FlagShip

Object Oriented Database Development System

Cross-Compatible to Unix, Linux and MS-Windows

MULTISOFT

Release 8.1 **Section PRE**

The whole FlagShip 8 manual consist of following sections:

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

multisoft Datentechnik, Germany

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

Section PRE

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

PRE: FlagShip Preprocessor Directives

1. Overview

One of the FlagShip compiler tasks is the preprocessor. For details refer to section FSC.1.1. The preprocessor performs syntactical source checking and the translation and/or execution of special preprocessor directives. The output of this task is written into files with a .bp extension.

The preprocessor directives are part of the FlagShip source program, which also include other statements, declarations and commands. These directives are instructions to the compiler/ preprocessor rather than statements controlling the program execution.

By using the directives, user-defined commands (UDC) may also be specified - or in extreme cases, the standard command meaning may be changed. In fact, most of the standard commands are automatically translated to FlagShip standard functions. This is done using the preprocessor directives specified in the automatically included "std.fh" file.

1.1 Notation

Preprocessor directives must appear on a separate line and always begin with a hash symbol #. Following the hash symbol, optional white space is allowed.

FlagShip preprocessor directives are also accepted if the line starts with white space (blanks or tabs).

With a semicolon ";" at the end of line, you may continue a preprocessor directive into the next source line.

Inline comments using `//`, `&&` and `/*...*/` are supported in FlagShip directives. During the preprocessor phase, such comments will first be removed and they therefore do not influence the final translation.

The FlagShip directives are comparable to directives or pragmas of the C compiler. In C, the hash symbol must appear at the first column of the source line in order to be accepted.

1.2 Case-Sensitivity

In FlagShip, preprocessor directives and their arguments are **not** case-sensitive, except for the identifier of the `#define`, `#ifdef` and `#ifndef` directives, which are.

Using the Open C API (see section EXT), all standard C directives and pragmas are also supported within the C program block. These are always case-sensitive.

1.3 Immediately Executed Directives

These directives instruct FlagShip to perform the required action, e.g. to insert another file, to compile the program block only when a specified condition is met, etc.

Directive	Description
#include "<filename>"	Include a file into the current source
#ifdef <identifier>	Compile only when the identifier exists
#ifndef <identifier>	Compile when identifier does not exist
#else	Optional part of the #ifdef or #ifndef structure
#endif	End of the #ifdef or #ifndef structure
#comments	Transfer full-line comments into the .c file
#nocomments	No transfer of full-line comments into the .c file
#stdout [<message>]	Display a message during compilation
#error [<message>]	Display error message and terminate compilation
#debug_off	Temporarily disable debugger, if active
#debug_on	Enable debugger, unset #debug_off status
#Cinline	Start of the inline C source. Transfer all following lines directly into the .c file
#endCinline	End of the inline C source

The executable or conditional directive operates only in the current program line.

1.4 Translation Directives

The translation directives are used to modify standard program statements, re-define user-defined commands and to declare symbolic names instead of constants. When the preprocessor encounters such directives, it does not execute them immediately, but stores them on an internal stack in order to perform a translation of all subsequent program lines, when the directive rule applies.

Directive	Description
#define <identifier> [<constant>]	Defines a manifest constant
#undef <identifier>	Remove the #defined identifier
#command <pattern> => <result>	Translates a user-defined command starting with a pattern which may be abbreviated
#xcommand <pattern> => <result>	Translates a user-defined command starting with pattern. Does not allow abbreviation
#translate <pattern> => <result>	Translates a literal. The pattern may be abbreviated
#xtranslate <pattern> => <result>	Translates a literal, without abbreviation

These directives have a file-wide scope, i.e. they are valid from the current program line until the end of the current program file is reached.

1.5 Priority of Translation Directives

The preprocessor collects the translation directives on an internal stack (in first-in / last-out order) and checks all following program lines to determine if the translation directive applies.

The `#define` directive can also be specified when invoking the FlagShip compiler. The `-D` switch is then valid for the whole .prg file.

Translation directives can also influence other, specified directives on the stack. The directive given last has highest priority overriding a previous one.

Nesting of translation directives is allowed. This means that one directive may influence all previously defined directives. For example:

```
#command EFGH   <file> => myudf(<"file">)       /* Priority three */
#command ABCDEF <file> => EFGH <file>           /* Priority two   */
#define  XyZ ABCD                               /* Priority one   */

XyZ   myname                                    // myudf("myname")
AbcDE anyname                                   // myudf("anyname")
EFGH  other                                     // myudf("other")
XyZ   XyZ                                       // myudf("ABCD")
```

As the preprocessor encounters a translation directive, it checks previous definitions in order to perform substitutions. The order of precedence is: `#define`, `#[x]translate`, and `#[x]command`. When there is a match, the substitution is made to the resulting text and the entire line is reprocessed until there are no more matches for any of the translating definitions.

For this reason, you must avoid recurrence. The following directives would produce an infinite loop and would result in a compiler error:

```
#define Aaa BBB
#define BBB Aaa
x = BBB + 10                                    // compiler error
Aaa = 4                                         // compiler error
aaa = 4 + Aaaa                                  // o.k.
```

Because the translation directives are processed first-in / last-out, place the most general case first, followed then by the more specific ones. This assures that the appropriate rule will match the command specified in the program.

Since the `#include "std.fh"` file, which contains many `#command` and `#define` directives, is issued automatically by the FlagShip preprocessor (except when the compiler switch `- nI`<file> is used) at the program start, all subsequently defined `#command`, `#translate` or `#define` directives (e.g. in the .prg file or in other `#include` files) will override the default translation of the same pattern.

1.6 Examining the Results

To check the preprocessor translation, invoke the FlagShip compiler with the -a option and examine the .bp file produced with the same name.

