal (preferably an FSxxx one for full support of special keys and the extended character set. See also section REL: Predefined Terminals and system dependent notes). Apply for Unix/Linux only. Example:

```
$ TERM=FSansi ; export TERM
```

TERMINFO

Optional, Terminal i/o mode only. Path of the terminfo definition (e.g. FStinfo.src), if not installed in /usr/lib/terminfo. Apply for Unix/Linux only. Example:

```
$ TERMINFO=/usr/home/myterm ; export TERMINFO
```

TMPDIR

Path used to create temporary files for the FlagShip and the C compiler, linker and librarian. If not set, the default /tmp directory is used. Make sure to have full access rights (rwx) to this directory and have enough free space available.

TZ Time conversion, mostly set to a specific time zone. Used for function TIME(), DATE() etc. See also "man environ" and "man tz". Example:

```
$ TZ=CET-1CEST;M3.5.0,M9.5.0/3 ; export TZ
```

SCRMAP Optional, FlagShip specific. Path for the language dependent sorting tables for INDEX and messages, when FS_SET("loadlang") is used and the required file is not in the current directory. The default FSsortab.def file used for sorting and for Upper() and Lower() translation of characters > 127. Example:

```
$ SCRMAP=/usr/data ; export SCRMAP
```

x_FSDRIVE Optional, FlagShip specific. Substitution of a DOS drive letter with a Unix directory. "x" represents an A...Z drive letter, e.g.:

```
$ C_FSDRIVE=/usr/data ; export C_FSDRIVE
$ D_FSDRIVE=/usr/data ; export D_FSDRIVE
```

FSPACKDIR If specified, path used to create temporary files during the PACK execution, when the available disk space on current file system is not sufficient. Make sure to have full access rights (rwx) to this directory and have there enough free space available for the .dbf database (and the associated .dbt or .dbv file).

LD_RUN_PATH Path specifying where the executable should search for dynamic libraries (e.g. libFlagShip_8*.so) if not installed in the default /usr/lib directory. Apply for Unix/Linux only. In Windows, the %SYSTEMROOT% environment is used instead.

LD_LIBRARY_PATH Path similar to LD_RUN_PATH.

FLAGSHIP_DIR Path used by some tools like newfs*, distribute* etc.

c. Environment variables used by the GUI debugger:

FSDEBUG_AUTO es auto save/restore debugger status, see FSC.5.1.
Enabl

FSDEBUG_COMPILER Path of the FlagShip executables (compiler) used by the GUI source-code debugger. The default setting is /usr/local/FlagShip8/bin in Unix/Linux and C:\FlagShip8\bin for MS-Windows.

FSDEBUG_INCLUDE Path containing the std.fh file for GUI source-code debugger. The default setting is /usr/local/FlagShip8/include or C:\FlagShip8\include

FSDEBUG_SOURCE Path containing the .prg source files used by the GUI source debugger. The default setting is the current directory. You may define several paths separated in Unix/Linux by colon(:) e.g.

	`/home:/home/src:/foo`, or by semicolon(;) in MS-Windows, e.g. `D:\src;\my\project\source;C:\xyz` Use absolute paths only.
FSDEBUG_TMPREAD	Path and file name containing a pipe used by the GUI source-code debugger. The default setting is /tmp/fs<pid>.dbgin
FSDEBUG_TMPWRITE	Path and file name used by the GUI source debugger. The default setting is /tmp/fs<pid>.dbgout

The above variables may also be set during log-in or in subsequent shells, when included in the file ~/.profile (or ~/.login using csh). Also, a start-up script including the settings may be used to correct the state of the environment variables.

To check the current values of the environment variables, use:

```
   $ env                                             -or-
   $ printenv                                        -or-
   $ echo $PATH
```

To check the specific environment variable from within a running application, use the GETENV() function. To assign a temporary value to an environment variable while the application is running, use FS_SET("setenv"). You may modify the behavior of the Curses initialization for special requirements, see section SYS.2.7.

3.3.2 Environment Variables for MS-Windows

a. Environment variables used by the FlagShip compiler:

FSARCH=32	Optional, set the architecture for 32bit objects/executable. Same as the -32 compiler switch.
FSARCH=64	Optional, set the architecture for 64bit objects/executable. Same as the -64 compiler switch.
FS8CONFIG	Optional, path of the used FS8config_32 or FS8config_64 file. The default path is C:\FlagShip8\etc
INCLUDE	Mandatory for the MS-VC environment, set by "FlagShip console" Paths for standard Windows include files, required for MS-VC.
LIB	Mandatory for the MS-VC environment, set by "FlagShip console" Paths for standard Windows libraries, required for MS-VC.
PATH	Standard Windows search path. Should include the directory of FlagShip executable.

b. Environment variables used at execution of the application:

FLAGSHIP_DIR Path used by some tools like distribute*.bat etc.

FSCONSOLE Optional, FlagShip specific for MS-Windows. Specifies the position and size of newly created console window in Terminal i/o mode and for stdout/stderr printout. Syntax:

```
C:> SET FSCONSOLE=x,y,w,h
```

Where x, y =	Position in pixel of the upper left window corner, default is a random position set by Windows. Value of -1 sets the horizontal and/or vertical window position centered on the desktop.
w, h =	width and height of the console window in columns and rows. 0 (zero) does not change the current setting.

Example: set the console window 10 pixels from the left and centered in the hight, containing 100 columns and 40 rows:

```
C:> SET FSCONSOLE=10,-1,100,40
```

You may set/modify this setting also by ConsoleSize() function.

FSERRORLOG Additional log protocol of RTE (run-time-errors) into ASCII file. See section FSC.4 for details.

FSOUTPUT Optional, FlagShip specific. Path for the standard spool printer file (exename.nnnnn), if the current directory is not to be used. Example:

```
C:> SET FSOUTPUT=C:\data\spool
```

or

```
C:> SET FSOUTPUT=%TEMP%
```

FSPACKDIR If specified, path used to create temporary files during the PACK execution, when the available disk space on current file system is not sufficient. Make sure that this directory exist and to have there enough free space available for the .dbf database (and the associated .dbt or .dbv file).

FSTERM_SMALL Optional, FlagShip specific, Terminal i/o mode only. If set 1, accepts also terminal screen smaller than 56 cols x 15 rows, but the error message (RTE) and Alert() may loose some information. Use with care, since in special seldom cases protection fault may occur. If not set and the screen is smaller, corresponding message is displayed. Example:

```
C:> set FSTERM_SMALL=1
```

LINES,
COLS Optional, Terminal i/o mode only. Overrides the CMD window specification (property setting), e.g.:

```
C:> SET LINES=25
C:> SET COLS=80
```

Note: trying to set LINES and/or COLS greater than current CMD window layout (see it property) is ignored (with message to stderr), otherwise MS-Windows would cause unpredictable errors.

x_FSDRIVE Optional, FlagShip specific. Substitution of a DOS drive letter with a Unix directory (supported also in MS-Windows). "x" represents an A...Z drive letter, e.g.:

```
C:> SET A_FSDRIVE=D:\user\data
C:> SET D_FSDRIVE=X:\common\files
```

This will map access of A: to D:\user\data and access of D: to the X:\common\files directory. Be careful not to produce so an infinite loop.

PATH Standard Windows search path. Local directory search is not required, since Windows searches it automatically. Example:

```
C:> SET PATH=%PATH%;D:\FlagShip8;E:\data\MyApplic
```

SCRMAP Optional, FlagShip specific. Path for the language dependent sorting tables for INDEX and messages, when FS_SET ("loadlang") is used and the required file is not in the current directory. The default FSsortab.def file used for sorting and for Upper() and Lower() translation of characters > 127. Example:

```
C:> SET SCRMAP=D:\user\data
```

TMPDIR Path used to create temporary files for the FlagShip and the C compiler, linker and librarian. If not set, the default Windows TEMP directory is used.

c. Environment variables used by the GUI debugger:

FSDEBUG_AUTO	Enables auto save/restore debugger status, see FSC.5.1.
FSDEBUG_COMPILER	Path of the FlagShip executables (compiler) used by the GUI source-code debugger. The default setting is <FlagShip_dir>\bin
FSDEBUG_INCLUDE	Path containing the std.fh file for GUI source-code debugger. The default setting is <FlagShip_dir>\include
FSDEBUG_SOURCE	Path containing the .prg source files used by the GUI source debugger. The default setting is the current directory. You may define several paths separated by semicolon (;) e.g. `C:> SET FSDEBUG_SOURCE=D:\my\project\source;C:\xyz` Use absolute paths only.
FSDEBUG_TMPREAD	Path and file name containing a pipe used by the GUI source-code debugger. The default setting is \temp\fs<pid>.dbgin
FSDEBUG_TMPWRITE	Path and file name used by the GUI source debugger. The default setting is \temp\fs<pid>.dbgout

The above variables may also be set during log-in when specified in the start->setup->system->extend->environment. You also may set them in a batch file, if such is used. In the **FlagShip-Console** window, they are usually already set at the time of FlagShip installation, check the file C:\Windows\system32\FlagShip_console.bat

To check the current values of the environment variables, simply type SET in cmd.exe or current window.

To check the specific environment variable from within a running application, use the **GETENV()** function.

3.4 System Settings

Beside setting the correct environment variables, the following system data and parameters must also be properly set.

3.4.1 System Setting for Unix/Linux

Date,Time

System date and time are essential for all time/date functions and for the execution of an evaluation license. Check or set it using:

```
$ date                  # Tue Dec 29 15:05:52
$ su                    # password ...
# date 1231160016       # format mmddHHMM[yy]
# exit                  # leave the su mode
```

Additionally, check the environment variable TZ which represents the time zone differences to GMT.

System i/o mapping

On some systems, an additional terminal mapping is set up during system installation. When using the extended input and output mapping of FlagShip according to the environment variable TERM (refer to the section SYS), the standard i/o mapping should be disabled, e.g.:

```
$ su                    # password ...
# machan -n             # disable tty mapping
# mapkey /usr/lib/keyboard/FSkeys.us
# exit                  # leave the su mode
```

The global setting may be included in the boot script /etc/rc2 or in the user's .profile script.

This mapping is generally OS dependent. Therefore, refer first to the section REL for system dependent notes and then to the section SYS for additional information on terminal input/output mapping.

stty settings

Check the terminal settings using stty -a. tab3, cs8, -istrip. The correct baud rate and parity should be set.

