Program Coco

Dan Nagle (converted to bookdown and expanded by Erik Olofsen)

2023-08-12

Contents

Chapter 1

About the Program Coco

CoCo provides preprocessing as per Part 3 of the Fortran Standard (CoCo is short for “conditional compilation”). It implements the auxiliary third part of ISO/IEC 1539 (better known as Programming Languages- Fortran), and supports several extensions (see Chapter 6). (Part 1 of the standard defines the Fortran Language proper. Part 2 is the ISO_VARYING_STRINGS standard, which is sometimes implemented as a module.) A restore program, similar to that described in the CoCo standard, is also available for download.

Note that Part 3 of the Fortran Standard has been withdrawn, and Part 2 very likely will be withdrawn within a few years. At least some of the functionality of Part 2 is available via allocatable character entities, whether all of it will be made available is not yet decided.

Generally, CoCo programs are interpreted line by line. A line is either a CoCo line or a source line. Lines with the characters “??” in columns 1 and 2 are CoCo lines. All other lines are expected to be Fortran source lines (of course, the other lines need not actually be Fortran source lines, but they are called source lines in the discussion on this page). Except for the “??” characters in columns 1 and 2, CoCo lines follow the same rules as Fortran free form source lines. Like Fortran free form lines, CoCo lines are continued by placing an “&” as the last character of the line to be continued. A CoCo comment is any text following a “!” following the “??” characters. A CoCo comment may not follow the “&” used to continue a quoted string onto the next line. Text in CoCo lines may appear in upper case or lower case interchangeably, CoCo preserves case in CoCo lines written to the output file and ignores case in source lines. Either single quotes or double quotes may be used to delimit strings. CoCo directives are rendered in upper case on this page for clarity, and because the standard document uses upper case.

CoCo directives may define integer constants and variables, and logical constants and variables. CoCo uses a strong type system, which means that all symbols must be declared and that integer symbols and logical symbols are not interchangeable. The distinction between constants and variables provides a further level of control. Conditional compilation is expressed by CoCo IF blocks, which start with CoCo IF directives. The conditions are determined by CoCo logical expressions, which may include relational operations between CoCo integer expressions. CoCo IF-blocks function analogously to Fortran IF-blocks, with IF, ELSE IF, ELSE and END IF directives. CoCo IF-blocks select which lines will be passed to the output source file for compilation. An INCLUDE directive is also available so preprocessing may be applied to the included lines as well. The complete set of standard directives is listed in Chapter 2.

As implemented here (see Chapter 4), a one invocation of CoCo reads one or more input source files and writes one output source file to be compiled. A short example of a CoCo program follows, the lines with initial "??" are the CoCo lines, lines without the initial "??" are Fortran source lines:

```
?? ! used to choose whether the intrinsic module sets stdout
?? logical, parameter :: has_intrinsic_module = .false.

program hello_world

?? ! use number or asterisk
?? if( has_intrinsic_module )then
!  output_unit is stdout
use, intrinsic :: iso_fortran_env, only: output_unit

write( unit= output_unit, fmt= *) 'Hello, world!'
?? else
!  * is stdout
write( unit= *, fmt= *) 'Hello, world!'
?? endif

stop

end program hello_world
```

Above, lines starting with "?? !" are CoCo comments (lines starting with "!" are, of course, Fortran comments), **has_intrinsic_module** is a CoCo logical constant given the value false, and the CoCo IF block is used to select which of two sets of source lines supplies the Fortran comment and write statement. If the above example is contained in a file named `hello_world.fpp`, then a command line **coco hello_world** produces `hello_world.f90` with unit * written. A command line **coco -Dhas_intrinsic_module hello_world** produces `hello_world.f90` with unit output_unit written. (The -D option changes the value of **has_intrinsic_module**. See command line options for more.)

1.1 The Portability Project and CoCo

The Portability Project has been largely superseded by improvements to the standard-defined `IS_FORTRAN_ENV` intrinsic module. There is an env2inc program to write the type kind values provided in `ISO_FORTRAN_ENV` to a CoCo include file for use when preprocessing programs. Of course, env2inc must be compiled by the same processor to be used with the preprocessed program.

Chapter 2

Standard CoCo

A CoCo program consists of CoCo lines and source lines. The source lines comprise the Fortran program proper. The CoCo lines direct CoCo in preprocessing the program. A CoCo program may have an optional set file associated with it (see Chapter 3). The set file may be used to control what CoCo does with CoCo lines and with source lines which are not intended to be part of the output source code. Some directives may appear only in the set file, some may appear only with the source code, and some may appear in either. The set file may declare CoCo symbols and if so, the set file values override the value contained in the program declaration. The declarations must match, however, as far as type and whether the symbol is a constant (this keeps a CoCo program self-contained and self-consistent).

As an extension (see Section 6.9), the set file also allows the programmer to set some values that may also be set by command line options (see Section 4.2) in case CoCo doesn't have access to the command line (if the processor used to compile CoCo does not support command line access).

The standard CoCo directives are the INCLUDE directive, INTEGER and LOGICAL declarations and assignments, IF/ELSE IF/ELSE/END IF directives, and MESSAGE and STOP directives. An integer or logical variable may be given a value where declared, and if so, it may be declared to be a constant. Any directive may appear in upper case or lower case interchangeably. Names of CoCo symbols are interpreted without regard to case.

A source line in the input is said to be active when it is selected to appear in the output as a source line (that is, not as a comment). A line not selected to appear in the output as a source line is said to be inactive. The fate of inactive source lines and CoCo lines is controlled by the ALTER directive in the set file (see Section 3.1). Active lines and inactive lines are selected by CoCo IF blocks, analogously to the selection of executed statements by Fortran IF blocks.

The standard CoCo directives are described on the list below:

- `?? INCLUDE 'file-name'`

  The INCLUDE directive is replaced in the output by the contents of the named file. There must not be a comment after the file name.

  The file name may appear in single quotes or double quotes. The include directive may not be continued onto subsequent lines. Depending on the mode selected by the ALTER directive, the include file's contents may be marked in the output source file with INCLUDE and END INCLUDE comments identifying the file included.