1.7 Example of Use

```
*** file test.prg
/* #include "std.fh"           this file is included automatically */

#ifdef FlagShip                                   // compiling with FlagShip ?
   #stdout Compiling with FlagShip
   #include "inkey.fh"
#else
   #stdout Compiling with Clipper
   #include "inkey.ch"
#endif

#translate XYZ => abc
#COMMAND   FLUSH            => DBCOMMIT()      /* available in Fox */
#command   OPENDBF <*line*> => USE <line>    ;
                             ; IF NETERR() ;
                             ; QUIT        ;
                             ; ENDIF
#define    ESCAPE_KEY    27
#define    NONEXCLUSIVE  SHARED

#Command   SET FILES TO LOWERCASE ;
        => fs_set ("lower", .T.); fs_set ("pathl", .T.)

#Command   SET AUTOTRANSLATE [<how:ON,SET>][OFF][UNSET] ;
        => fs_SET ("lowerfile", <.how.>) ;
           fs_SET ("pathlower", <.how.>)

Set File to Lower                         // fs_set ("lower", .T.)
                                          // fs_set ("pathl", .T.)
SET AUTOtransl ON                         // fs_set ("lowerfile", .T.)
                                          // fs_set ("pathlower", .T.)

OpenDbf Myfile NONEXCLUSIVE NEW           // USE myfile SHARED NEW
                                          // IF NETERR ()
                                          //    QUIT
                                          // ENDIF

Flush                                     // DBCOMMIT()
xyz = Xyz ()                              // abc = abc ()
SET AUTOtransl OFF                        // fs_set ("lowerfile", .F.)
                                          // fs_set ("pathlower", .F.)

#ifdef FlagShip
   #Cinline
      chdir ("/tmp")                      /* execute C statement */
   #endCinline
   ? "changed to directory", CURDIR()
#else
   #error Cannot change directory in DOS
#endif
Close Data                                // close database
*** eof test.prg
```

1.8 Difference to Clipper

Both FlagShip's and Clipper's preprocessor work nearly equivalent, while FlagShip's does it more precisely:

1. The main difference between FS and Clipper is in the "translation" handling of the <match marker>s. In Clipper, a simply textual replacement is done, whereas in FlagShip the result is checked to see if it is actually **valid** expression.
2. The use of operators (such as "`:=`", "=", "`-=`", "`:`", "<" etc.) as a keyword (match marker) may result in unexpected results or will not be translated at all, because of (1). If so, replace the operator at the left side of the translation directive by a keyword (for example the "`:=`" operator by the "_IS_" keyword, "=" by "_EQ_" etc.). Your directives and sources then remain backward compatible to Clipper as well.
3. The optional list match [<var,...>] works slightly differently from Clipper, because of (1). As a general rule, you should specify markers following fix command parts first and put the optional (repeating) markers at the command end.
4. Note that non-optional repeating markers <var,...> (i.e. comma-separated markers) are resolved differently from the optional markers [<var,...>] where the comma is also optional.
5. When possible, don't translate command names to themselves `#command` but translate to another command (see e.g. the @..GET in std.fh) which simplifies the preprocessor's task.
6. If two or more optional matches e.g. [<var,...>] are used but not coupled with command keywords e.g. [COLOR <var,...>], the preprocessor is not always sure about how to match because of (1). You should then use e.g. <keyw1> <xx1> [<xx2>,...] <keyw2> <yy1> [<yy2>,...].
7. You should also use #command instead of `#xcommand` in repeated translations with the std.fh (such an additions to @..SAY etc), since the clauses are often abbreviated there (refer e.g. to std.fh approx. line 530).
8. You may check the translation of your directives in the <source>.bp file when compiling with "FlagShip -a <source>.prg".

#Cinline ... #endCinline

Syntax:

```
#Cinline /* case sensitive */
     <any valid C statements> /* case sensitive */
#endCinline /* case sensitive */
```

Purpose:

Delimits C code included in the .prg source.

Arguments:

none.

Description:

Along with the Extend C and Open C API System, FlagShip also supports programming C directly within the .prg file. This is a very comfortable way to invoke a function from the Unix or Windows library, speed up complicated calculation, etc.

The FlagShip preprocessor transfers all the lines between `#Cinline` and `#endCinline` directly into the resulting .c file. The C code must therefore comply with the standard C syntax (statements are terminated with a semicolon, only /*...*/ and //... comments are supported, a sequence of code is enclosed in curly brackets {...} etc.). Additionally, no other FlagShip directives apply to the C code between `#Cinline`...`#endCinline`, while C pragmas and directives may be used.

To access variables in the .prg part, use LOCAL...AS or STATIC...AS typed variables given in lowercase, or use macros and functions of the Open C System. For details, refer to the section EXT.4.

If local or external C variables are required, the whole C program block has to be enclosed in curly brackets {...}.

To be able to generate automatic PROCEDURE <filename> (see LNG.2.3) when compiled w/o -na switch, at least one FlagShip statement must precede the `#Cinline` directive. Best to place `#Cinline`..`#endCinline` within usual PROCEDURE or FUNCTION body. To declare C function(s), place `#Cinline`...`#endCinline` at the begin of .prg file and compile with -na switch.

For more information about inline C programming, refer to sections EXT.3 and LNG.8.

Example:

See examples in sections LNG.8, EXT.3, EXT.4 and CMD.CALL.

Compatibility:

Available in FlagShip only.

Related:

EXT.3, Open C System, CALL command

#command, #xcommand

Syntax:

```
#command <pattern> => [<result>]
#xcommand <pattern> => [<result>]
```

Purpose:

Specifies a user-defined command directive.

Arguments:

<**pattern**> is the input which the text should match. It defines the translated command (refer also to sections CMD and LNG) and must follow the command syntax rule. The <pattern> consists of literal symbols (keywords and clauses) and optional variables, enclosed in angle brackets, e.g. <var>, complying to the FlagShip variables naming convention. The keyword must be given. The entire entry is not case-sensitive.