X11 setting

When running Terminal i/o based application under X/windows, the proper terminal TERM must be selected (see section REL) and an adequate character set with fixed width must be assigned in the current window. You also may use the provided "newfswin" script to do it automatically, see details in Release Notes.

When running GUI based application in X/windows, none special requirements needs to be met - except the used X11 version must fit to the used FlagShip port.

LANG setting

Some Linux versions use Unicode for the terminal and console, which avoids proper display of PC-8 character set in textual i/o mode. Check "echo $LANG" and if ...UTF8 displays, set

```
  export LANG=en_EN.ISO-8859-1
```

or your national LANG/ISO environment. You may set it also in the ~/.bashrc or ~/.profile or in the <FlagShip_dir>/bin/newfswin script. On some systems, you may need additional steps to disable Unicode, see http://www.fship.com/faq_vfs.html for current details.

Tunable kernel parameters

The Unix operating system may be adapted to your needs by tuning and rebuilding the Unix kernel. See details in section SYS.

On SVR4 Unix you may examine the (default, minimum and maximum) kernel values in the /etc/conf/cf.d/**mtune** file, and the current changes in the /etc/conf/cf.d/**stune** file (using e.g. cat /etc/ conf/cf.d/?tune). As super-user, you may also examine the system wide settings using e.g. the sar -v 1 command.

On Linux, the current settings are in /etc/**fstab** file and in the /proc directory.

3.4.2 System Setting for MS-Windows

Date,Time

System date and time are essential for all time/date functions and for the execution of an evaluation license. Check or set it using the DATE and TIME command or via Windows task manager.

4. The Run-Time Error System

FlagShip offers two different error systems, provided in source code (installed in the <FlagShip_dir>/system directory):

- The standard Error system, very similar to the Clipper's, accessible from the FlagShip language via error objects. (For detailed information about objects, refer to the section OBJ).
- The alternative, compact Run-Time error system. It reports errors in the FlagShip library, or detected by the run-time system or during macro evaluation.

The aim is to supply error reports with more detail on the nature, place and context of error. This should help the application designer to obtain a precise idea of where the error occurred as well as what caused it, without having to use extensive debugging.

There are four types of runtime errors: warnings, runtime, fatal and internal errors.

- **Warnings** generally occur in the developer mode when set by FS_SET("devel"), and signal minor errors, such as missing parameters, setting defaults etc.
- **Runtime errors** (RTE) are mostly due to programmer errors. Pressing the ESC key will continue the program execution, but a sequence of errors may then occur.
- **Fatal errors** inevitably abort further execution of the application. I/O errors are a subset of fatal errors.
- **Internal errors** should never occur, but if they do, they should be reported to the FlagShip customer support division.

Sometimes execution can be continued after an internal or runtime error. However, there may be unpredictable results. Much effort has been put into the error explanations and context description.

The FlagShip run time errors are weighted and depend on the current "developer" status. See also FS_SET("devel"). If the error is insignificant for the user (such as a wrong PICTURE), it will be not reported in developer mode, not in application mode.

The run-time error or warning is displayed in a small sub-window and contain the error-number (see also section APP), the corresponding function, it line number, and the textual description. In most cases, you have multiple choices how to react on it, e.g. abort the application (same as QUIT), ignore the error, get additional callstack information and so on. See further details in following chapters.

For support purposes, you may protocol the displayed RTE messages with additional debug and tracing data, as well as developer's warnings (if set by FS_SET() function) into an ascii log file by setting the environment variable FSERRORLOG :

FSERRORLOG=1	The name of error log file is created from the path and name of current executable by replacing the application's extension by ".err" For example, errors from the executable D:\tempdir\myexe.exe will be written into D:\tempdir\myexe.err, or from the ./a.out executable into file ./a.err (e.g. /home/john/a.err)
FSERRORLOG=2	The error log is written into stderr. You may then re-direct it to any device or file of your choice by executing "myexe 2>myfile" etc.
FSERRORLOG=filename	where <filename> is a name of the error log file, optionally prefaced with a drive letter and path, e.g. FSERRORLOG=/tmp/myerror.log or FSERRORLOG=C:\my path\ errors.txt

If the file already exists, the new message is appended. Otherwise, the log file is created in the current or specified directory. Examples:

```
  C:> set FSERRORLOG=1   # in MS-Windows: activate error log
  C:> myapplic           # protocol in <path of myapplic.exe>\myapplic.err
  C:> set FSERRORLOG=    # de-activate error log

  $ export FSERRORLOG=1  # in Unix/Linux: activate error log
  $ myapplic             # protocol into <path of myapplic>/myapplic.err
  $ unset FSERRORLOG     # de-activate error log
```

The error log behavior is user modifiable and available in the source file <FlagShip_dir>/system/FSerror.prg

4.1 Standard Error System

The standard Error system is supplied in FSerror.prg and the FlagShip library. It is object oriented and allows to integrate user supplied error handlers, see details in section OBJ.5.

Suppose, the following program is executed:

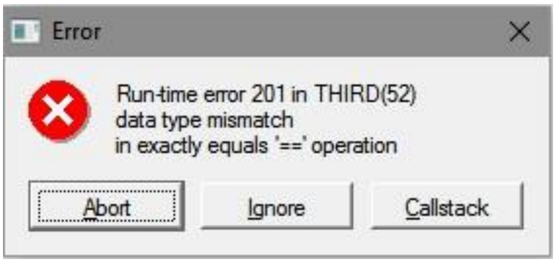
```
   1| *** file test.prg
   2| sub1()
   3| FUNCTION sub1
   4| DO other

  35| PROCEDURE other
  36| third(20)

  50| FUNCTION third(par)
  51| b := "today"
  52| ? par == b          // <-- RTE will occur (par is num, b is char)
```

When reaching the statement at the line 52, a run-time error will occur, since the comparison of different data types (numeric and string) is not possible. A pop-up window displays:

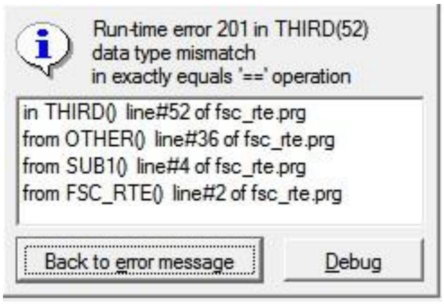
```
+-----------------------------------------+
|    Run-time error 201 in THIRD (52)     |
|           data type mismatch            |
|    in exactly equals '==' operation     |
|                                         |
|  Print  Abort  Ignore  Callstack  Debug |
+-----------------------------------------+
```

The textual lines in the box explain the error and its location (here the function THIRD, line 52). The error number corresponds to defaults in <FlagShip_dir>/include/FSerrors.h and error.fh and is described in detail in appendix (section APP) of this manual.

In GUI based application, a pop-up window is used with the same information:



Run-time error 201 in THIRD(52)
data type mismatch
in exactly equals '==' operation


Run-time error 201 in THIRD(52)
data type mismatch
in exactly equals '==' operation

in THIRD() line#52 of fsc_rte.prg
from OTHER() line#36 of fsc_rte.prg
from SUB1() line#4 of fsc_rte.prg
from FSC_RTE() line#2 of fsc_rte.prg

The user now has a chance to react, pressing the return key on the selected choice:

- **ABORT** terminates program execution with QUIT.
- **IGNORE** is present only on non-fatal errors. The current statement is ignored, the execution passed to the next source statement. Note: subsequent errors or unpredictable results may occur. Therefore, you need to confirm this choice in the following dialog

```
+---------------------------------------------------+
| Warning: Ignoring an error may cause subsequent  |
| follow-up errors or an unpredictable behaviour!  |
|  Are you sure, you want to ignore this error ?   |
|                                                  |
|  No,back to error message    Yes,ignore error    |
+---------------------------------------------------+
```

This confirmation dialog can be disabled by SET(_SET_WARN_IGNORE,.F.)

- **CALLSTACK** displays the call stack sequence, e.g.:

```
+-------------------------------------+
|       TEST  52: THIRD()             |
|       TEST  36: OTHER()             |
|       TEST   4: SUB1()              |
|       TEST   2: TEST()              |
|                                     |
|  Abort  Ignore  Errormessage  Debug |
+-------------------------------------+
```

The left column is the source code file name and line number (if available), the right column the corresponding PROCEDURE or FUNCTION name, similar to calling the standard PCALLS() function. The most important choices of the main error window are available again. Additionally, choosing the **ERRORMESSAGE** option returns you back to the main window.

- **DEBUG** is present only on non-fatal errors. It assumes the IGNORE choice and activates the FlagShip debugger (see FSC.5) for the next source line (or in the next program not compiled with the -nd option).

- **PRINT** prints the screen contents and the callstack into the current printer file. This option is available only if you recompile the FSerror.prg file. See below. It disappears once invoked to avoid subsequent printouts.

You may change the standard error handling by adopting the FSerror.prg file. Note: do not compile the FSerror.prg file in the original directory, since the alternative FSerror.c system file will be overwritten. Instead, if changes are required, copy the FSerror.prg file into your working directory and add the file name when invoking FlagShip.

A user supplied error code block may replace the default error handler, but may execute the default one from within. See details in section OBJ.5.

4.2 Alternative Error System

In the <FlagShip_dir>/system directory, there is file FSerror.prg available which may be freely modified. It also supports output of error messages to text file (or to additional screen). To enable it, issue:

a) MS-Windows: in FlagShip console

```
cd \my\src
copy %FLAGSHIP_DIR%\system\FSerror.prg
FlagShip -na -c -w -m FSerror.prg [-FSintern]
FlagShip myapplic*.prg -Mmyapplic FSerror.obj
set FSERRORLOG=myapplic.err
myapplic
type myapplic.err   # if error(s) occurs
```

b) In Linux:

```
cd ~/my/src
cp /usr/local/FlagShip8/system/FSerror.prg .
FlagShip -na -c -w -m FSerror.prg [-FSintern]
FlagShip myapplic*.prg -Mmyapplic FSerror.o -o myapplic
export FSERRORLOG=myapplic.err
./myapplic
cat myapplic.err   # if error(s) occurs
```

Additional information is available in the FSerror.prg source.