- `?? INTEGER [, PARAMETER ] :: name [ = expression ] [, name [ = expression ] ] ...`

  The INTEGER directive declares one or more integer variables with the names given; the values are assigned if the expressions are present. The name must not be the name of another integer, a logical, a macro, a text block, or a file variable. A value must be assigned before the variable may be used. If the PARAMETER attribute is present, the expressions must be as well, and the names are the names of constants and may not be assigned a value subsequently in any source file. An integer declaration in the set file must supply a value for either a constant or a variable.

- `?? LOGICAL [, PARAMETER ] :: name [ = expression ] [, name [ = expression ] ] ...`

  The LOGICAL directive declares one or more logical variables with the names given, the values are assigned if the expressions are present. The name must not be the name of an integer, another logical, a macro, a text block, or a file variable. A value must be assigned before the variable may be used. If the PARAMETER attribute is present, the expressions must be as well, and the names are the names of constants and may not be assigned a value subsequently in any source file. A logical declaration in the set file must supply a value for either a constant or a variable.

- `?? IF( logical-expression )THEN`

  Introduces an IF-block, the IF-block must be closed by an END IF directive.

  CoCo IF-blocks function analogously to Fortran IF-blocks. Lines in the block following the IF or ELSE IF directive with the first true expression are active (copied to the output as source lines), lines following other IF or ELSE IF directives in a block are inactive (are not copied to the output as source lines). Lines following the ELSE directive are active if none of the logical expressions on the preceding IF or ELSE IF directives in the block evaluated to true.

- `?? ELSE IF( logical-expression )THEN`

Introduces an ELSE IF portion of an IF-block. The logical expression is evaluated to see whether it is true, the lines following the first true expression encountered in an IF-block are active. There may optionally be whitespace between the ELSE and the IF.

- `?? ELSE`

  This directive must follow all ELSE IF directives in an IF-block. The lines following an ELSE directive are active if no other lines within the if-block were. If no ELSE directive is present in an IF-block and no expression evaluates to true, no lines from the block are active.

- `?? END IF`

  Ends an IF-block. There may optionally be whitespace between the END and the IF.

- `?? MESSAGE [ item [, item ] ... ]`

  Each item is a quoted string or an expression whose value is printed to the log file, or to stderr.

- `?? STOP`

  Causes CoCo processing to halt. A message is printed to the log file, or to stderr.

- `?? name = expression`

  Assigns a new value to the integer or logical variable whose name appears on the left side of the equals.

  The expression may be a literal value (for example, 42 or .true.), a CoCo variable, or a CoCo expression using CoCo operators (+, -, *, / with integers; .and., .or., .eqv., .neqv., .not. with logical operands; relational .eq., ==, .ne., /=, .lt., <, .le., <=, .ge., >=, and .gt., > produce a logical result between integer operands). The usual Fortran precedence rules apply. Parentheses are honored. As an extension, the \ character is treated as a modulus operator, with the precedence of the multiply operator and the division operator.

- `?? ! [ commentary ]`

  Is ignored by CoCo and may be used to document the CoCo program, as with any CoCo line its fate is set by the ALTER directive. A CoCo comment may appear on any CoCo directive (except the INCLUDE directive or when continuing a quoted string) following a ! character. A line which is blank after the ?? characters is considered to be a comment.

The `name` and `name = expression` forms of the integer variable and logical variable declarations may be mixed; a name declared to be a constant, of course, must be supplied with a value. A declaration in the set file must have a value, the purpose of the set file declarations (or a command line values) is to supply alternative initial values.

All the statements comprising an `IF` construct must appear in the same source file and CoCo directives in blocks not appearing in the output must be well formed directives. This implementation does minimal checking of directives in inactive lines. Input lines in TEXT blocks are treated similarly. An `IF` block within a TEXT block is interpreted during execution of the COPY directive.

An example CoCo program follows. As in the example above, there are CoCo comments, CoCo lines, Fortran comments and Fortran source lines:

```
?? ! the logical variable will be used
?? ! to choose whether to use array syntax
?? ! or explicit loops
?? logical :: array_syntax

?? ! suppose the compiler is either v1.0 or v2.0
?? ! the integer variable here supplies
?? ! the default value here and might be overridden
?? ! by a value from the set file or command line
?? integer, parameter :: version = 20

?? ! which version selects whether array syntax is used
?? if( version == 20 )then       ! efficient array code
??    array_syntax = .true.      ! so use array syntax
?? else if( version == 10 )then  ! inefficient array code
??    array_syntax = .false.     ! so no array syntax
?? else                          ! unknown version
??    message 'error: unknown version: ', version
??    stop                       ! go no further
?? end if

program sum_arrays
implicit none

real, dimension( 100) :: a, b, c

?? ! need loop index when using explicit loops
?? if( .not. array_syntax )then
integer :: i             ! loop index
?? endif

read( unit= *, fmt= *) b, c

! element-wise sum
?? if( array_syntax )then
```

```
a = b + c
?? else
do i = 1, 100
   a(i) = b(i) + c(i)
end do
?? end if

! evaluate the final sum and print it
write( unit= *, fmt= *) 'sum of a: ', sum(a)
stop

end program sum_arrays
```

Above, a CoCo integer, **version**, is used to choose a value for a CoCo logical variable, **array_syntax**. If version has been set incorrectly (the CoCo else clause), CoCo will print the explanatory message and stop. The CoCo logical variable is then used to choose whether an auxiliary integer is compiled into the Fortran program to serve as the index of the explicit loop. Then the CoCo logical variable is used to select either explicit loops or array syntax.

Note the strong typing CoCo uses: the value of the logical symbol can be true or false, while the value of the integer is numeric. This provides a degree of error checking. As extensions, macros, TEXT blocks, and file variables also have distinct declarations and uses.