=> This equals sign immediately followed by a greater than sign is a literal part of the syntax and separates the <pattern> from the <result>.

<**result**> is the text to be produced if a portion of the input text matches the <pattern>. It may include literal constants, clauses and result variables <var> enclosed in angle brackets. The whole <result> part of the syntax is optional. The resulting expression must comply with valid FlagShip syntax. If no <result> is specified, an empty line is produced.

Description:

The `#[x]command` directive provides a way of re-defining a user- defined command as any other command or function. You can use a command in place of an expression or function call to define the order of keywords, required arguments, combinations of arguments that must be specified together, and mutually exclusive arguments, at compile time rather than at runtime.

The `#command` directive is also used in the std.fh file to substitute standard commands by a function invocation, providing its arguments in the proper order.

The `#command` directive supports abbreviating the matching pattern (keyword and clauses) up to four characters, while `#xcommand` only translates patterns exactly as given. All other rules apply for both directives.

`#[x]command` is similar to `#[x]translate`, but it matches only if the input text is a complete statement, while `#[x]translate` also matches input text which is not a complete statement. In general, `#[x]command` is used for most user-defined commands, while `#[x]translate` is used only in special cases.

There are several subtle issues you need to recognize to properly specify a command definition. Many commands require more than one `#command` directive because of mutually exclusive clauses containing different keywords or arguments, for example the SET command in the std.fh file. This is also true when a result pattern contains different expressions, functions, or parameter structures for different clauses specified in the same command (e.g., the @...SAY command).

You may re-define standard [x]command's from std.fh or stdfoxpro.fh by your own (e.g. in the .prg or your own .fh file), the last definition is the most current (lifo order = last-in-first-out).

Pattern:

The <pattern> of the directive may consist of several parts, separated by at least one white space (blank or tab):

1. **Keyword**, corresponding to the translated command in the program source. (The command starts at the beginning of the line, leading blanks are ignored. For more information, refer to section LNG.2.4). Specify the full length of the keyword. `#command` will then also accept a program entry abbreviated with a maximum of four leading characters.

```
#command COMMIT  => DBcommitALL()
#command FLUSH   => DBcommit()
```

2. **Mandatory clauses** and literal symbols which **must** appear in the input text in order to activate the translation directive. The `#command` directive will also accept a program entry abbreviated with up to four leading characters of the clause. Examples:

```
#command CLEAR SCREEN      => CLS
#command CLEAR MEMORY      => __MClear()
#command DO WHILE <*line*> => WHILE <line>
```

3. **Optional clauses** and literal symbols which can appear in the input text. These optional clauses are enclosed in brackets [...], along with their arguments (variables) and additional clauses. A sequence of optional clauses allows their order to be interchanged in the matching text. The optional clauses may be nested. Examples:

```
#command GO [TO] <rec> => GOTO <rec>
#command SET PRINTER [TO <(fi)>] [<how:ADDIT,APPEND>] ;
         => SET (24, <(fi)>, <.how.>)
```

4. Mandatory or optional **variables**, specifying the command arguments. The variables, holding the input text, are enclosed in angle brackets <...>. There are five types of pattern variable notations:

<var>	Regular match variable
<var,...>	List match variable
<var:word list>	One-of-list match variable
<*var*>	Wildcard match variable
<(var)>	Extended expression variable

● **Regular match variable** <var> saves the next legal expression of the input text, e.g. the command argument. The matched text is represented by a simple variable (having scope and visibility for the current directive only). Examples:

```
#command MYCMD <arg> => myfun (<arg>, .T.)
#command GOTO  <rec> => DBGOTO (<rec>)
```

● **List match variable** <var,...> saves a comma-separated list of legal expressions. If there is no input text match, the specified variable does not contain anything.

The list match variable defines command clauses which have lists as arguments. Typically these are FIELD clauses or expression lists used by database commands. When there is a match for such a variable, the list is usually written to the resulting text using either the "simple" or the "conditional" stringify result variable. Lists are often also written as literal arrays by enclosing the result variable in curly {...} braces. Examples:

```
#command  ?? [<list,..>]   => QQOUT (<list>)
#xcommand MYLIST <fld,..>  => __dblist ({<fld>}, .T.)
```

● **One-of-list match variable** <var:text[,text]> matches the input only to one text in a comma-separated list. If the input text does not match at least one of the searched text patterns, the match fails and the variable contains nothing. It is often used with the "logified" result variable to write a .T. if there is a match, or .F. otherwise (when the match variable is defined as optional). There are two special signs which may be used in the list: the ampersand &, which will match any valid macro expression, and the hash #, which will match any valid identifier, such as variable or function names. Examples:

```
#command MYCMD <arg:ON,OFF>  => myfun ( <.arg.> )
#command SETTING <arg:ON,OFF,&>  => myset ( <(arg)> )
#command SET KEY <num> TO <proc:#>[([<dummy,...>])] ;
      => SetKey (<num>, {|p1,p2,p3| <proc>(p1,p2,p3)} )
```

● **Wildcard match variable** <*var*> matches any input text from the current position to the end of the statement. It is often used to stringify or ignore the rest of a command line. Examples:

```
#xcommand TEXT1 <*txt*>          => QOUT ( <(txt)> )
#command  SET PATH [TO] <*path*> => SET  (6, <(path)> )
#command  SHUSE <*arg*>          => USE <arg> SHARED
#command  ENDDO <*txt*>          => END
```

● **Extended Expression variable** <(var)> matches a regular or extended expression, including a filename, path specification etc., where the regular match variable may fail, e.g.

```
#command SET DEFAULT [TO] <(path)> => SET (7, <(path)>)
#command DELETE FILE <(file)>      => FErase( <(file)>)
```