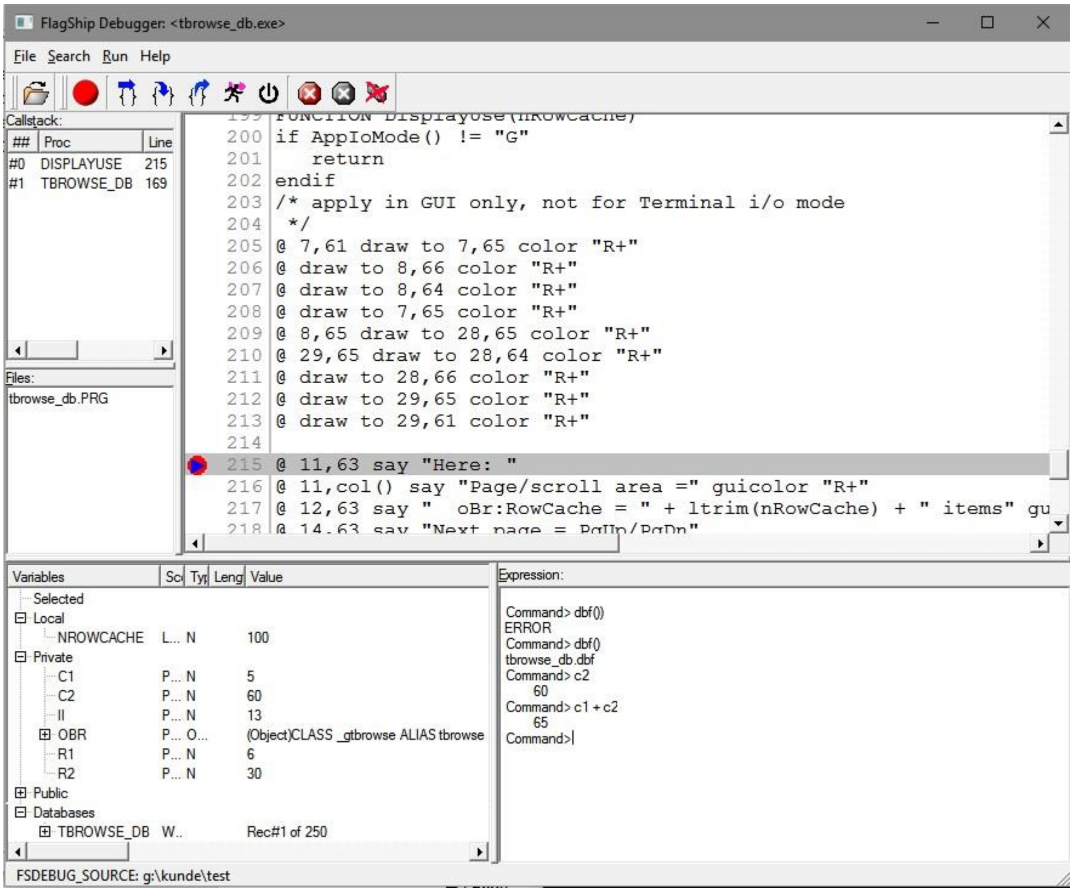
5. The FlagShip Debugger

The design philosophy of FlagShip was to give most of the necessary information in the error report itself so that little or no debugging is required.

There are two different FlagShip debuggers available. The GUI debugger is fully featured, whilst the Terminal i/o debugger lets you perform the most significant debugging actions. Both are described below.

5.1 GUI Source-Code Debugger

When compiling the application **with the -d switch** (and the line numbers was not disabled by the -nl switch), debugger window appears at program start of the GUI based application:



```
## Proc         Line
#0 DISPLAYUSE   215
#1 TBROWSE_DB   169
```

```
tbrowse_db.PRG
```

```
FUNCTION DisplayUse(nRowCache)
if AppIoMode() != "G"
   return
endif
/* apply in GUI only, not for Terminal i/o mode
 */
@ 7,61 draw to 7,65 color "R+"
@ draw to 8,66 color "R+"
@ draw to 8,64 color "R+"
@ draw to 7,65 color "R+"
@ 8,65 draw to 28,65 color "R+"
@ 29,65 draw to 28,64 color "R+"
@ draw to 28,66 color "R+"
@ draw to 29,65 color "R+"
@ draw to 29,61 color "R+"

@ 11,63 say "Here: "
@ 11,col() say "Page/scroll area =" guicolor "R+"
@ 12,63 say "  oBr:RowCache = " + ltrim(nRowCache) + " items" gu
@ 14,63 say "Next page = PgUp/PgDn"
```

```
Variables         Sc  Typ Leng Value
  Selected
⊟ Local
    NROWCACHE     L... N       100
⊟ Private
    C1            P... N       5
    C2            P... N       60
    II            P... N       13
  ⊞ OBR           P... O...    (Object)CLASS _gtbrowse ALIAS tbrowse
    R1            P... N       6
    R2            P... N       30
⊞ Public
⊟ Databases
  ⊞ TBROWSE_DB    W..          Rec#1 of 250
```

```
Command> dbf())
ERROR
Command> dbf()
tbrowse_db.dbf
Command> c2
    60
Command> c1 + c2
    65
Command>|
```

```
FSDEBUG_SOURCE: g:\kunde\test
```

```
+-----------------------------------------------------------------------+
| Menu bar ...                                                          |
| Toolbar ...                                                           |
+------------------+----------------------------------------------------+
| Callstack window | Source window                                      |
|                  |                                                    |
|                  |                                                    |
+------------------+                                                    |
| Files window     |                                                    |
|                  |                                                    |
|                  |                                                    |
+------------------+--------------+-------------------------------------+
| Variable window                 | Expression window                   |
|                                 |                                     |
|                                 |                                     |
|                                 |                                     |
+---------------------------------+-------------------------------------+
```

The debugger window is re-sizeable by clicking (and holding) the left mouse on the window edges or on the vertical/horizontal frame. The sub-windows are re-sizeable by clicking (and holding) the left mouse on the frame of the corresponding sub-window.

In the Debugger **header**, the name of the executable is displayed.

The **Menu Bar** contain following entries:

—> File

• Open Source	=	Open new source file displayed in "Source window" You will need to use this menu to display sources not available in the current directory.
• Restore Status	=	Restore breakpoint status saved by "Save Status" The checkbox is set when the debugger status was restored manually or automatically by FSDEBUG_AUTO.
• Save Status	=	Save breakpoints and selected variables to a file of your choice. Pre-set is <exefile>.debug in the current directory. The checkbox is set when the debugger status was already saved in this session.
• Auto Save	=	Automatically save breakpoints and selected variables to file specified by FSDEBUG_AUTO environment variable. You may toggle (enable/disable) this feature by click on the menu item. Not available when FSDEBUG_AUTO environment variable is not set or is empty.
• Source Path	=	additional path/directory of source files
• Quit Debugger	=	Quit debugger. Restart with Ctrl-Q or Altd()
• Quit Program	=	Quit the application

—> Search

• Find	=	find a pattern in current source
• Find Next	=	continue searching
• Goto Line	=	skip to specified line number

—> Run

• Interrupt	=	interrupt running application, stop in debugger same as click on the green or yellow run-button. The interrupt can also be activated by pressing ctrl-O (^O) key, redefinable by FS_SET("debug"), or via the ALTD() function.
• Breakpoint		
• Set/Clear	=	set or clear breakpoint at current source line
• Enable/ Disable	=	if a breakpoint is already set, disable it temporarily. It is enabled with a new selection of this menu being currently on disabled break
• Clear all	=	delete all breakpoints
• Step-over	=	step to next executable statement but do not step into user defined Function/Procedure
• Step-in	=	step to next executable statement. If the current pointer is a UDF call, step into the function
• Finish Funct()	=	execute the UDP/UDF until RETURN or next breakpoint occurs. Stop on the next executable statement in the caller UDF. If the current UDF is top-level function, the execution continues until QUIT.
• Continue	=	continue execution until next breakpoint - or finish the execution if no breakpoint applies.

—> Help

Displays on-line help, same as <F1> key

The **Tool Bar** is a shorthand of some most important Menu-Bar entries. When the mouse cursor is over the tool bar picture, a short description is displayed. A color signal button specifies the current debugger status:

• Red signal	=	the application is interrupted in debugger
• Yellow signal	=	application is running partially, "Step" button pressed
• Green signal	=	application is running (up to next breakpoint), "Continue" button pressed

Clicking on the signal button interrupts the application and stops in debugger at the next valid source line.

Callstack Window

In the Callstack Window, the current stack of the application is displayed, similarly to ProcName(n) and ProcLine(n). The current position is marked as #0 same as in ProcName(0), the parent UDF (if any) is marked with #1 etc. With left-mouse-double-click on the callstack item, you can switch the corresponding Source display.

Files Window

The Files Window displays the selected (or auto read) source files. The current source file is searched automatically (if not yet open)

- in the local directory, or/and
- in a path specified by the environment variable FSDEBUG_SOURCE (you may define several paths there, see details in FSC.3.3)

in that order, whichever apply first. The auto-read source file may not exceed 10 characters plus the ".prg" extension. Only lowercase file names are auto-considered by the debugger in Unix/Linux. You can add additional sources in any directory and of any name via Menu->File->Open Source. With left-mouse-double-click on the File item, you can switch the corresponding Source display.

Source Window

The current or selected source is displayed in the Source Window, including the corresponding line number. You can scroll the view with the vertical and/or horizontal scrollbar. The currently selected line is visually marked by a bar.

The program execution point is marked by a shadow line and a small triangle left of the line number and denotes the .prg source line which will be executed next (i.e. the marked line was not executed yet). You may watch variables available at this point in the "Variables Window" or perform any command or expression in the "Expression Window".

Breakpoints: You can set, disable/enable or clear a breakpoint

a) at the currently selected line
   - by by a click on the "Breakpoint" icon
   - by Menu->Breakpoint selection
b) at any line in the Source Window
   - by a left mouse click on the displayed line-number

An active breakpoint is displayed with a red point left of the source line number, a temporarily disabled (inactive) breakpoint by a gray point. The application stops every time when an active breakpoint is reached, you can then view variables and databases, perform an expression, or use "Step-over"/"Step-in" to process the current statement, use "Finish Funct()" to finish the UDF and stop thereafter, or click "Continue" to continue the program execution until the next breakpoint is reached.

The breakpoints cannot be tested for statement validity, so breakpoint set in e.g. comment line will simply be ignored. Since some commands are translated to several functions (see std.fh and/or *.bp pre-processor file created by -a switch), you may sometimes need to click "Step" twice (or even more) to continue the execution to the next visible source code line.

The number of breakpoints is not limited. You can save the current break- points and restore them later (e.g. after re-compiling the application) by Menu->Save/Restore Status.