Chapter 3

The CoCo Set File

The programmer may use a separate file, called a set file, which permits the programmer to change the values of variables and constants outside the CoCo program, and to specify what happens to inactive source text and CoCo directives. At most one set file is read and processed for each invocation of CoCo. Within the set file, variable declarations and constant definitions may appear, the values supplied override those of the same name which appear in any of the input files. There must be a declaration of the same variable or constant within one of the input files and the declaration must appear before any use of the name, only the value may be changed by the set file. See also Section 6.9 for more directives which may appear in the set file. A declaration in a set file must supply a value. The value can differ from that in a source file. Other attributes of a set file declaration must match those in the source file. A declaration is unknown to CoCo until the declaration in a source file is found. This keeps a CoCo program self-consistent and self-contained.

Note that while CoCo can't tell that a symbol defined in the set file or command line isn't defined within a source file until it's finished processing, CoCo complains if the symbol is referenced before a definition appears in a source file. It's required to declare a CoCo variable in the source file before using it, otherwise a missing set file or forgotten command line option would cause the preprocessor to operate unexpectedly. The set file is appended, following a message, to the end of the output file, the command line options are available via cmdline predefined macro and the CMDLINE directive. The effects of the command line and set file may also be checked via the OPTIONS directive.

The set file is appended to the end of the CoCo program's output (depending on the mode set by the ALTER directive). If visible, its contents are separated from the program source by a standard-specified Fortran comment (thus, the set file may be the cause of several blank lines after the last line of source).

The set file is named according to the -s command line option (see Section 4.2).

If there isn't one, the name is set according to the output file name. The file name suffix, if any, is discarded and ".set" is appended to make the set file name. If a file with that name is not found, the value of the environment variable `COCO_SET_FILE` is checked. If a file with that name is not found, a file with the default name of "coco.set" is sought. This allows a programmer to have a default set file for a project or directory, or to control the preprocessing on a file-by-file basis. There is only one set file read for a multi-input file invocation of CoCo. See Using CoCo for more on the command line.

3.1 The ALTER Directive

The fate of inactive source lines and of CoCo directives (that is, of all lines not appearing in the output file as active source lines) is controlled by the ALTER directive. At most one alter directive may appear in a set file. The -a command line option (see Section 4.2) overrides the alter directive. The possible alter modes and their effects are described in the following list:

- `?? ALTER: DELETE`

  All CoCo lines and inactive source lines are deleted from the output file. These lines will not be seen by the compiler. While this produces the easiest to read output source file, it does potentially change line numbering (see Line Numbering for more details).

- `?? ALTER: BLANK`

  All CoCo lines and inactive source lines are replaced by blank lines in the output file. These lines will not be seen by the compiler, but line numbers are preserved.

- `?? ALTER: SHIFT0`

  All CoCo lines are printed with the leading "?" replaced by a "!". Inactive source lines are printed with a "!" in place of the character in column one. These lines will be seen by the compiler as comments. This means the entire line, less only the initial character, is visible in the source to be compiled; no characters are shifted off the end.

- `?? ALTER: SHIFT1`

  All CoCo lines are printed with a "!" added before the leading "?". Inactive source lines are printed with a "!" preceding the leading character, CoCo issues a warning if a line so extended exceeds 132 characters. These lines will be seen by the compiler as comments.

- `?? ALTER: SHIFT3`

  All CoCo lines are printed with a leading "!?>" before the leading "?". Inactive source lines are printed with "!?>" preceding the leading character, CoCo issues a warning if a line so extended exceeds 132 characters. These lines will be seen by the compiler as comments. The leading "!?>" makes it easy to write a program to undo the effects of the CoCo preprocessor (at least so long as the use of CoCo conforms to the standard without use of any extensions and the source code doesn't have any comments beginning with "!?>" which would confound the restore process). See the restore program available for download (see Chapter 10) (this is similar to the program specified in the standard).

The following table summarizes the effects of the alter modes:

ALTER Mode	Preserve Line Numbers?	Preserve Lengthen Line?	Undo?
DELETE	NO	N/A	NO
BLANK	YES	N/A	NO
SHIFT0	YES	YES	NO
SHIFT1	YES	NO	NO
SHIFT3	YES	NO	YES

A good default set file contains the line `?? ALTER: DELETE`, which overrides the standard specified default of `SHIFT3`. The same effect may be had by adding -ad to the CoCo command line. An example set file follows:

```
?? ! no CoCo lines or inactive source lines
?? ! appear in the source to be compiled
?? alter: delete
?? ! no debugging (unless -Ddebug is on the command line)
?? ! debug should also be declared as a logical constant
?? ! in a source file before being used in a source file
?? logical, parameter :: debug = .false.
```

The alter line causes CoCo lines and inactive source lines to disappear from the output source file, and the declaration of the logical constant in the set file may override or confirm the declaration of the same name appearing in the source input file. (There should be a declaration named 'debug' as a logical constant in the source input.)

Chapter 4

Using Coco

CoCo is distributed as source code and is written in standard Fortran 2003. You will have to find a replacement for the command line access routines if your processor does not support them (but most compilers do support them), or code them yourself using the command line routines your processor does support (for example, iargc() and getarg()). Procedures mimicking the standard procedures are available from I.S.S. Ltd. via their free F2KCLI Module for a very wide variety of compilers.

4.1 Synopsis

```
coco -V

coco -h

coco [ [options] [--] ] [ base-name | output input [...] ]
```

CoCo responds to the "-V" option by printing its version information and quitting. CoCo responds to the "-h" option by printing its command line options and quitting. If it finds no file name arguments on its command line, CoCo reads its input from stdin and writes its output to stdout (but see the input set file directive and the output file set file directive extensions). A lone file name command line argument is a file basename: it has ".fpp" appended to it and used as the single input file name, ".f90" is appended and used as the output file name. If more than one file name argument appears on the command line the first is taken to be the output file name, the rest are treated, in the order of occurrence, as input file names.