Result:

The <result> portion of a translation directive is the text the preprocessor will produce if the <pattern> was found in the input text. The <result> syntax starts after the => symbol of the `#command` directive and may consist of several parts, separated by at least one white space (blank or tab):

1. Any **literal text**, that is written directly to the result text, including any valid keyword, clause or name of the FlagShip language. These words are written directly to the resulting text. Special characters (like `[`, `]`, < or >) of the text must be preceded with a backslash \ . Examples:

```
#command    SET TALK <*rest*> ;
                                => * \<unsupported\> SET TALK <rest>
#command    GOFIRST          => GOTO TOP
#xtranslate LASTGET          => ATAIL(GetList)
#xtranslate GET_NUM (<nr>)  => GetList \[<nr>\]
#command MYCMD [EVERY [TIMES] [<xx>]] => OTHER (<xx>)

Set Talk Off                 // * <unsupported> SET TALK Off
x = LastGet                  // x = ATAIL(GetList)
y = GET_NUM(2)               // y = GetList [2]
GOFIRST                      // GOTO TOP
GOFIR                        // GOTO TOP
GOFIRST anything             // GOFIRST anything  [no match]
MYCMD                        // OTHER ()
MYCMD EVERY                  // OTHER ()
MYCMD EVERY 20               // OTHER (20)
MYCMD EVER TIME anything     // OTHER (anything)
```

2. Mandatory and/or repeating result variables, matching the <pattern> variable names, enclosed in angle brackets <...>. There are six types of result variable notations:

<var>	Regular result variable
#<var>	Always-stringified result variable
<"var">	Simple stringified result variable
<(var)>	Conditional stringified result variable
<{var}>	Conditional blockified result variable
<.var.>	Logified result variable

● **Regular result variable** <var> writes the contents of the input pattern text, represented by <var>, to the resulting text as is. If no input text for <var> was found, the result of <var> is empty. Examples:

```
#command GOTO  <rec>   => DBGOTO (<rec>)
#command MYCMD [<arg>] => myfun  (<arg>, .T.)
GOTO 25                // DBGOTO (25)
GOTO anything          // DBGOTO (anything)
GOTO anything else     // GOTO anything else [no match]
MYCMD                  // myfun (, .T.)
MYCM  anything         // myfun (anything, .T.)
```

● **Always stringified result** variable #<var> stringifies the contents of the input <var> with `"..."`, `'...'` or `[...]` depending on the text contents and writes it to the resulting text. No match in the input text results with a null string `""`. If the input is a list `<var,...>`, the resulting string is the entire list enclosed in `""` symbols. It is generally used for commands where the arguments are specified as a literal value but the resulting text must always be written as a string, even if the argument is not specified.

```
#command MYCMD <arg,...>        => myfun ( #<arg> )
#command SET COLOR TO [<*arg*>] => SetColor ( #<arg> )
MYCMD xyz                 // myfun ( 'xyz' )
MYCMD abc, xyz            // myfun ( 'abc,xyz' )
SET COLOR TO              // SetColor ('')
Set Color To &var         // SetColor ("&var")
set colo  to W+/B,N/W     // SetColor ('W+/B,N/W')
set colo  to 'W+/B,N/W'   // SetColor ("'W+/B,N/W'")
```

● **Simple stringified result** variable <"var"> writes the contents of the input <var> to the resulting text. It is similar to the always stringified variable, but if the input is a list `<var,...>`, it stringifies each element of the list. This is generally used for commands, when results should be separately stringified, for example:

```
#command MYCMD <arg,...>  => myfun ( <"arg"> )
#command SET FILTER TO <expr> => ;
                    DBSETFILTER ( <{expr}>, <"expr">)

MYCMD abc                  // myfun ('abc' )
MYCMD abc, def, ghi        // myfun ('abc','xyz','ghi')
SET FILT TO zip > 1234
             // DBSETFILTER ( {|| zip>1234}, 'zip > 1234')
```

● **Conditional stringified result** variable <(var)> is similar to <"var"> but it stringifies text only if the input <var> text is not enclosed in quotes or parentheses or is not any form of macro. If no input text matched, it writes nothing to the resulting text. If the input is a list `<var,...>`, every element of the list is stringified by this conditional rule. It is usually used for commands which can be specified as a literal or a parenthesed expression, e.g. the standard commands USE, SET INDEX etc.

```
#command MYCMD <arg>    => myfun ( <(arg)>, .T. )
MYCMD abc               // myfun ('abc', .T. )
MYCMD "abc"             // myfun ('abc', .T. )
MYCMD (var + "xyz")     // myfun (var + 'xyz', .T. )
```

● **Conditional blockified result** variable <{var}> writes the matched input <var> text as a code block without any arguments to the resulting text, provided it is not already a code block. In the latter case, no additional blockifying is done. If no input text is matched, it writes nothing to the resulting text. If the input is a list `<var,...>`, every element of the list is blockified by this rule.