Variable Quick-View: When the mouse cursor is placed over a valid and visible variable in the Source Window, the variable name and it content is displayed in a small pop-up window (tool-tip).

Variables Window

In the Variables Window, you can view or change variables and databases available at the current program execution point. Of course, the variables, it content as well as open databases are valid only when the debugger stops (yellow or red signal shines). Initially, there are five items visible:

- **Selected** variables thread allows you to specify any variable which you will explicitly watch. Click on the "Selected" item and use right mouse to select, or use the "Insert" key. To delete a variable from the watch, click on the variable and use either "Delete" key or right mouse click. See also performance hint below.
- **Local** (and static) variables thread If some local, static, local parameter or self: variables are currently available, a [+] sign is displayed left of the "Local" item. A left mouse click on the [+] sign opens the thread view of currently visible vars or local parameters. If the variable is of a compound type (array or object), a new thread marked with [+] display which can again be opened by a mouse click. You may change the variable content by left-mouse- double-click on the corresponding variable, or via := assignment in the Expression window. To hide the thread, click on the [-] sign.

  Note that local and static variables are visible only when the current source module was compiled with -d switch. C-like-typed variables are not visible at all, since already resolved directly to addresses by the compiler (but are visible in C debugger like gdb, kdebug, ddd, cv etc. see 5.3 below).
- **Private** variables thread is similar to the Local thread, except it displays the currently visible private and auto-private variables. Note: if the same variable name is declared also Local, the application prefer the Local instead of the Private variable, see also section LNG.2.6.3 for further details.
- **Public** variables thread is similar to the Local thread, except it displays the currently visible public variables and constants. Note, the public variable may be hidden by a local or private variable of the same name, see also LNG.2.6.3 for details.
- **Databases** thread displays the fields and it content of all currently available working areas, if any. Any open database is marked by [+] preceding the used Alias and some important database characteristics. You may open the thread by the same way as with other variables, i.e. by a click on the [+] sign. You will then see the Area number, database fields and it content, which can also be changed same as with variables.

  The currently selected working area is always the first in the Database thread. You may retrieve additional details about the database using the "Expression" window and corresponding command or function, e.g. "Eof()", "myAlias->(IndexCount())" or "DbObject('myAlias'):OrderInfo(24)" and so on. Note: in the application, a field of the current working area is preferred instead of private variable with the same name, see also section LNG.4.2 for further details.

You may **resize** the column width of the Variable-Scope, Type, Length, Value and Address by mouse click & movement on the vertical bar in the variable header.

Expression Window

In the Expression Window, you may enter any command or expression, same as in your program source. You also may use public, private, local or static variables to build an expression. If the command or expression cannot be evaluated, corresponding message is displayed, otherwise the expression result is shown. You may repeat (and change) the command/ expression entry by using the Cursor-Up and Cursor-Down key.

There are only few limitations in the Debugger Expression compared to compiled source code:

- You cannot use mnemonic constants (e.g. from *.fh files - except of those header files defined in the std.fh, i.e. except of set.fh and inkey.fh) but need to enter the corresponding constant instead. The same is valid for #define, #command or #translate directives specified outside of std.fh.
- The second difference is, that commands and expressions are interpreted in slightly simplified way from the compiled application source: first, a simple macro evaluation by the run-time system is tried, considering the preference of locals. If this fails, the FlagShip compiler is used to translate the entry according to std.fh file, the result is displayed and tried to be resolved by the run-time system anew.
- C-like-typed variables cannot be supported at all within the debugger, since these are resolved directly by the compiler to an object address.
- Only already linked-in standard functions are available in the Debugger Expression window. If en error is reported on processing standard function, you will need to use "EXTERNAL <FunctionName>" somewhere in the application, see also section CMD:EXTERNAL

Notes:

The full expression evaluation requires **installed FlagShip** of the same version as the application, so the command translation will not work properly at customer's site. This is usually also not required, since the GUI debugger needs the corresponding source code, which is seldom distributed. Best to compile/link the final, distributed executable **without the -d** or with the -nd switch, where the debugger will not be available at all, see (e) below.

The source-code window does not consider **sources** included by the #include directive and may here interpret the source-code line numbers incorrectly. To avoid it, use the #debug_off and #debug_on directive in the included source (or before/after the #include "mysource.prg" directive), see section PRE and example in the .../include/fspreset.fh file. This restriction does not apply for including .fh files containing preprocessor directives only.

You may disable (hide) the debugger window by click on the [X] button in the debugger window header or by pressing Ctrl-Q in Debugger, and activate (view) it anew by pressing Ctrl-O or by executing the Altd() function.

You may activate the debugger also by the Ctrl-O (^O) key, redefinable by FS_SET("debug",key). This of course apply only if compiled by -d switch.

Performance hint: if you have many variables and use single step, all the open threads are updated on any halt. So if only few variables are of current interest, you may add them to "Selected Variables" and close other variable threads - the application will perform faster.

To speed-up the development, i.e. to avoid manually restoring already set breakpoints every time you restart the application with debugger, you may set environment variable **FSDEBUG_AUTO=**ON or FSDEBUG_AUTO=file.name, e.g.

```
   export FSDEBUG_AUTO=ON   on Linux or
   SET FSDEBUG_AUTO=ON      on MS-Windows.
```

This will instruct debugger to read previously saved debugger status at program start and to save current status automatically at program end.

FSDEBUG_AUTO=ON	uses the default debugger status file name <executableName>.debug
FSDEBUG_AUTO=filename	uses the given file name instead
FSDEBUG_AUTO=OFF	disables auto save/restore
FSDEBUG_AUTO=	same as FSDEBUG_AUTO=OFF

When you start new project with the same executable name (e.g. a.out) and FSDEBUG_AUTO is ON, simply delete the old file <executableName>.debug to create new debugger session.

Optional environment variables used: FSDEBUG_AUTO (see above) as well as FSDEBUG_COMPILER, FSDEBUG_INCLUDE, FSDEBUG_SOURCE, FSDEBUG_TMPREAD and FSDEBUG_TMPWRITE, see description in section FSC.3.3

Compilation notes:

a. To activate and link the GUI debugger into the application, the linking stage must include the -d and may not include -nl nor -nd compiler switch.
b. If compiled modularly: to enable access to local variables and self: properties, the module must be compiled with the -d and may not include the -nl or -nd compiler switch.
c. If you want to test in-line C code (Open-C API) together with the .prg code, you need to use -d and -g switches (optionally also -nL, but not -nl). Also the linking stage must be compiled with -d and -g. Start the C debugger (see FSC.5.3) first and set corresponding breakpoints in the C part, then "run" the application from the C debugger.
d. To disable any already tested, modularly compiled source module from the debugger view, compile it with the -nl or -nd switch.
e. To disable debugger at all in the released application, at least the link stage must be processed without the -d compiler switch. If you recompile also all other modules without the -d switch, it will decrease the size of the application slightly.

5.2 Terminal i/o Debugger

The Terminal i/o debugger is based on the run-time evaluator. Therefore, the same rules apply as for the macro evaluation. Only the visible PRIVATE, PUBLIC and FIELD variables can be used. LOCAL and STATIC variables are only visible when the compiler option -d is used. C-typed variables are invisible for the debugger.

When executing UDFs or some standard functions by the debugger you will receive an error "undefined function" if the function is not referenced elsewhere in the application. To make the function available, declare EXTERNAL <udfname> anywhere in the application in order for the linker to include it into the executable.

Activating the debugger:

It can be activated by pressing ctrl-O (control O-letter, or some other key assigned by FS_SET ("debug")) or via the ALTD() function.

When the application is compiled with the "-d" switch (see LNG.1.3), it stops automatically in the debugger at the first executable statement.

Note, that you can press ctrl-O (^O) key whenever you wish (not only in a "wait state"), but the debugger itself will be entered after the current FlagShip command or function completes. That means it will become active when the currently executed FlagShip command or function is finished (for example in getsys.prg for READ or after MENU TO, ACHOICE etc. is completed).

Examine/set value:

If you want to examine any public or private value, just enter the variable name. To reach the LOCAL variables in the debugger, precede the variable name by the pseudo-alias `_L->` (underscore L), e.g. _L->MYLOCAL := 20. Similarly, use the pseudo-alias `_S->` for STATIC variables and `_P->` for local formal parameters. Note: LOCALs, STATICs and local formal parameters are visible in the debugger only if the source was compiled with the `-d` option. To assign a value to any of the PRIVATE or PUBLIC (and visible LOCAL, STATIC) variables, use the `:=` operator followed by any expression. For example:

```
name                                    // displays: John Miller
pi := 3.141592653
name :="Peter Jones"
f_date  := {04.09.1990}
actyear := YEAR((DATE())
name                                    // displays: Peter Jones
newvar := NIL                           // creates new
autoPRIVATE
_L->mylocvar                            // displays: 15
SET (2, .T.)                            // SET FIXED ON
xxx := SET (3, 5)                       // SET DECIMALS TO 5
_S->mystatvar := pi + _L->mylocvar      // displays: 18.14159
SET (3, xxx)                            // reSET DECIMALS
```

Display all variables:

Three functions are available to display variables in the current or any previous UDF/UDP:

`_DISPLVAR()` or `_DISPLVAR`(<depth>) displays all local, static variables and formal parameters in the current or previous procedure or function. The <depth> is similar to PROCNAME(), zero (the default) represents the current UDF, 1 the caller etc.

`_DISPLPRIV()` or `_DISPLPRIV`(<depth>) displays all PRIVATE and autoPRIVATE variables and PARAMETERs visible in the current or previous procedures or functions. The <depth> is similar to PROCNAME(), zero (the default) represents the current UDF, 1 the caller etc. Note, that LOCAL or STATIC variables with the same name do not hide the visibility of dynamically scoped variables by using this function, as the compiled code does.

`_DISPLPUBL()` also displays all PUBLIC and PRIVATE variables declared in the main procedure. It is similar to invoking _DISPLPRIV(999), i.e. a larger number than the current callstack depth.

In addition to displaying the variables of the current or previous UFD/ UDP by using the _DISPLxxx() functions, you may store this information into an array of any name. Call the similarly named functions with the syntax:

```
<arrname> := __DISPLVAR  ([<depth>])
<arrname> := __DISPLPRIV ([<depth>])
<arrname> := __DISPLPUBL ()  or  __DISPLPRIV (999)
```

Execute a command, function or code block:

Any of the linked standard or user defined functions may be executed or used in expressions. To execute a command, enter the corresponding function name. Examples:

```
name := "Peter Smith"                        // new variable
name                                         // Peter Smith
DBUSEAREA(.T., ,"address")                   // .T.
SELECT()                                     // 6
address->name                                // John Miller
address->name := M->name
address->name                                // Peter Smith
myblock := {|| QOUT(name, city)}
DBEVAL(myblock, RECNO() <= 20)               // display 20 records
```

Step:

By entering S (in upper/lowercase) at the prompt line, program execution will continue to the next source line.