Examples of CoCo usage follow:

```
coco <input.f >output.f90
```

causes CoCo to read the file input.f and to write the file output.f90. A set file named "coco.set" is sought.

```
coco -V
```

causes CoCo to print its version information to stderr and stop. See also the coco macro.

```
coco -h
```

causes CoCo to print short summary of its command line options to stderr and stop.

```
coco source
```

causes CoCo to read source.fpp and write source.f90. The file name source.set is checked to see if it exists, if so, it is processed as the set file. If source.set is not found, the value of the environment variable `COCO_SET_FILE` is checked to see if it exists, if so, it is processed as the set file. If not, coco.set is checked to see if it exists, if so, it is processed as the set file. If coco.set is not found, no set file is processed.

```
coco output.f90 input1.f90 input2.f90 input3.f90
```

causes CoCo to read input1.f90, input2.f90, input3.f90 in that order and to write output.f90. The file name output.set is checked to see if it exists, if so, it is processed as the set file. If output.set is not found, the value of the environment variable `COCO_SET_FILE` is checked to see if it exists, if so, it is processed as the set file. If not, coco.set is checked to see if it exists, if so, it is processed as the set file. If coco.set is not found, no set file is processed.

The specification of input and output file names has a number of effects. In addition to setting the name of the set file (see Chapter 3), several macros are predefined by CoCo. The input file name and the set file name are available as macros (`file` and `setfile`, respectively). Also, there are `date` and `time` macros to document the preprocessing. See the predefined macros for more information about CoCo's predefined macros.

4.2 Options

Generally, the option on the command line overrides the set file directive controlling the same behavior. However, some options are cumulative or must have a matching directive, see the description of each option and directive for specifics. The following command line options are recognized:

- `-a?` sets the alter mode, with ? being one of d (delete), b (blank), 0 (shift0), 1 (shift1), 3 (shift3). This option overrides an ALTER directive in the set file. Only one `-a` option can appear on a command line.

- `-Dname[=val]` idefines name to be a logical or integer name and supplies the value, as if in the set file (see the -D rules). This option overrides the value provided by a definition of the same name by an integer directive or a logical directive in the set file or in an input file.

- `-ffile-name` names a freeze file to be read. The freeze file must contain only declarations. A freeze file may be prepared manually, or by a freeze directive appearing in an input file. The freeze directive behaves differently when present in a set file. There, the freeze file is read. In an input file, the freeze file is written with definitions and values current at its location. This option overrides a freeze directive in the set file. Only one `-f` option can appear on a command line.

- `-F` specifies fixed form source. By default, CoCo assumes free form source files The option sets the file suffix for the output file to ".f" from the default ".f90" and causes line wrapping to occur at column 72 rather than column 132 (see the form directive). Only one `-F` option can appear on the command line.

- `-h` makes CoCo print a short summary of the command line options to stderr and stop.

- `-Idirectory-name` adds the named directory to the list of directories CoCo searches for include files not found in the current directory (see Section 5.1 and the the directory directive). The directory occurs before those listed by directory directives in the set file. CoCo always searches the current directory first when seeking include files.

- `-lfile-name` names a log file to receive CoCo's reports, stderr is the default (see the logfile directive). This option overrides a logfile directive. Only one `-l` option can appear on a command line.

- `-m` turns on placing a comment line in the output marking the end-of-file of subsequent input source files (see the mark directive). This option overrides a mark directive. Only one `-m` option can appear on a command line.

- `-n` turns on source line numbering (see the number directive). This option overrides a number directive. Only one `-n` option can appear on a command line.

- `-p` disables posting a copy of the set file at the end of the output file and the separating line specified by the standard (see the post directive). This option overrides a post directive. Only one `-p` option can appear on a command line.

- `-r` writes a summary report to the logfile at the end of preprocessing. See the set file report directive and the source file report directive. Only one `-r` option can appear on a command line.

- `-sfile-name` names the set file. If the named set file does not exist, a set file named from the output file name is sought. If that file does not exist, the file named by the environment variable `COCO_SET_FILE` is sought. If that file does not exist, coco.set is sought. Only one `-s` option can appear on a command line.
- `-S` prevents CoCo from seeking a set file, unless a set file is named via a `-s` option. That set file will be sought. No other set file will be sought.
- `-v` makes CoCo work verbosely, reporting all file opening and closing, and a few other events, to the log file (see the verbose directive). Only one `-v` option can appear on a command line.
- `-V` makes CoCo print its version information to stderr and stop. See also the coco macro.
- `-w` turns off source line wrapping (see the wrap directive). This option overrides a wrap directive. Only one `-w` option can appear on a command line.
- `-W` turns off warning messages (see the warning directive). This option overrides a warning directive. Only one `-W` option can appear on a command line.

In general, the value of a command line option is used in preference to a value in the set file; the value in the set file is used in preference to a value in a source file.

Chapter 5

Topics on Using CoCo

5.1 About the -D rules

The following rules apply to the -D option: if the “=” is present, it must be followed by an integer literal value in which case the name is defined to be an integer name with the value specified. This value overrides an integer with the same name declared within the set file, if there is one, and within (one of) the input source file(s) (or an include file). If no “=” is present, then name is defined to be a logical name with a value of “.true.”. This value overrides a logical with the same name declared within the set file, if there is one, and within (one of) the input source file(s) (or an include file). Definitions from the command line override those in the set file, but like definitions in the set file they must match definitions in (one of) the source file(s). There is no way to give a value to a text variable or macro, or a file variable directly via `-D` (see Section 4.2). A name is effectively undefined until the declaration is seen in a source file. The set file value or command line value supersedes the value (if there is one) defined in the source file. A declaration in the set file must supply a value, even for declarations of variables.

5.2 About the INCLUDE search rules

The rules for directories specified by the `-I` command line option (see Section 4.2) and the DIRECTORY directive are that the directories specified by the `-I` command line option are searched first in the order they appear on the command line. Then directories appearing in DIRECTORY directives are searched in the order the directives appear in the set file. This way, the command line may override the set file. DIRECTORY directives may not appear in the source file(s) proper so all include files with the same name are known to come from the same directory, and thus each occurrence of any one named include file will be the same file. CoCo guesses what the file separator character (between directory names in an absolute file name) by examining the value found for the `cwd` predefined macro. CoCo gets this value from the `PWD` environment variable. If this fails, CoCo doesn't know what appropriate separator is, in that case, the appropriate separator (for example, "/", "" or":") must be appended after the directory name, whether it appears in the `-I` command line option, or in a DIRECTORY directive.