It is usually used in conjunction with the regular match variable <var> to create a code block from the <var> input text. Using code blocks instead of macros may speed up the application significantly. In the std.fh file, blockifying is often used for database commands and for the FOR/WHILE conditions. Look out for the scope and visibility of variables in code blocks, as described in section LNG.2.3.3.

```
#command MYCMD <arg>    => myfun ( <{arg}>, <"arg"> )
#command SET FILTER TO <expr> ;
      => DBSETFILTER ( <{expr}>, <"expr"> )

MYCMD abc          // myfun ({|| abcd}, 'abc')
MYCMD var == "xx"
                   // myfun ( {|| var=='xx'}, 'var=="xx"' )
SET FILTER TO TRIM(country) $ "USA,CND,D"
       // DBSETFILTER ( {|| TRIM(country) $ "USA,CND,D"}, ;
       //                'TRIM(country)$"USA,CND,D"')
```

● **Logified result** variable `<.var.>` writes .T. to the resulting text if the input <var> text is matched; otherwise nothing or .F. . It is usually used with the one-of-list variable to write TRUE to the resulting text if an optional clause is specified, or FALSE otherwise.

```
#command MYCMD [<arg:ON,SET>][OFF][UNSET] ;
      => myfun ( <.arg.> )
#command SET ALTERNATE [<what: ON>][OFF] [<how: NEW>] ;
      => SET (18, <.what.> , <.how.>)
MYCMD SET               // myfun (.T.)
MYCMD ON                // myfun (.T.)
MYCMD OFF               // myfun (.F.)
MYCMD                   // myfun (.F.)
SET ALTER On            // SET (18, .T., .F.)
Set Altern Off New      // SET (18, .F., .T.)
```

3. **Repeating result** clauses are enclosed in square brackets [...] and instruct the preprocessor to write the text of <var> in the resulting text as many times as it has matches in the input text for any result variables within the clause. If there is no matching input text, the repeating clause is not written to the result. Repeating clauses cannot be nested. They are often used for commands which support repeating clauses, e.g.

```
#command MYSAVE  <x1> TO <v1> [, <xN> TO <vN>] ;
      => REPLACE <v1> WITH <x1> [, <vN> WITH <xN>]
#command STORE <value> TO <var1> [, <varN>] ;
```

```
      => <var1> := [ <varN> := ] <value>

MYSAVE 25 TO idnum
               // REPLACE idnum WITH 25
MYSAVE "Smith" TO name, "Peter" TO first, 1 to num
               // REPLACE name WITH "Smith", ;
               //         first WITH "Peter", num WITH 1

STORE 1234 TO var1, var2, abcd
      // var1 := var2 := abcd := 1234
```

4. To create **multi-command** statements, separate each command with a semicolon (`;`) which will be written to the result as a literal. As a semicolon at the line ending is used to continue the statement into the following line, use two semicolons in this case.

```
#command XIF <cond>, <true_res>, <false_res> ;
      => IF <cond> ; <true_res> ;;
         ELSE ; <false_res> ; ENDIF
XIF a > 0, x := a, x := 0
             // IF a > 0 ; x := a ; ELSE ; x := 0 ; ENDIF
* which is equivalent to:
             // IF a > 0
             //    x := a
             // ELSE
             //    x := 0
             // ENDIF
```

Example:

See examples given above and study the std.fh file.

Compatibility:

Available in FS4 and C5 only. In Clipper 5.0 and 5.2, variable names <var> of the result pattern are case-sensitive, as opposed to the case-insensivity of all other syntax parts. In FlagShip, the whole syntax is not case-sensitive.

Related:

std.fh, #define, `#translate`, `#xtranslate`

#comments, #nocomments

Syntax:

`#comments`

Syntax:

`#nocomments`

Purpose:

Enables or disables the transfer of full-line comments into the .c code produced. The default is #comments.

Arguments:

none.

Description:

Normally, FlagShip transfers all full-line comments (*, NOTE, // or && at line beginning and the entire line or multiline /*..*/ comments) into the .bp and C code produced for better orientation.

When the comments are not required, e.g. to avoid comments from the `#include` file, specify the `#nocomments` directive. To enable the transfer again, use the #comments directive. See also the *<FlagShip_dir>/include/std.fh* file.

Also, when more than five subsequent empty lines occur in the .prg source, they will be not transferred to the .bp and .c files. The synchronization of the .prg line number is then maintained by the FlagShip #line pragma.

Example:

```
#nocomments
* this comment line is not transferred to C
x = 5
#comments
// this comment line is transferred to C
y = x
```

Compatibility:

Available in FlagShip only.

Related:

CMD.NOTE, * comments

#debug_off #debug_on

Syntax:

`#debug_off`

Syntax:

`#debug_on`

Purpose:

Temporarily disables and enables debugger. Considered only if the debugger is active, i.e. if not the -nd or -nl switches was given. Should always apply as a pair #debug_off ... #debug_on within the same source file.

Arguments:

none.

Description:

This directive is useful and should be used in sources inserted by the `#include` directive. This is because the GUI debugger does not consider the included sources and hence may then report or stop on an incorrect line number.

The directives may also be used to automatically skip a large part of already tested application.

The #DEBUG_OFF directive disables debugger information so the debugger will continue execution and ignore breaks until the debugger information is enabled again via the #DEBUG_ON directive. If the debugger info is not active at all (e.g. when `-nd` or `-nl` compiler switches are used), these directives are ignored.

Example:

```
? "hello world"
#debug_off
#include "mysource.prg"      // not a good idea,see #include
#debug_on
? "continuing"
```

Compatibility:

Available in FlagShip5 only.

Related:

compiler switches FSC.1.3, debugger FSC.5

#define, #undef

Syntax:

```
#define <identifier> [<constant>]
#define <identifier>([<args>]) [<constant>]
```

Syntax:

```
#undef <identifier>
```

Purpose:

Define or remove a manifest constant or pseudo-function.

Arguments:

<identifier> is a case-sensitive name, which conforms to the FlagShip naming convention, i.e. the <identifier> can contain any combination of letters (A..Z, a..z), numbers (0..9) and underscore character ("_"). Special characters like -, /, $, :, umlauts, etc. are not allowed. The <identifier> name is significant in full length and is case sensitive. As a convention, identifiers are usually specified in uppercase to distinguish them from other identifiers (variables, function names etc.) used within a program. Pre-defined manifests are available in *<FlagShip_dir>/include/*.fh* files (see e.g. set.fh, inkey.fh, error.fh, box.fh, etc.) which are assigned to current source file by the `#include` directive.