Set a breakpoint:

You may set a breakpoint at any of the executable statements and continue the execution until the breakpoint is reached. If the source statement consists of several lines, set the breakpoint at the line number where the statement starts. To determine the line number within the source, you may load your editor in any other virtual screen. To set a breakpoint, enter at the debug prompt:

```
--> BREAKPOINT(<udfname>, <line>)  // specify module name and line no.
--> B                              // continue with active breakpoints
```

You may set a breakpoint at each line of a function. This is helpful to single-step through a function without stepping through subsequent function calls. To set a breakpoint at each line of a function, enter:

```
--> BREAKPOINT(<udfname>, 0)       // specify a module  name and line 0
--> B                              // continue with active  breakpoints
                                   // to the next line of this function
```

You may use up to ten breakpoints simultaneously. If using more than one breakpoint, specify additionally the breakpoint number 1 to 10:

```
--> BREAKPOINT(<udfname>, <line>, <breakpointno>)
                                   // specify  additional breakpoint no
--> B                              // continue with  active breakpoints
```

List active breakpoints:

To list the active breakpoints, at the debug prompt enter:

```
--> BREAKVIEW()                   // the active  breakpoints are listed
```

Delete a breakpoint:

To Delete a breakpoint, at the debug prompt enter:

```
--> BREAKVIEW("d", <breakpointno>) // the active breakpoints are listed
```

Quit the debugger:

Once you enter the debugger you can specify any expression and have it evaluated. To exit the debugger, you should type Q<enter>. All break- points are then disabled and the program continues. To leave the de- bugger with activated breakpoints instead, type B<enter>.

Example of a debugging session,

after activating it with ALTD() or with ^O (ctrl-O):

```
    1: *** program test.prg
    2: PRIVATE a:= 1, b:= {||QOUT("Hello")}, cc:= "xxyy"
    3: dd := DATE()
    4: ALTD()
    5: WHILE LASTKEY() # 27
    6:    DO myproc with cc
    7:    WAIT
    8: ENDDO
    9:
   10: PROCEDURE myproc (par1)
   11: pp := par1
   12: par1 := pp + "-" + LTRIM(STR(a++))
   13: RETURN
```

The screen is cleared and you are in debugging mode:

```
Enter Expression or Q to quit or S for step to next line:

TEST/4     --> dd
12/31/1993
TEST/4     --> substr(cc,2)
xyy
TEST/4     --> cc := "abcd"
abcd
TEST/4     --> S
TEST/5     --> S
TEST/6     --> S
MYPROC/11  --> par1
UNDEFINED                                 (since LOCAL)
MYPROC/11  --> _P->par1                   (LOCAL parameter)
xxyy
MYPROC/11  --> S
MYPROC/12  --> pp
abcd
MYPROC/12  --> breakpoint("test",7)       (declare module name and line)
.T.
MYPROC/12  --> breakview()                (list breakpoints)
BP1: test(7)
MYPROC/12  --> breakpoint("test",8, 2)    (set 2nd breakpoint at line 8)
.T.
MYPROC/12  --> breakview()
BP1: test(7)                              (list breakpoints)
BP2: test(8)
MYPROC/12  --> breakview("d", 2)          (delete breakpoint 2)
BP1: test(7)
MYPROC/12  --> B
TEST/7     --> cc + " " + EVAL(b)
abcd-1 Hello
TEST/7     --> B                        (break line in "test")
TEST/7     --> cc
abc-1-2
TEST/7     --> m->B                     (display the variable named "B")
BLOCK
TEST/7     --> Q                        (the application continues)
```

Note: In GUI mode, there are extended debugger capabilities available, see previous chapter FSC.5.1.

5.3 Unix Debugger

Of course, you may also use standard Unix debuggers, such are adb, sdb, cv, gdb, ddd, xgdb, xldb, codeview, etc. to check the internal or extended C modules. When using a C source debugger, you may preferably use `-g` (or the `-Wc,-g`) compiler switch in FlagShip. It may be combined with the `-d` (debugger) switch as well.

Example using gdb (or ddd):

```
gdb a.out           # invoke C debugger
l myfile.c:532      # display source file myfile.c at line 532
br 535              # set breakpoint in line 535
r mypar1 mypar2     # start execution, passing two parameters to main
                    # when compiled with -d, FS debugger displays
bt                  # stopped in myfile.c line 535, check backtrace
p myvar             # display content of C variable myvar
s                   # step to next line
c                   # continue to next breakpoint or until quit
```

For more information about C debugger, see your Unix documentation or the man pages of ddd, gdb, adb etc.

5.4 Windows Debugger

Similarly, to Unix, also in MS-Windows you may debug your C modules using e.g. MS-VC++ debugger from VisualStudio. You need to compile your application using the -g -c compiler switches and add the produced (and your) .c files into the project folder.

6. Tools, Utilities

In the FlagShip package, additional tools, especially for the handling the DOS-UNIX file transfer, are included.

All the tools have additional parameters and switches. All of them are language configurable, using the optional switch -e (English) or -d (German). To get quick on-line help, invoke the utility without parameters.

6.1 FSload - loads sources from diskette

For the transfer from DOS to Unix you may load all the files from diskette or USB stick onto Unix using the script file <FlagShip_dir>/bin/**FSload**. It will copy all *.prg, *.c, *.h, *.frm, *.lbl, *.dbf and *.dbt files at once and convert the sources to Unix style using **dos2unix** or **FSadopt**:

```
$ FSload [-e | -d]  device

$ FSload -e                 on-line help, English
$ FSload -e a:              OS dependent, predef.drive (A:)
$ FSload -e /dev/fd096ds15   for 5" (1.2 MB) in 1.drive (A:)
$ FSload -e /dev/fd196ds15   for 5" (1.2 MB) in 2.drive (B:)
$ FSload -e /dev/fd0135ds18  for 3" (1.4 MB) in 1.drive (A:)
$ FSload -e /dev/fd0135ds18  for 3" (1.4 MB) in 2.drive (B:)
$ FSload -e /dev/fd048ds9    for 5" (360 KB) in 1.drive (A:)
$ FSload -e /dev/fd148ds18   for 3" (720 KB) in 2.drive (B:)
```

The program will communicate with you interactively, asking for the kind conversion of source files etc.

For more information on this transfer and about drives/devices, see section SYS.

6.2 dos2unix - converts sources to Unix

The script file <FlagShip_dir>/bin/dos2unix (spell dos-to-unix) translates the DOS ASCII files (such as source code) into Unix style. It translates the CR/LF end-of-line marks into LF only, to make it comfortable when using it with a text editor like "vi" or "emacs". As opposed to FSadopt (see FSC.6.3), no translation of 8-bit chars to 7-bit will be performed.

```
   $ dos2unix [-e | -d] [-a]  file  [file ...]
   $ dos2unix -e                                  help, English
   $ dos2unix -e prg1.prg myudf2.prg              specified source files
   $ dos2unix -a *.c *.prg                        all given source files,
                                                  do not prompt
```

You may use wildcards (* or ?) within the file name or extension. If you enter file name <file> without the extension and if this file is not found, dos2unix looks for a file <file.prg> to convert it.

dos2unix checks the file to see if it contains programs or data (only ASCII files should be converted) to avoid destroying databases on incorrect entry. Sometimes, if the file contains graphic or 8-bit chars, you will be asked to confirm the translation. Using the -a (auto) switch, no prompt or confirming is necessary.

On success, the new file receives the name of the old one; the contents are overwritten. See also FSC.6.3 (FSadopt) and FSC.6.4 (unix2dos).

Note: The FlagShip compiler also accepts DOS source files, without any conversion. On some systems, this file may be named **d2u**, see RELease notes.

6.3 files2lower - converts files to lowercase

The script file <FlagShip_dir>/bin/files2lower (available in the Linux distribution of FlagShip) translates the given or all file names in the current directory to the common Unix/Linux lowercase. This is helpful e.g. to proper handle file names copied from DOS/Windows, since Linux names are case sensitive, while DOS/Windows are not.

Example:

```
cd /myapplic/source
cp /media/cdrom/myapplic/* .
/usr/local/FlagShip8/bin/files2lower *.PRG *.DBF
chmod 666 *.prg *.dbf
ls -la
```

You also may copy or link the script to e.g. /usr/local/bin (or /usr/bin) to avoid giving the full path every time:

```
ln -s /usr/local/FlagShip8/bin/files2lower /usr/local/bin/files2lower
```

6.4 FSadopt - converts sources to 7bit

If you need to convert DOS ASCII files (such as source programs) into Unix 7-bit character set, use the script <FlagShip_dir>/bin/**FSadopt**. It translates the 8-bit PC character set of the source file to the most similar 7-bit character (e.g. Ä to A, ö to o, graphics to "-", "|" and "+" and special chars to "." or "?"). The CR/LF to LF conversion will also be done. To convert only CR/LF, while keeping the PC-8 character set of the file, use the similar **dos2unix** (FSC.6.2) instead.

```
   $ FSadopt [-e | -d] [-a]   file  [file ...]
   $ FSadopt -e                                  help, English
   $ FSadopt -e prg1.prg myudf2.prg              specified source files
   $ FSadopt -d -a *.c *.prg                     all sources, German, do not
                                                 prompt
```

You may use wildcards (* or ?) within the file name or extension. If you enter file name <file> without the extension and if this file is not found, FSadopt looks for a file <file.prg> to adopt it.

FSadopt checks the file to see if it contains program or data (only ASCII file should be adopted) to avoid destroying databases on incorrect entry. Sometimes, if the file contains graphic or 8-bit chars, you will be asked to confirm the translation. Using the -a (auto) switch, no prompt or confirmation is necessary.

On success, the new file receives the name of the old one. The contents are overwritten. See also chapter FSC.6.2 (dos2unix) and FSC.6.4 (unix2dos).