5.3 About Line Numbering and Wrapping

By default, source lines are wrapped starting at column 132 for free form source, and starting at column 72 for fixed form source. This position is called the wrap length. Wrapping means that if text extends beyond the wrap length, text beyond the wrap length on the line is ended, a continuation character is added (in free form source), and the remainder of the line is written as a continuation line following the wrapped line. This can be defeated with the `-w` (see Section 4.2 option, or with the WRAP directive. CoCo does not attempt to wrap comments.

When lines are numbered, the file name and line number are added to the end of the source line. Line numbering occurs starting past the end of the line whether the source form is free form (after column 132) or fixed form (after column 72). If the source line extends longer than the standard-specified length, the number appears after all text on the line. Line numbering adds, after the end of a line, a comment character, a blank, the file name, a colon and blank, and the line number of the source input file. This may help tracing a compiler message back to the original source file if CoCo is deleting inactive lines from the file seen by the compiler, or when macro expansion (see Section 6.3 or ASSERT directives lengthen lines too much, causing extra lines in the output, or when copy directives write text blocks or include files add source lines, or when lines are diverted to different output files via OUTPUT directives in source files.

5.4 About Multi-File versus Single-File Invocation

The value of CoCo variables change whenever a new value is assigned. CoCo constants, of course, do not change value during execution. When several input files are processed in sequence to make a single output file, the values of CoCo variables seen for each subsequent source file depend upon values that might be changed by previous input files. This allows, for example, the programmer to count the number of times a text block is copied throughout the entire preprocessing run. However, it also means that not all source input files are preprocessed in identical environments. Thus, the programmer must choose whether a given quantity should be represented by a CoCo variable or a CoCo constant, and whether a single input file or a multiple input file run is appropriate for a given purpose.

CoCo assumes a single invocation is for source files of either free form source or fixed form source, but that source form is never mixed. Since there is a single output file, it is difficult to mix the two. Automatic conversion utilities exist and some Fortran IDEs support reformatting but CoCo does not. Without due care, mixing free form source and fixed form source in a single CoCo execution may not work as intended. In any case, free form source is preferred for many reasons, so conversion of fixed form source files to free form source files is a good idea. CoCo only supports fixed form source to permit preprocessing of legacy codes.

As discussed above, CoCo may be used to make a single output source file from one or several input source files. The advantage of one input source file making one output source file is that one has a clear set of CoCo symbols in use. One may use several runs to preprocess several input source files, so a different set file can be used for each input file allowing customization. The disadvantage is that to conveniently have the same set of CoCo symbols used to preprocess several input source files, one must prepare a set file and/or an include file to contain the CoCo symbol declarations, and use the same set file for each. This might be done by using the default set file name for all input files. Alternatively, one may use a freeze file to capture symbol declarations and values and read that file as an include file. See the FREEZE directive for more. The advantage of making one output file from several input files is that one set of CoCo symbols may be used to control the preprocessing of all input files, and the compiler may be able to do a better job of optimization if it can see more of the program at one time. The disadvantage is that one may have a symbol present when CoCo processes a subsequent input file which is unneeded and possibly confusing or having a value giving unexpected effects. Of course, an include file may be used to reset a set of integer or logical variables to a desired initial value for each source input file.

Chapter 6

Extensions to Standard CoCo

This implementation supports some extensions to standard CoCo. Extensions to the standard CoCo include editing extensions (Section 6.1), file handling extensions (Section 6.7), and diagnostic extensions (Section 6.8),. These directives may appear in source files. Other extensions (Section 6.9) may appear only in the set file.

6.1 Editing Extensions

A name of an integer, logical, macro, or file variable, when it appears between the argument key characters without embedded blanks, is replaced by the value of the name, as described in the following. Source lines may be edited, CoCo directives are not edited.

6.2 Integers and Logicals as Macros

The string `?name?` is checked to see if name is the name of a CoCo integer or logical constant or variable. If it is, it is replaced by the value of the CoCo integer (as a possibly signed digit string) literal or logical (as ".true." or ".false.") literal. The name may be in either case. The following example illustrates the use of CoCo integers. The repeat count in the character declaration of the format must be a character substring, which is provided by the string substituted in place of ?size?:

```
?? ! set problem size
?? integer :: size = 10
integer, parameter :: nmax = ?size?
```

```
real, dimension( nmax) :: a
character, parameter :: my_fmt = '(?size?es18.9)'
```

6.3 Macros

Macros are another extension. A macro is a name whose value is a string. There may be arguments to the macro, which are strings replaced by the strings used in place of the arguments when the macro is referenced. The macro is used by surrounding its name with the argument key characters, ?name?. If the macro has arguments, the arguments follow the trailing key character thus, ?name?(args). Take care if a macro name appears within another macro's string. A previously declared macro name may not appear in a macro value, subsequent macro names are not checked. This is a basic measure to prevent recursive macro definitions.

The programmer may define macros via the MACRO directive. There are some predefined macros, they are listed in the following table:

6.4 Predefined Macros

The predefined macros may not be redefined, they may be considered to be constants (even if their value changes during execution). These names cannot be used for any purpose, they cannot be dummy argument names to macros or text blocks.

name	definition	set by
file	input file name	input file
line	input file line number	count input lines per file
date	date preprocessing	wall clock
time	time preprocessing	wall clock
coco	coco's RCS Id string	CoCo RCS check-out
setfile	set file name	set file
logfile	log file name	log file
output	output file name	output file
cmdline	CoCo command line	O/S shell
user	user identifier USER	or LOGNAME environment variables
cwd	directory where CoCo is run	PWD environment variable
incpath	directories searched for include files	command line and set file
freeze	set file freeze or -f command line	command line or set file
null	a null string	constant
blank	a blank character	constant

The MACRO directive defines a macro. If a `?name?` is found which is the name of a macro, it is replaced with the value of the macro. Arguments may be present, if so, they are substituted in the value. There is one name space for all symbol names, but macro dummy argument names have a scope of the macro only.