<identifier>() is a case-sensitive name of a pseudo-function without arguments. The parentheses must immediately follow the identifier.

<identifier>(<args>) is a case-sensitive name of a pseudo-function with a comma-separated argument list. The parentheses must immediately follow the identifier. The arguments <args> are case- sensitive, since the syntax is only a special case of the regular `#define` directive. The argument names used are visible only for the `#define` declaration.

<**constant>** is the replacement literal text or expression to substitute the <identifier> whenever it is encountered. When an expression is used, enclose it in parentheses to guarantee the precedence of later evaluation. At least one white space character (blank or tab) separates the <identifier> from the <constant> part.

Description:

The `#define` directive defines an identifier and, optionally, associates a text replacement to it. If the <constant> text is specified, its contents will replace all subsequent occurrences of the <identifier> within the source file (similar to performing a search/replace in a text editor), except for the replacements in string constants.

If the <constant> is not specified, all occurrences of <identifier> are removed by the preprocessor. The `#define` directive also influences the contents of other preprocessor directives, but you cannot change the directive name itself (like `#define` define undef).

The `#undef` directive removes a previously declared <identifier> from the internal preprocessor stack. All subsequent program lines which include the <identifier> remain unchanged. To prevent a compiler warning which occurs when an existing identifier is redefined, use `#undef` to remove an identifier before you redefine it with `#define`.

Using a manifest constant instead of the constant itself increases program readability and reduces maintenance time. Although there is some similarity between `#define`-ing manifest constant and assigning the constant to a variable, using the `#define` directive decreases run-time overhead and the code size required for the variable handling.

Since the <identifier> is case-sensitive, it is a general convention to define and use it in uppercase to distinguish it from other identifiers and variables used within a program. Additionally, in the standard FlagShip `#include` files, the `#define` identifiers are prefixed with a group of unique letters (e.g. K_ for keys, B_ for boxes, `_SET_` for settings, `DBS_` for database structures, `F_` for files, etc.) to distinguish them from other identifiers.

Pseudo-Functions

By using the <identifier>([<args>]) syntax, you can also define pseudo-functions which are resolved at compile time. Pseudo-functions differ from manifest constants in that they support arguments.

Whenever the preprocessor scans a source line and encounters a function call that matches the pseudo-function definition, it substitutes the function call with the replacement expression along with its arguments. In the program text, the argument count need not exactly match the number of arguments in the `#define` directive.

The advantage of the pseudo-function is increased program readability, as a result of reducing the maintenance time and the run-time overhead compared to invoking the regular function.

You may also use the similar `#xtranslate` directive to avoid the case-sensitivity of the identifier and to use optional arguments.

Compiler define's

When starting the compiler, you may also define an <identifier> and optionally the <constant> by the `-Didentifier` or `-Didentifier="constant"` switch (see section FSC), which is equivalent to `#define` directives in all the compiled programs.

Specifying the `#define` directive in the std.fh file has a similar effect, but affect all .prg programs as well as subsequent compilations.

#define FlagShip

The FlagShip preprocessor automatically issues the `#define` FlagShip `FLAGSHIP` directive (note the upper/lower case letters) at the beginning of each .prg file, similar to the compiler switch `-DFlagShip=FLAGSHIP`. You may therefore distinguish between FlagShip and Clipper source lines simply by using the `#ifdef` FlagShip directive.

Example:

```
#ifdef FlagShip                          // true on UNIX
  #nocomments
  #include "inkey.fh"                    // INKEY() codes, e.g. K_ESC
  #define MYPATH /usr/data               // comments are possible
  #define K_ABORT_TEXT "Ctrl-K"
#else                                    // code for DOS follows
  #include "INKEY.CH"
  #define MYPATH \usr\data
  #define K_ABORT_TEXT "Alt-D"
#endif
#define myMAX(arg1, Arg2) (IF(arg1 > Arg2, arg1, Arg2))
SET Path To MYPATH                       // SET PATH TO /usr/data
? "text including MYPATH"                // ? "text including MYPATH"
WAIT "to abort the program, use " + K_ABORT_TEXT
IF LASTKEY() == K_ESC                    // IF LASTKEY() == 27
   QUIT
ENDIF
? myMAX(2,3)                             // ? (IF(2 > 3, 2, 3)) --> 3
a = b * myMAX(a,5)                       // a = b * (IF(a > 5, a, 5))

Note:  do  not  use  pre/post-increment  or  pre/post-decrement  of
arguments  in the  pseudo-functions, since  they will  be evaluated
twice, e.g.

a := 5 ; ? myMAX (a++, 0), a           // 6  7
a := 5 ; ? MAX   (a++, 0), a           // 5  6  (std. function)
```

Compatibility:

Available in FS4 and C5 only.

Related:

#ifdef, #ifndef, `#translate`, `#xtranslate`

#error

Syntax:

`#error [<text>]`

Purpose:

Generates a compiler error and displays a message on the terminal.

Options:

<text> is the literal string of the message to be displayed. Do not enclose the message <text> in quotations unless you want them to appear as part of the display.

Description:

The `#error` directive causes the compiler to terminate compilation with return code 1. If the optional <text> is specified, an error message is displayed to stderr.

Example:

```
#ifndef FlagShip
#error This program is developed for UNIX only.
#endif

#ifdef FINAL
  #ifdef TEST
    #error Remove #define TEST for the final compilation
  #endif
#endif
```

Compatibility:

Available in FS4 and C5 only.

Related:

#stdout, #ifdef, #ifndef

#ifdef, #ifndef ... #else ... #endif

Syntax:

```
#ifdef <identifier>
     any valid statement ...
[#else
     any valid statement ... ]
#endif
```

Syntax:

```
#ifndef <identifier>
     any valid statement ...
[#else
     any valid statement ... ]
#endif
```

Purpose:

Compiles a section of code if an identifier is defined or not defined.