6.5 unix2dos - converts sources to DOS

With the script file <FlagShip_dir>/bin/**unix2dos** (spell unix-to-dos) you may convert Unix sources to DOS style. The <LF> for end-of-line will be converted to CR/LF and a ^Z (end-of-file) will be added.

```
$ unix2dos [-e | -d] [-a]  file  [file ...]
$ unix2dos -e                      help, English
$ unix2dos -a *.c *.prg            all source files, do not prompt
```

You may use wildcards (* or ?) within the file name or extension. If you enter file name <file> without the extension and if this file is not found, unix2dos looks for a file <file.prg> to convert it.

The program checks the file to see if it contains program or data (only ASCII file should be converted) to avoid destroying databases on incorrect entry. Sometimes, if the file contains graphic or 8-bit chars, you will be asked to confirm the translation. Using the -a (auto) switch, no prompt or confirmation is necessary.

On success, the old file <filename> will be renamed, the new one receives the name of the old one. If you no longer require the original files, simply delete them with

```
$ rm \#*
```

See also chapter FSC.6.2 (dos2unix) and FSC.6.3 (FSadopt). Note, on some systems, this file is named **u2d**, see RELease notes.

6.6 fscheck - checks the environment

Apply mainly for **Terminal i/o**.

Before starting with your application, or when changing the terminal description, you should check the current environment settings (stty, TERM, ttymap etc.) according to chapter FSC.3.3, FSC.3.4 and section REL. To make your job more comfortable, you may use the semi-automatic checking program "fscheck.prg" stored in <FlagShip_dir>/examples. Copy it into your working directory, compile and run it. It will check and display the input and output mappings, the default settings and so on:

```
$ cd /usr/home                    (or other user directory)
$ cp <FlagShip_dir>/fscheck.prg .
$ FlagShip fscheck.prg            (see section FSC)
$ TERM=FSansi                     (see release notes, section REL)
$ export TERM
$ a.out     -or-    ./a.out
```

Now, you may perform a full check, including terminal, mapping, screen handling, colors, special keys etc. - or a partial check of the step selected. The program gives you context-sensitive help and suggestions for configuring your system properly.

6.7 newfscons, newfswin, newfsterm

Note: this chapter apply for Terminal i/o mode for Unix/Linux only

Before the compiled application is invoked, a proper environment (especially the TERM variable) should be set according to the RELease Notes, i.e. for an enhanced support of colors, FN keys etc.

For your convenience, we have added three (user modifiable) scripts to start your executable from various terminal environments:

a. From the system console:

```
$ newfscons <executable> [parameters]
```

-or-

```
$ newfscons
$ <executable> [parameters]
$ ... other shell commands
$ exit
```

b. From the X/window (xterm):

```
$ newfswin <executable> [parameters]
```

-or-

```
$ newfswin
$ <executable> [parameters]
$ ... other shell commands
$ exit
```

c. From a remote terminal (or terminal emulator):

```
$ newfsterm <executable> [parameters]
```

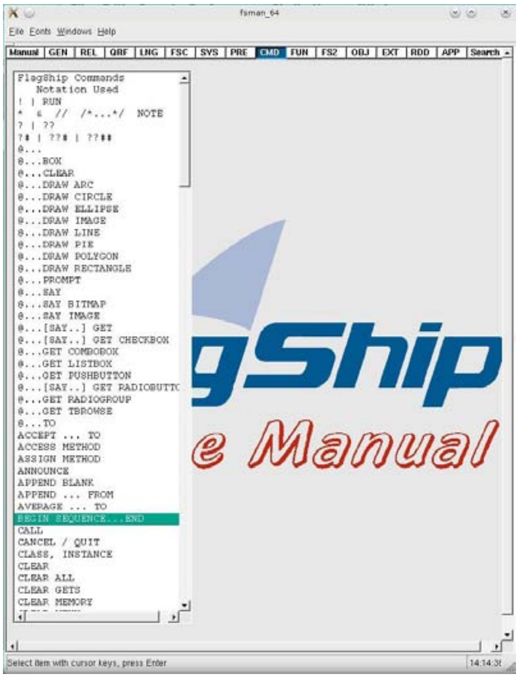
-or-

```
$ newfsterm
$ <executable> [parameters]
$ ... other shell commands
$ exit
```

Please refer to section REL (release notes) for a detailed, system specific description.

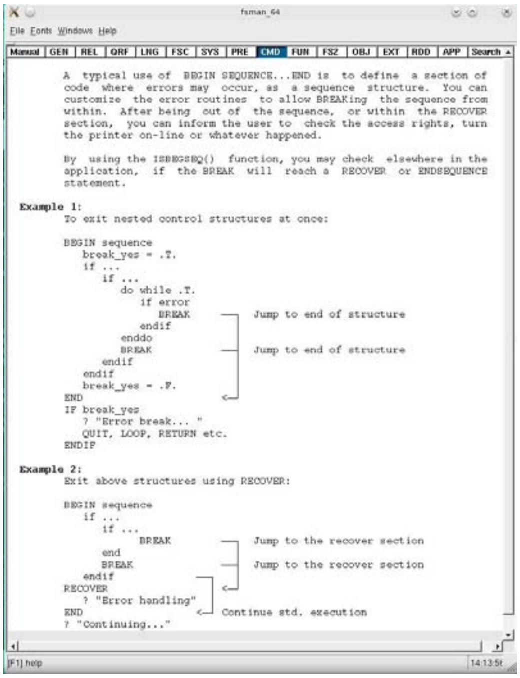
6.8 fsman - the FlagShip on-line manual

In MS-Windows, the on-line manual is an icon named "FlagShip8 Manual", generated by setup on your desktop, referring to <FlagShip_dir>\bin\fsman*.exe. For special purposes, you may use fsman_t.exe for processing the manual in terminal i/o mode.


fsman_64
File Fonts Windows Help
Manual GEN REL QRF LNG FSC SYS PRE CMD FUN FS2 OBJ EXT RDD APP Search
gShip
e Manual

```
FlagShip Commands
  Notation Used
! | RUN
*  &  //  /*...*/  NOTE
? | ??
?# | ??# | ??##
@...
@...BOX
@...CLEAR
@...DRAW ARC
@...DRAW CIRCLE
@...DRAW ELLIPSE
@...DRAW IMAGE
@...DRAW LINE
@...DRAW PIE
@...DRAW POLYGON
@...DRAW RECTANGLE
@...PROMPT
@...SAY
@...SAY BITMAP
@...SAY IMAGE
@...[SAY..] GET
@...[SAY..] GET CHECKBOX
@...GET COMBOBOX
@...GET LISTBOX
@...GET PUSHBUTTON
@...[SAY..] GET RADIOBUTTC
@...GET RADIOGROUP
@...GET TBROWSE
@...TO
ACCEPT ... TO
ACCESS METHOD
ASSIGN METHOD
ANNOUNCE
APPEND BLANK
APPEND ... FROM
AVERAGE ... TO
BEGIN SEQUENCE...END
CALL
CANCEL / QUIT
CLASS, INSTANCE
CLEAR
CLEAR ALL
CLEAR GETS
CLEAR MEMORY
```

Select item with cursor keys, press Enter   14:14:3t


fsman_64
File Fonts Windows Help
Manual GEN REL QRF LNG FSC SYS PRE CMD FUN FS2 OBJ EXT RDD APP Search

```
        A  typical use of  BEGIN SEQUENCE...END is  to define  a section of
        code  where  errors may  occur, as  a sequence  structure.  You can
        customize  the error routines  to allow BREAKing  the sequence from
        within.  After being  out of  the sequence,  or within  the RECOVER
        section,  you can inform the user to  check the access rights, turn
        the printer on-line or whatever happened.

        By  using the ISBEGSEQ()  function, you may check  elsewhere in the
        application,  if  the BREAK  will  reach a  RECOVER  or ENDSEQUENCE
        statement.

Example 1:
        To exit nested control structures at once:

        BEGIN sequence
           break_yes = .T.
           if ...
              if ...
                 do while .T.
                    if error
                       BREAK       ──┐   Jump to end of structure
                    endif            │
                 enddo               │
                 BREAK             ──┤   Jump to end of structure
              endif                  │
           endif                     │
           break_yes = .F.           │
        END                        <─┘
        IF break_yes
           ? "Error break... "
           QUIT, LOOP, RETURN etc.
        ENDIF

Example 2:
        Exit above structures using RECOVER:

        BEGIN sequence
           if ...
              if ...
                    BREAK          ──┐   Jump to the recover section
              end                    │
              BREAK                ──┤   Jump to the recover section
           endif            ──┐      │
        RECOVER               │    <─┘
           ? "Error handling" │
        END                 <─┘ Continue std. execution
        ? "Continuing..."
```

[F1] help   14:13:5t

In the Linux distribution, the on-line manual named `fsman` is available as `fsman_32` and `fsman_64` (with a link to fsman corresponding to the current OS architecture), installed into the <FlagShip_dir>/bin directory, with a symbolic link to /usr/bin. For special purposes, you may use `fsman_t` (link to `fsman_t_32` or `fsman_t_64`) for processing the manual in terminal i/o mode, see below.

The on-line manual is divided in sections (displayed at the top of pages) and includes all the descriptions from the printed manual. Since a usual, page oriented "index" is not possible here, the on-line manual offers you several hypertext-like search features. The on-line FlagShip manual consists of

- the executable named fsman[.exe],
- database FSman.dbf and FSman.dbv,
- database fsmanusr.dbf containing user configurations (created/updated by fsman)
- index files (created when first invoked, or if not available)
- release notes stored in the relnotes.asc file,
- description how to use it in the fsman.doc file

To invoke the on-line manual in Windows, simply click on the icon on your desktop, or manually invoke C:\FlagShip8\bin\fsman.exe from CMD prompt.