The following table summarizes CoCo substitutions where a symbol's name appears between the argument key characters.

CoCo Symbol	Replacement
Integer	Literal Value
Logical	Literal Value
Macro	String Value
File Variable	File Name

Take care with integers; a negative value could result in two Fortran operators appearing consecutively.

6.5 Text Blocks

The TEXT and COPY directives define a text block (that is, one or more lines), and copy it into the source. A text block may be considered a multi-line macro (note that Fortran distinguishes lines and statements). The text block is defined by the TEXT and END TEXT directives. The COPY directive copies a text block of the same name. Arguments may be present, if so, they are substituted in the value. Conditional compilation applies within text blocks, it is effective during the copy operation. Editing of names also applies during the copy operation. Text blocks must be nested correctly within IF blocks.

6.6 Source File Directives

The following directives edit the source code:

- `?? ASSERT ( condition )`

  This causes code to be written to the output Fortran source to verify that the logical condition is true during program execution, and to halt execution with an error message written to unit= * if it is false. The error message includes the file name and line number where the assert directive was found. The assert directive should be placed only where executable Fortran code is allowed, and the condition should refer only to those Fortran symbols in scope at that location in the source file. CoCo cannot check that this is true.

- `?? MACRO [, PARENS] :: name [ ( arg [, arg ] ... ) ] = string`

This causes subsequent strings of the form ?name? to be replaced by string in the source code. The name must not be the name of an integer, a logical, another macro, a text block, or a file variable. If the (arg[, arg]. . .) is present, this causes subsequent strings of the form ?name?(str[, str]. . .) to be replaced by string in the source code, with strs substituted for the corresponding ?arg? within string. The macro dummy argument list, if present, must not be empty. A macro definition may not appear in the set file, because a macro may not be redefined (the set file overrides existing definitions). The number of actual arguments must match the number of dummy arguments (the number of args must match the number of strs). When PARENS is present, actual arguments are enclosed in parenthesis (if not already) when substituted in the macro string. If PARENS is present, there must be a dummy argument list present as well.

- `?? TEXT [, PARENS] :: name [ ( arg [, arg ] ... ) ]`

  Defines the lines which follow, up to the next END TEXT directive, as being the text block with the given name. The name must not be the name of an integer, a logical, a macro, another text block, or a file variable. This text may be copied into the CoCo output by using the copy directive. The text dummy argument list, if present, must not be empty. See below for information about which directives may appear between the text directive and the matching end text directive. TEXT and COPY directives may appear only in source files. A text block may not be redefined. The number of actual arguments must match the number of dummy arguments (the number of args on the TEXT directive must match the number of strs on the COPY directive). When PARENS is present, actual arguments are enclosed in parenthesis (if not already) when substituted by the COPY directive. If PARENS is present, there must be a dummy argument list present as well. Text blocks may not be nested.

- `?? END TEXT [ name ]`

  Marks the end of the text block which started with the previous TEXT directive. The name, if present, must match the name on the preceding TEXT directive.

- `?? COPY :: name [ ( str [, str ] ... ) ]`

  Copies the text block named by name into the output. If the (arg[, arg]. . .) was present on the text directive, the COPY directive must have the (str[, str]. . .) present with one str for each arg, and causes each strs substituted for the corresponding ?arg? within the text block. The number of actual arguments must match the number of dummy arguments (the number of args on the TEXT directive must match the number of strs on the COPY directive). A COPY directive may not appear within a text block.

- `?? GETENV :: name = string`

  This causes an environment variable named string to be sought. If found, a macro is defined with name and a value of the value of the environment variable. That is, the GETENV declaration is the same as ?? MACRO :: name = $string (using the usual shell convention). It is an error if the environment variable is not found.

There is one name space for all integer names, logical names, macro names, text block names, and file variable names. A dummy argument on one macro or text block may have the same name as a dummy argument on another macro or text block, but must not be the same as an integer, logical, macro, text block, or file variable. This also avoids ambiguity when expanding macros and copying text blocks. Note that dummy arguments are substituted within a macro value or a text block, but not within CoCo directives contained within text blocks. Therefore, conditional compilation of text blocks during copying can be a function of variables, but not actual arguments. A string enclosed in parenthesis or brackets (for example, array constructors) is treated as a unit when used as an actual argument.

Only the following directives may appear between the TEXT and END TEXT directives: the ASSERT, IF, ELSE IF, ELSE, END IF, MESSAGE, STOP and assignment directives. While a text block may appear in an include file, an INCLUDE directive may not appear within a text block. Declaration directives may not appear because each declaration may only occur once per program. Text blocks may not be nested.

An example of a macro declaration and use follows. Note the lack of parenthesis around the actual arguments (the parentheses are added by the PARENS on the macro statement), and the lack of space between the macro name and opening parenthesis of the actual argument list:

```
?? ! used to compute radius
?? macro, parens :: hypot( x, y) = sqrt( ?x?*?x?+?y?*?y?)
?? ! r1, r2, s1, s2 are variables
! r = sqrt( (r2-r1)*(r2-r1) + (s2-s1)*(s2-s1) )
r = ?hypot?( r2-r1, s2-s1)
```

An example of a text block follows:

```
?? ! define stack operations as a text block
?? ! with 'type' as an argument

?? text :: stackops( type)
subroutine push_?type?( item)
type( ?type?_t), intent( in) :: item
?type?_stack( next_?type?) = item
next_?type? = next_?type? + 1
return
end subroutine push_?type?
```

```
subroutine pop_?type?( item)
type( ?type?_t), intent( out) :: item
next_?type? = next_?type? - 1
item = ?type?_stack( next_?type?)
return
end subroutine pop_?type?
?? end text stackops

?? ! write stack ops for type fermion_t
?? copy :: stackops( fermion)

?? ! write stack ops for type boson_t
?? copy :: stackops( boson)
```

An interface block could be provided to allow the push and pop routines to be called with a generic name, if desired. The stack arrays and stack indexes must be declared outside the routines, as defined here.