Arguments:

<**identifier**> is the case-sensitive name of a definition whose existence is being verified.

Description:

`#ifdef`...`#endif` performs conditional compilation when the identifier is defined by using the `#define` directive or the -D compiler switch. The optional #else directive specifies the code to be compiled if the <identifier> is not defined.

`#ifndef`...`#endif` performs a conditional compilation when the identifier is not defined. The optional `#else` directive specifies the code to compile if <identifier> is defined.

The `#endif` directive ends the conditional compilation block.

Conditional compilation is particularly useful when maintaining many different versions of the same program.

To create portable programs for DOS or Windows/Unix, `#ifdef FlagShip` directive is the most comfortable way of observing the slight differences between these two systems.

Pre-defined preprocessor compiler directives (case sensitive):

#ifdef FlagShip	always true in FlagShip
#ifdef FlagShip5	true with FlagShip 5.x and newer
#ifdef FlagShip6	true with FlagShip 6.x and newer
#ifdef FlagShip7	true with FlagShip 7.x and newer
#ifdef FlagShip8	true with FlagShip 8.x and newer
#ifdef FS_WIN32	true with FlagShip for MS-Windows 32bit or 64bit
#ifdef FS_WIN64	true with FlagShip for MS-Windows 64bit

#ifdef FS_BCC32	true with FlagShip for MS-Windows BCC32 32bit
#ifdef FS_LINUX	true with FlagShip for Linux 32bit or 64bit
#ifdef FS_LINUX32	true with FlagShip for Linux 32bit
#ifdef FS_LINUX64	true with FlagShip for Linux 64bit
#ifdef VFS_FS2TOOLS	true with available FS2 Toolbox

Example:

```
*** test.prg
#ifdef FlagShip
   #include "fspreset.fh"
   #include "inkey.fh"
   #stdout Compiling FlagShip specific
#else
   #include "INKEY.CH"
   #stdout Compiling DOS specific
#endif
#define TEST          // or use: FlagShip test.prg -DTEST
// later

#ifdef FS_LINUX
   RUN ("ls -la * | less")
   REFRESH
#else
   RUN ("DIR *.* | more")
#endif
#ifdef TEST
   ? "The current directory is", CURDIR()
   password = "MyTest"
   ? "The test password is", password
   WAIT
   #stdout The test option IS ACTIVE now
#else
   #stdout Compiling w/o test option
#endif
CLEAR SCREEN
#ifndef TEST
   ACCEPT "Please enter your password:" TO password
   IF EMPTY(password)
      QUIT
   ENDIF
#endif
```

Compatibility:

Available in FS4 and C5 only.

Related:

`#define`, section FSC

#include

Syntax:

```
#include "<filename>"
```

Purpose:

Includes a source or header file into the current source file.

Arguments:

<**filename**> specifies the name (optionally preceded by a path) of another source or header file to be inserted at the current position in the source file. The <filename> must be enclosed in quotation marks ".." or '..'.

FlagShip will try to infer the `#include` "filename" if it is not able to find it as given, using the following search algorithm:

1. Look for the file name as given (by e.g. `#include` "File.Ext") in the:
   a. current directory,
   b. path given by the -I switch (if specified),
   c. /usr/include directory.
2. Repeat step 1.a to 1.c with "file.ext" (in lowercase),
3. Repeat step 1.a to 1.c with "FILE.EXT" (in uppercase)

Note: The FS_SET("lower", "pathlower") etc. has no effect on the preprocessor, since the FS_SET() function is an executable statement invoked at run-time.

Description:

`#include` inserts the contents of the specified file in place of the directive in the source file. In FlagShip the convention is that header files with .fh extensions should contain only preprocessor directives and external declarations.

Header files often contain general purpose constants, such as the manifests for key values in "inkey.fh", file attributes in "fileio.fh" or user-defined `#define`(s). Including these header files in the .prg source makes their definitions automatically available in it.

The scope of directives from the included header file is the current .prg program file.

The `#include` directives may be nested up to a depth of 512 levels. That means, one `#include` file may include another one, which includes another file, etc. It is a good programming practice to omit files already included by using the `#define` and `#ifdef` directives in order to avoid infinite include loops.

std.fh, set.fh

When the compiler switch -nI is not used, FlagShip automatically includes the "std.fh" file at the beginning of each .prg program. These standard header files (by default located in *<FlagShip_dir>/ include*) contain the definitions of all FlagShip commands and standard functions. It also automatically includes the "set.fh" and "inkey.fh" files, if they are not yet included.

If changes of the std.fh or set.fh file contents' are desired, you have two basic choices:

a. Create an additional header file, e.g. "stdadd.fh" containing all changes and additions required and insert the line `#include` "stdadd.fh" at the end of the default std.fh file. Since directives specified later have higher priority, they will override standard directives given in std.fh and set.fh.

b. Copy the <FlagShip_dir>/include/std.fh file to a new name (e.g. mystd.fh), make the changes and compile with the -nI=mystd.fh option.

The first is the better solution, since your additions will not be affected by the likely changes to the standard header files in a new FlagShip release.

Note: it is usually not a good idea to `#include` .prg files, since it would be hard to debug it (the line-numbering of every included file starts with 1), and the automatically created code blocks (e.g. for @..GET..) may get confused (the line number is a part of these auto codeblock names) - linker error may result. Better is either to use SET PROCEDURE TO.. or to compile separately to object, or simply add these .prg sources to FlagShip command line at compilation.

Standard header files

In addition to the std.fh and set.fh described above, FlagShip provides a number of header files containing manifest constants for common operations. Refer to sections GEN, CMD, FUN and look out for *.fh files in the default installation directory <FlagShip_dir>/include (e.g. ls -l /usr/local/Flag*/include/*.fh).