To invoke the on-line manual in Linux, enter

```
   $ fsman              # in GUI mode
   $ fsman_t            # in Terminal i/o mode when TERM is properly set,
or
   $ newfscons fsman_t  # (Terminal i/o) to set TERM automatically, see 6.7
   $ newfswin  fsman_t  # (Terminal i/o) to set TERM automatically, see 6.7
   $ newfsterm fsman_t  # (Terminal i/o) to set TERM automatically, see 6.7
```

If the databases are available in other than the default directory, you may store the directory name in the environment variable FSMAN

```
   $ export FSMAN=/var/flagship/man
   $ fsman
```

or supply the directory name as a parameter, e.g.

```
   $ fsman /var/flagship/man                # or
   $ newfscons fsman_t /var/flagship/man    # etc.
```

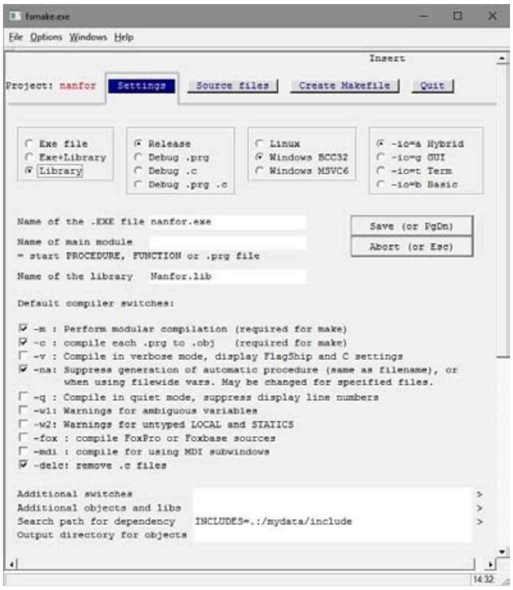
For your convenience, a "man" page named "fsman" is also installed, so you may retrieve the info about the fsman usage by

```
   $ man fsman
```

Performance hint: by disabling the scroll bar (menu Manual, Setup), you may increase the output speed significantly, especially on terminals with a slow transfer rate (in Terminal i/o mode only).

6.9 fsmake - creates Makefile

In the <FlagShip_dir>/tools is an IDE named 'fsmake' available, which semi- automatically builds your current or future projects (or library .LIB or .a) by using standard make utility. This fsmake tool records and manages source files of your specific project, and creates corresponding Makefile for the used operating system. You will then build the application (or rebuild only the changed sources) just by invoking 'make' (or 'nmake' for MS-VC) in your working / project directory. The short 'make' description is available in section FSC.2.



Project: nanfor

Exe file [✓ Library] Exe+Library

[✓ Release] Debug .prg Debug .c Debug .prg .c

Linux [✓ Windows BCC32] Windows MSVC6

[✓ -io=a Hybrid] -io=g GUI -io=t Term -io=b Basic

Name of the .EXE file nanfor.exe

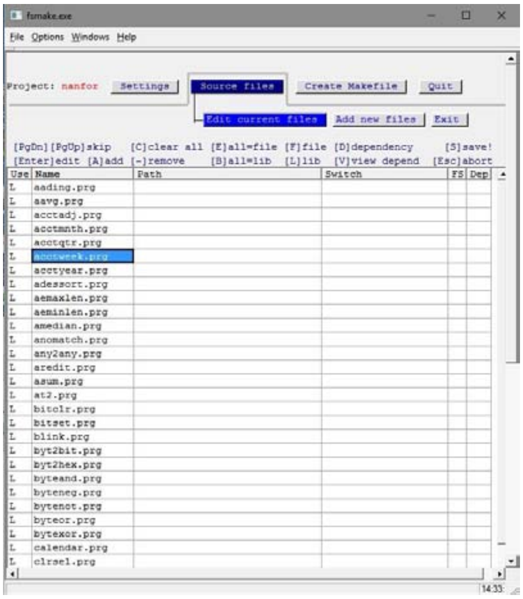
Name of main module
= start PROCEDURE, FUNCTION or .prg file

Name of the library Nanfor.lib

Default compiler switches:

- [✓] -m : Perform modular compilation (required for make)
- [✓] -c : compile each .prg to .obj (required for make)
- [ ] -v : Compile in verbose mode, display FlagShip and C settings
- [✓] -na: Suppress generation of automatic procedure (same as filename), or when using filewide vars. May be changed for specified files.
- [ ] -q : Compile in quiet mode, suppress display line numbers
- [ ] -w1: Warnings for ambiguous variables
- [ ] -w2: Warnings for untyped LOCAL and STATICS
- [ ] -fox : compile FoxPro or Foxbase sources
- [ ] -mdi : compile for using MDI subwindows
- [✓] -delc: remove .c files

Additional switches
Additional objects and libs
Search path for dependency INCLUDES=.:/mydata/include
Output directory for objects



Project: nanfor

```
[PgDn][PgUp]skip   [C]clear all [E]all=file [F]file [D]dependency     [S]save!
[Enter]edit [A]add [-]remove    [B]all=lib  [L]lib  [V]view depend [Esc]abort
```

Use	Name	Path	Switch	FS	Dep
L	aading.prg				
L	aavg.prg				
L	acctadj.prg				
L	acctmnth.prg				
L	acctqtr.prg				
L	acctweek.prg				
L	acctyear.prg				
L	adessort.prg				
L	aemaxlen.prg				
L	aeminlen.prg				
L	amedian.prg				
L	anomatch.prg				
L	any2any.prg				
L	aredit.prg				
L	asum.prg				
L	at2.prg				
L	bitclr.prg				
L	bitset.prg				
L	blink.prg				
L	byt2bit.prg				
L	byt2hex.prg				
L	byteand.prg				
L	byteneg.prg				
L	bytenot.prg				
L	byteor.prg				
L	bytexor.prg				
L	calendar.prg				
L	clrsel.prg				

1. Create the fsmake executable

a. on Linux, execute

```
$ cd /usr/local/FlagShip8/tools
$ FlagShip -delc fsmake.prg -o fsmake
$ ln -s `pwd`/fsmake /usr/local/bin/fsmake   # or: sudo ln -s ...
```

b. in MS-Windows, execute

```
C:> cd %FLAGSHIP_DIR%\tools
C:> FlagShip -delc fsmake.prg
C:> copy fsmake.exe %FLAGSHIP_DIR%\bin\fsmake.exe
```

2. Usage

The usage is simple and almost intuitive. In GUI mode, a tool-tip informs you additionally about options and gives you help for most entries. You may navigate by cursor keys or by mouse in GUI mode.

a. In your working/project directory, invoke

```
     fsmake
or:  fsmake my_project_name
```

b. Enter the project name, if not given during invocation of fsmake (a). The project name should not contain special or national characters, nor spaces. For multiple projects, you may press F2 to list available projects in this working directory. If not available yet, 'fsmake' creates database named fsmake_<projectname>.dbf and .dbt with corresponding .idx file.

c. Click tab "Settings" and check default settings. You need at least to enter

   - name of the executable, if differs from your project name, and
   - the start procedure (Main module name) of your application.

   The common choice is plain "Executable". If your project contains many files, you may select "Exe + Library" to put the most objects into library (selectable in step [d] by "L"; selecting current files in work as "F" may speed-up rebuild process). This "Exe + Library" option is required in MS-Windows for large projects with hundreds source files, since it linker buffer is only of limited size (8 KB in Windows, at least 250KB in Linux). You may change these settings at any time later. Click the "Save" button or PgDn key to save your changes

d. Click tab "Source Files" to manage your project sources. If there are not source files recorded yet, you will be navigated to section "Add new files". Usually, you will need at least the ".prg" checkbox activated. If (part of) your sources are in the current directory, leave the field "Source files directory" empty and press Enter or PgDn to display and select or de-select sources of your project. Press "S" key to save this selection. To add sources from another directory, select "Add new files" anew and enter the corresponding directory. By "Edit current files" you may display and manage files of this project. If required, add special switches (e.g. -na for selected source file). Press "S" to save changes or Esc to leave unchanged. You may leave the "Source Files" section by "Exit".

e. By selecting "Create Makefile", the 'fsmake' utility creates corresponding template for standard make (or nmake) utility. The created template (usually named 'Makefile') is an ASCII file, it syntax is described in FlagShip manual (fsman) section FSC.2

f. Exit 'fsmake' by selecting "Quit".

g. Invoke "make" (or "nmake" with MS-VC) to build (or rebuild changed sources of) your application. To rebuild all from scratch, invoke "[n]make clean" and then "[n]make". This is also required when you significantly change "Settings" (see [c] above).

Since all the data are saved in a database, you may interrupt the process and continue at any time later. As your project grows, you simply add new sources by "Source Files -> Add new files" and confirm it by "S"(save) in "Edit current files", let create Makefile and invoke make.

6.10 fsi - small interpreter

'**fsi**' is a simple interpreter executing FlagShip (xBase) based statements. It accepts standard commands and functions available in the FS library, as well as assign and access of given variables.

The 'fsi' is not intended to replace the FlagShip compiler and the compiled executable. Instead, it allows you a quick on-line checking the intended syntax in your application, maintain databases etc. in a similar way as for example the dBase, Foxbase, FoxPro or similar interpreters do.

The given syntax will be tested by the FlagShip Preprocessor and translated according to the *<FlagShip_dir>/include/std.fh* file. Since the given commands are not compiled to executable here, the given statements will be macro-evaluated (some says macro-compiled) by the FlagShip run time system instead. Hence, installed FlagShip is required when 'fsi' is running.

The source of 'fsi' is available in *<FlagShip_dir>/tools/fsi* directory. Simply compile by the attached Makefile

```
  make   (or 'nmake' with MS-VC)
```

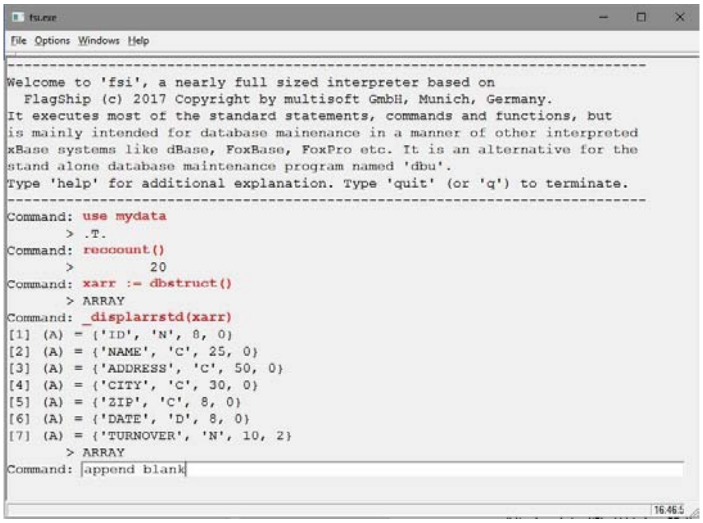
or manually by

```
  FlagShip -na -m -Mfsi fsi.prg [-io=t] -o fsi[.exe]
```

and execute by

```
  fsi   -or-  ./fsi  -or in Linux-  newfswin ./fsi
```

You may then copy the fsi executable to any directory of your choice, e.g. to *<FlagShip_dir>/bi*n or other directory accessible via PATH, or in your working directory.



```
--------------------------------------------------------------------------------
Welcome to 'fsi', a nearly full sized interpreter based on
  FlagShip (c) 2017 Copyright by multisoft GmbH, Munich, Germany.
It executes most of the standard statements, commands and functions, but
is mainly intended for database mainenance in a manner of other interpreted
xBase systems like dBase, FoxBase, FoxPro etc. It is an alternative for the
stand alone database maintenance program named 'dbu'.
Type 'help' for additional explanation. Type 'quit' (or 'q') to terminate.
--------------------------------------------------------------------------------
Command: use mydata
       > .T.
Command: reccount()
       >         20
Command: xarr := dbstruct()
       > ARRAY
Command: _displarrstd(xarr)
[1] (A) = {'ID', 'N', 8, 0}
[2] (A) = {'NAME', 'C', 25, 0}
[3] (A) = {'ADDRESS', 'C', 50, 0}
[4] (A) = {'CITY', 'C', 30, 0}
[5] (A) = {'ZIP', 'C', 8, 0}
[6] (A) = {'DATE', 'D', 8, 0}
[7] (A) = {'TURNOVER', 'N', 10, 2}
       > ARRAY
Command: append blank
```