6.7 File Handling Extensions

These directives allow some limited control over files.

- `?? ENDFILE`

  Causes CoCo to end processing of the current input file. The endfile directive may not be continued onto subsequent lines. This might be used, for example, if all lines containing interface blocks have been processed and the procedure bodies are not to appear in the output.

- `?? FREEZE [ 'filename' ]`

  Causes CoCo to disallow further declarations during the run. If present, the filename names a file to receive declarations for all symbols currently declared, and their current value if they are defined. The filename may be used as an include file in subsequent runs, or as a freeze file on a `-f` command line option (see Section 4.2) or a freeze set file directive. The file is created, its previous contents are lost. Note that the freeze directive behaves differently when present in the set file. See also the `-f` command line option.

- `?? OPEN :: name = 'filename'`

  Causes CoCo to open the named file for output, and define the name as a file variable that refers to the file.

- `?? OUTPUT [ : name ]`

  Causes CoCo to write subsequent lines to the file indicated by the named file variable. If the file variable is missing, output reverts to the output file. The lines are treated as inactive lines in the original output file (and therefore controlled by the alter state).

In the example that follows, a module contains several functions. A separate file is to contain a procedure with tests, one for each function. The OPEN and OUTPUT directives solve this as follows:

```
module functions
contains

?? ! open the file for output
?? open :: t = 'testfunctions.f90'

?? ! write to testfunctions.f90 via the file variable t
?? ! this might be visible in the regular output file,
?? ! depending upon the alter mode
?? output: t
subroutine test_functions()
! declarations
?? ! resume writing to the regular output file
?? output

?? ! this is the test reference wanted for each function
?? text :: call( cond)
call testit( ?cond?)
?? end text call

?? ! add the test for first
?? output: t
?? copy :: call( first( x) == x+1 )
?? output

function first( x)
! rest of first
end function first

?? ! add the test for second
?? output: t
?? copy :: call( second( x) == x-1 )
?? output

function second( x)
! rest of second
end function second

!  other functions
```

```
?? ! finish the test procedure
?? output: t
end subroutine test_functions
?? output

end module functions
```

When the file containing the module is preprocessed, the file containing the test procedure is automatically written. It may be easier to keep the tests synchronized with the functions this way. The alternate output file need not be Fortran, of course. Any documentation, or even mathematics, perhaps written in LaTeX, might be considered for similar treatment.

6.8 Diagnostic Extensions

These directives allow the programmer to monitor and debug the processing of source files:

- '?? CMDLINE

  Causes CoCo to print the command line of its invocation to the logfile, or to stderr. The command line does not change during processing. This may be useful capturing values set by -D options (see Section 4.2), or other options that override those from the set file.

- `?? DOCUMENT`

  Causes CoCo to act as if the following were encountered in the input file.

```
    !
    ! Preprocessor executed: ?date? ?time?
    !
    ! Preprocessor command line: ?cmdline?
    ! Preprocessor set file: ?setfile?
    ! Preprocessor log file: ?logfile?
    ! Preprocessor version: ?coco?
    !
    ! Source file: ?file? line: ?line?
    ! Compile file: ?output?
    ! Include path: ?incpath?
    ! Freeze file: ?freeze?
    !
    ! User: ?user?
    ! Current directory: ?cwd?
    !
```

This has the effect of documenting the particulars of the preprocessing. (If this were in an include file, the ?file? and ?line? would refer to the include file, which might not be the desired information.)

- `?? OPTIONS`

  Causes CoCo to print the currently in effect options to the logfile, or to stderr. Since options may be set only via the command line or the set file, the options report does not change during processing. This may be useful capturing values set by a set file (see Chapter 3) or command line, and thus not defined within the source files proper.

- `?? REPORT`

  Causes CoCo to print its end of processing report to the logfile, or to stderr, with data current as of the position where the directive is encountered. Compare with the set file REPORT directive, which causes the report to be written at the end of processing.

- `?? SYMBOLS`

  Causes CoCo to write to the logfile, or to stderr, a summary of all symbols (integers, logicals, macros, text blocks, and file variables) known at the point where the directive is encountered. If an integer or logical variable has no value when the SYMBOLS directive is encountered, the value printed is “”. The value of a macro is the replacement string without argument substitution, if any. The value of a text block is printed line-by-line as if it were a multi-line macro, and may contain CoCo directives such as if blocks. File variables have the name of the file printed.

If CoCo is being used along with the env2inc program to capture the processor-supported kinds, the **fortran_env.inc** file can be used to define CoCo symbols for the compiler used to compile the env2inc program. The env2inc program must be compiled by the same compiler as will be used to compile the program to be preprocessed, including any compiler options affecting the available kinds (for example, -r8), otherwise, unexpected, suboptimal, or incorrect results may occur. The fortran_env.inc file includes the results of the compiler_version and the compiler_options intrinsic module procedures to allow checking that this is so.

6.9 Set File Extensions

Some extension directives may appear only in the set file, and are intended to allow the programmer to control modes of CoCo otherwise controlled from the command line (for example, in case the program was compiled without the f2kcli module, or other access to the command line). An option on the command line generally overrides the corresponding directive in the set file. These directives are listed below:

- `?? DIRECTORY 'directory-name'`

Causes CoCo to search directory-name when an include file is not found in the current directory. Several DIRECTORY directives may each specify a directory-name, they are searched in the order they are declared. A DIRECTORY directive must appear in the set file (this is so all references to an include file name in one preprocessing run refer to the same file). See also the `-I` command line option (Section 4.2).

- `?? FORM: [ FREE | FIXED ]`

  Causes CoCo to treat the input source files as free form source files or as fixed form source files. By default, CoCo assumes input files are free form source. FIXED changes CoCo to assume fixed form source files. FREE is allowed to confirm the default choice. Only one FORM directive may occur in a set file. See also the `-F` command line option.