Example:

```
*** test.prg
#include "inkey.fh"
#ifndef _MY_INCLUDED
   #include "myinclude.fh"
   #define _MY_INCLUDED
#endif
LOCAL key, ok

displaymenu ()
key = INKEY()
ok  = perform_menu (key)

FUNCTION perform_menu (key)
DO CASE
CASE key == K_ESC              // Escape key, 27
   QUIT
CASE key == K_PGUP             // PgUp key, 18
   SKIP -1
CASE key == K_PGDN             // PgDn key, 3
   SKIP 1
CASE ....                      // other keys
OTHERWISE
   RETURN .F.
ENDCASE
RETURN .T.
```

Compatibility:

Available in FS4 and C5 only. To use the Clipper extensions for FlagShip *.fh files in an unmodified source code, make in Linux a link e.g.

```
  ln <FlagShip_dir>/include/inkey.fh inkey.ch
```

In extreme cases, you may use the original CA/Clipper *.CH include files by copying them to your working directory, with the exception of "STD.CH", "FILEIO.CH" and "ERROR.CH". You must use "std.fh", "fileio.fh" and "error.fh" which are FlagShip specific, but are backward compatible to Clipper.

Related:

`#define`, section FSC

#stdout

Syntax:

`#stdout [<text>]`

Purpose:

Displays a message on the terminal.

Options:

<**text**> is the literal string of the message to be displayed. Do not enclose the message <text> in quotations unless you want them to appear as part of the display.

Description:

The `#stdout` directive causes the preprocessor to display a message on stderr, the default device for compiler messages.

If the compiler stderr output is re-routed to a file, e.g.

```
  FlagShip my*.prg -q -Mmymain 2>protocoll.txt
```

the `#stdout` message is also printed to that file.

Example:

```
** test.prg
#ifdef FlagShip
#stdout Compiling on UNIX using FlagShip
#endif

#ifdef TEST_ONLY
  #stdout "TEST_ONLY" directive is used
#endif
```

Compatibility:

Available in FS4 and C5 only.

Related:

#error, `#define`, #ifdef, #ifndef

#translate, #xtranslate

Syntax:

`#translate <pattern> => <result>`
`#xtranslate <pattern> => <result>`

Purpose:

Specifies a user defined translation directive.

Arguments:

<**pattern**> is the input which the text should match. The <pattern> consists of literal symbols (keywords, clauses, etc.) and optional variables, enclosed in angle brackets, e.g. <var>, confirming to the FlagShip variables naming convention. The entire entry is not case-sensitive.

=> This equals sign immediately followed by a greater than sign is a literal part of the syntax and separates the <pattern> from the <result>.

<**result>** is the text to be produced if a portion of the input text matches the <pattern>. It may include literal constants, clauses and result variables <var> enclosed in angle brackets. The whole <result> part of the syntax is optional, the resulting expression must comply with a valid FlagShip syntax. If no <result> is specified, an empty line is produced.

Description:

The `#[x]translate` directive provides a way to re-define a part of a program statement. `#[x]translate` is similar to `#[x]command`, but it also matches input text that is not a complete statement, while `#[x]command` matches only if input text is a complete statement. In general, `#[x]command` is used for most user-defined-commands, while `#[x]translate` for special cases. `#[x]translate` is also similar to the `#define` directive, but is more powerful and not case- sensitive.

The `#translate` directive supports the abbreviation of the matching pattern (keyword and clauses) with up to four characters, while `#xtranslate` only translates patterns exactly as given (but case insensitive). All other rules apply to both directives.

You may re-define standard [x]translate's from std.fh or stdfoxpro.fh by your own (e.g. in the .prg or your own .fh file), the last definition is the most current (lifo order = last-in-first-out).

Pattern: Result:

Since all the rules of `#[x]command` also apply to `#[x]translate`, refer to section `#command` for a description of the <pattern> and <result> parts.

Example:

```
#translate AllTrim(<arg>) => LTRIM(RTRIM(<arg>))
#translate MYMAX (<max>, <min>)                      ;
  =>  (IF (<max> >= <min>, <max>, <min>))
#translate MYMIN (<max>, <min>)                      ;
  =>  (IF (<min> <= <max>, <min>, <max>))
#translate MAXunknown (<max>, <min>)                 ;
```

```
  =>  (IF ((VALTYPE(<max>)+VALTYPE(<min>))=="NN",   ;
           MAX (<max>, <min>),                      ;
           IF (VALTYPE(<max>)=="N", <max>, <min>)) )
#translate MAX3 (<arg1>, <arg2>, <arg3>)            ;
  =>  (IF (<arg1> >= <arg2>, MAX (<arg1>,<arg3>),   ;
           MAX (<arg2>,<arg3>) ))

LOCAL a, b, c
a = VAL (alltr ("   1234  "))     //  1234
b = MAXU (a, c)                   //  1234, no RTE
c = myMAX  (a, -b) +1             //  1235
c = myMIN  (a, -b)                // -1234
d = MAX3 (a, c, b)                //  1235

Note:  do  not  use  pre/post-increment  or  pre/post-decrement  of
arguments  in these pseudo-functions, since  they will be evaluated
twice, e.g.

a := 5 ; ? myMAX (a++, 0), a      // 6  7
a := 5 ; ? MAX   (a++, 0), a      // 5  6  (std. function)
```

Compatibility:

Available in FS4 and C5 only. In Clipper 5.0 and 5.2, variable names <var> of the result pattern are case-sensitive, as opposed to the case-insensitivity of all other syntax parts. In FlagShip, the whole syntax is not case-sensitive.

Related:

std.fh, `#define`, `#command`, `#xcommand`

Index

Notes

MULTISOFT

multisoft Datentechnik
Schönaustr. 7
D-84036 Landshut

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