There is an extensive on-line help available, type HELP when 'fsi' is running. Here an example, where data behind and within the "Command:" line are your entries, data below this line are preprocessed commands and results:

```
fsi (or "newfswin ./fsi" for Linux)
                         -------- (short info/help) -----------
Command: use mydbf
  prepr: dbUseArea(.f.,NIL,'tt',) ;_IF Used() .and. Set(87) ;
            DbGoTop() ;endif
   eval: (dbUseArea(.f.,NIL,'tt') , iif(Used() .and. Set(87), ;
            DbGoTop(), NIL))
       > NIL
Command: used()
       > .T.
Command: help
                         ------- (extensive help) -------------
Command: reccount()
       >       2896
Command: myvar := "hello"
       > hello
Command: _displarrstd(dbstruct())
         [1]  (A) = {'PARTNUM', 'C', 12, 0}
         [2]  (A) = {'DESC', 'C', 30, 0}
         [3]  (A) = {'FRAN', 'N', 9, 3}
         [4]  (A) = {'QNTY', 'N', 3, 0}
         [5]  (A) = {'EXT', 'N', 9, 2}
         [6]  (A) = {'MODEL', 'C', 6, 0}
         [7]  (A) = {'QNTY2', 'C', 3, 0}
         [8]  (A) = {'ORDER', 'C', 8, 0}
         [9]  (A) = {'DDATE', 'C', 8, 0}
         [10] (A) = {'ACCNO', 'C', 12, 0}
         [11] (A) = {'VERS', 'C', 13, 0}
         [12] (A) = {'SERIAL', 'C', 10, 0}
         [13] (A) = {'ACT_TAX', 'N', 9, 3}
         [14] (A) = {'ORDER2', 'C', 8, 0}
       > ARRAY
Command: append blank
 transl: dbAppend()
       > .T.
Command: recno()
       >       2897
Command: replace partnum with "abc def 123", qnty with 20,
         desc with myvar
  prepr: _FIELD->partnum :="abc def 123" ;(_FIELD->qnty :=20) ;
         (_FIELD->desc :=myvar)
       > hello
Command: partnum
       > abc def 123
Command: myvar
       > hello
Command: use
 transl: dbCloseArea()
       > .T.
Command: quit
```

As said above, valid FlagShip license is required also for the execution. FlagShip needs to be accessible via FlagShip_dir() function. When you get error saying *"cannot locate FlagShip compiler..."*, set the environment variable FSDEBUG_COMPILER to point to the main FlagShip directory, e.g.

```
  Windows: set FSDEBUG_COMPILER=C:\FLAGSHIP8
  Linux:   export FSDEBUG_COMPILER=/usr/local/FlagShip8
```

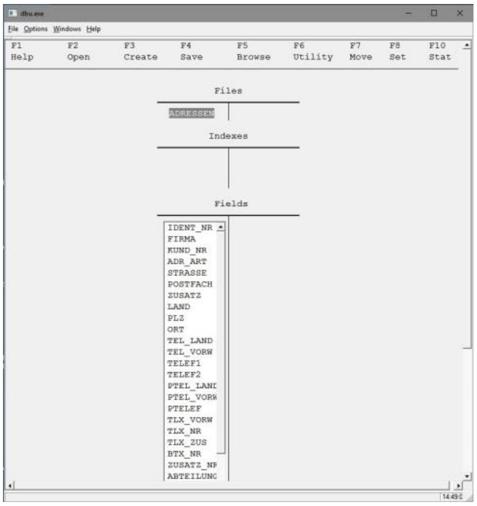
before executing fsi, see details in the source of fsi.prg and read the descriptive error message (if any displays).

6.11 dbu, calendar, creadb and other utilities

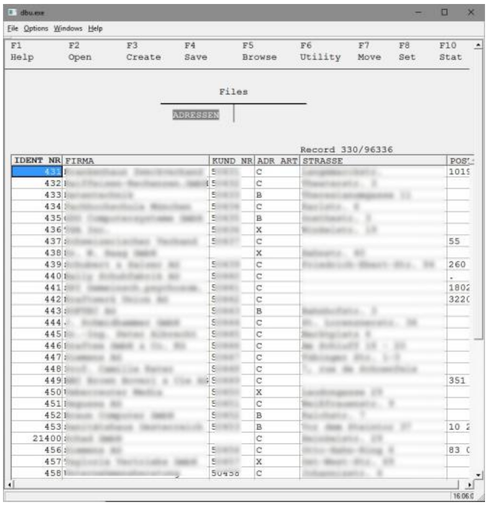
There are several examples and utilities, available in source code in the <FlagShip_dir>/system, <FlagShip_dir>/examples and <FlagShip_dir>/tools directory.

For example, **creadb.prg** allows you to create new databases according to an ASCII file description.

The **dbu** utility is well known from Clipper, and is available in the *<FlagShip_dir>/tools/dbu* directory. Simply compile there by “make” (or “nmake”)



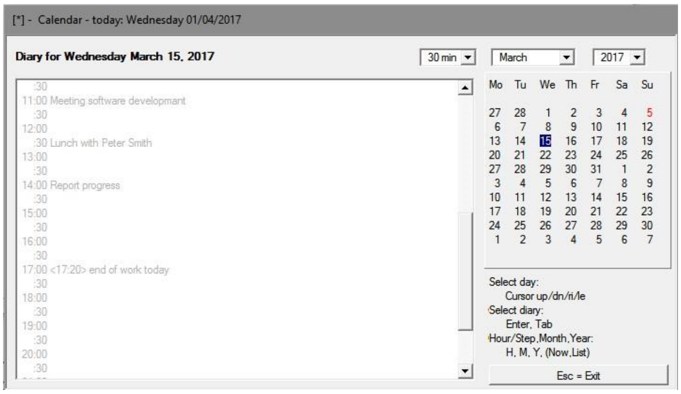
```
IDENT_NR
FIRMA
KUND_NR
ADR_ART
STRASSE
POSTFACH
ZUSATZ
LAND
PLZ
ORT
TEL_LAND
TEL_VORW
TELEF1
TELEF2
PTEL_LANI
PTEL_VORW
PTELEF
TLX_VORW
TLX_NR
TLX_ZUS
BTX_NR
ZUSATZ_NF
ABTEILUNG
```



Record 330/96336

IDENT NR	FIRMA	KUND NR	ADR ART	STRASSE	POS
431	[redacted]	[redacted]	C	[redacted]	1019
432	[redacted]	[redacted]	C	[redacted]	
433	[redacted]	[redacted]	B	[redacted]	
434	[redacted]	[redacted]	C	[redacted]	
435	[redacted]	[redacted]	B	[redacted]	
436	[redacted]	[redacted]	X	[redacted]	
437	[redacted]	[redacted]	C		55
438	[redacted]		X	[redacted]	
439	[redacted]	[redacted]	C	[redacted]	260
440	[redacted]	[redacted]	C		-
441	[redacted]	[redacted]	C		1802
442	[redacted]	[redacted]	C		3220
443	[redacted]	[redacted]	B	[redacted]	
444	[redacted]	[redacted]	C	[redacted]	
445	[redacted]	[redacted]	C	[redacted]	
446	[redacted]	[redacted]	C	[redacted]	
447	[redacted]	[redacted]	C	[redacted]	
448	[redacted]	[redacted]	C	[redacted]	
449	[redacted]	[redacted]	C		351
450	[redacted]	[redacted]	X	[redacted]	
451	[redacted]	[redacted]	C	[redacted]	
452	[redacted]	[redacted]	B	[redacted]	
453	[redacted]	[redacted]	B	[redacted]	10 2
21400	[redacted]		C	[redacted]	
456	[redacted]	[redacted]	C	[redacted]	83 0
457	[redacted]	[redacted]	X	[redacted]	
458	[redacted]	50458	C	[redacted]	

The **calendar.prg** (requires FS2 Toolbox) contains function CalendarDiary() for FlagShip, similar to Outlook's calendar. The tcalendar.prg is the test program for CalendarDiary(). See header of tcalendar.prg for details.



Diary for Wednesday March 15, 2017

```
   :30
11:00 Meeting software developmant
   :30
12:00
   :30 Lunch with Peter Smith
13:00
   :30
14:00 Report progress
   :30
15:00
   :30
16:00
   :30
17:00 <17:20> end of work today
   :30
18:00
   :30
19:00
   :30
20:00
   :30
```

Mo	Tu	We	Th	Fr	Sa	Su
27	28	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

Select day:
   Cursor up/dn/ri/le
Select diary:
   Enter, Tab
Hour/Step,Month,Year:
   H, M, Y, (Now,List)

In the *<FlagShip_dir>/examples* directory, there is Makefile available, which compiles and executes all examples there in GUI and Terminal i/o mode by simply "make" (or "nmake") invocation.

Additional utilities and programming solutions are given in the manual, e.g. sections LNG.10, CMD, FUN, OBJ, EXT, RDD. See the reference in the section APP of the on-line manual. For your convenience, it is not necessary to retype these examples. Instead, you enter the required chapter of the on-line manual (fsman, see 6.8) and simply extract it into an editable ASCII file.

Index

Notes

MULTISOFT

multisoft Datentechnik
Schönaustr. 7
D-84036 Landshut

http://www.fship.com
sales@multisoft.de
support@flagship.de