- `?? FREEZE 'file-name'`

  Names a freeze file. Freeze files contain declarations only. This directive also prohibits declarations from appearing in any input file. See also Chapter 4. A freeze directive is ignored when a `-f` command line option is present. Note that a freeze directive behaves differently when present in a source file.

- `?? INPUT 'file-name'`

  Names an input file. Several input directives may occur within a set file. Files are processed in the order the directives are encountered. See also Chapter 4. Input directives are ignored when input file names appear on the command line.

- `?? LOGFILE 'file-name'`

  The logfile directive is set to the named file. The file is created. Only one LOGFILE directive may occur in a set file. See also the `-l` command line option.

- `?? MARK: [ ON | OFF ]`

  Turns on of off the placing a line in the output to mark subsequent input source files. Only one MARK directive may occur in a set file. See also the `-m` command line option.

- `?? NUMBER: [ ON | OFF ]`

  Turns on of off the numbering of source lines. If on, source lines appearing in the output as active source lines have the input file name and line number appearing as a Fortran comment starting at the wrap column. Only one NUMBER directive may occur in a set file. See also the `-n` command line option and Section 5.3 on line wrapping and numbering.

- `?? OUTPUT 'file-name'`

Names the output file. The file is created. Only one OUTPUT directive may occur in a set file. See also Chapter 4. Output directives are ignored when an output file name appears on the command line.

- `?? POST: [ ON | OFF ]`

  Causes CoCo to copy the set file to the end of the output file at the completion of processing if ON is selected (the default, as specified by the standard). If OFF is specified, the set file is not copied to the end of the output file, nor is the standard-specified separating line present. Only one POST directive may occur in a set file. See the `-p` command line option.

- `?? REPORT: [ ON | OFF ]`

  Causes CoCo to write a summary report at the end of processing if ON is specified, or if OFF is specified, the default, the report is not written. Only one REPORT directive may occur in a set file. See the `-r` command line option and the REPORT directive that appears in a source file.

- `?? VERBOSE: [ ON | OFF ]`

  Turns on of off the reporting of file openings and closings. Unless verbose mode is set on the command line, the opening of the set file cannot be reported this way, because the set file is already being read when this directive is executed. Only one VERBOSE directive may occur in a set file. See also the `-v` command line option.

- `?? WRAP: [ ON | OFF ]`

  Turns on of off the wrapping of source lines. If on, source lines appearing in the output as active source lines are wrapped starting at the wrap column. Only one WRAP directive may occur in a set file. See also the `-w` command line option and Section 5.3 on line wrapping and numbering.

- `?? WARNING: [ ON | OFF ]`

  Turns on of off the issuing of warning messages. If on, CoCo will warn about a few patterns that may be errors. See also the `-W` command line option.

Since the set file is copied to the end of the output file, a record may be kept within the source of the CoCo options used to generate the output file. If the options specified in the set file are overridden by command line options, the command line could be present in the output file via the ?cmdline? predefined macro or a ?? CMDLINE directive.

Chapter 7

An Example of Using CoCo

Statement of the problem to be solved: A single source file is to be prepared which will specify a Fortran module containing a cube root function to support all real kinds on any processor at a computing center. Using the program env2inc (compiled and executed separately for each compiler at the computer center, see Chapter 10), to provide kind parameters in separate fortran_env.inc include files, this module may be written as follows:

```
?? ! define the coco integer constants
?? ! real32, real64, real128 (among others)
?? include 'fortran_env.inc'

module cube_root

!  import Fortran kind parameters
!  real32, real64, real128
use, intrinsic :: iso_fortran_env, only: real32, real64, real128

implicit none

private

interface cbrt
?? if( real32 > 0 )then
   module procedure real32_cbrt
?? endif
?? if( real64 > 0 )then
   module procedure real64_cbrt
?? endif
?? if( real128 > 0 )then
   module procedure real128_cbrt
```

```
?? endif
end interface

public :: cbrt

contains

?? if( real32 > 0 )then
elemental real( kind= real32) function real32_cbrt( x)
real( kind= real32), intent( in) :: x

   real32_cbrt = sign( exp( log( abs( x)) / 3.0_real32), x)

end function real32_cbrt
?? endif

?? if( real64 > 0 )then
elemental real( kind= real64) function real64_cbrt( x)
real( kind= real64), intent( in) :: x

   real64_cbrt = sign( exp( log( abs( x)) / 3.0_real64), x)

end function real64_cbrt
?? endif

?? if( real128 > 0 )then
elemental real( kind= real128) function real128_cbrt( x)
real( kind= real128), intent( in) :: x

   real128_cbrt = sign( exp( log( abs( x)) / 3.0_real128), x)

end function real128_cbrt
?? endif

end module cube_root
```

Note that the real32, real64, and real128 which appear in the Fortran source proper are the three kind parameters which are defined in the intrinsic module iso_fortran_env. The real32, real64 and real128 appearing in the CoCo IF directives are CoCo integer constants which are defined in the fortran_env.inc CoCo include file. They have the same values as the Fortran named constants but are available to CoCo to control the preprocessing of the source file. The fortran_env.inc is made automatically by the env2inc program (see Chapter 10). All the Fortran kind parameters actually appearing in the output source file are valid on the processor, because if the kind isn't supported, the corresponding

CoCo integer constant is set to a negative value, thereby preventing the code from being present in the version of the source for that processor. The relational expression in the IF-statements could be assigned to logical constants if desired.

Solving the same problem, but this time using the text-copy mechanism, is shown below:

```
?? ! define the coco logical symbols
?? ! real32, real64, real128 (among others)
?? include 'fortran_env.inc'

module cube_root

?? text :: cbrt( kind)
elemental real( kind= ?kind?) function ?kind?_cbrt( x)
real( kind= ?kind?), intent( in) :: x

   ?kind?_cbrt = sign( exp( log( abs( x)) / 3.0_?kind?), x)

end function ?kind?_cbrt
?? end text cbrt

!  define Fortran kind parameters
!  real32, real64, real128
use, intrinsic :: iso_fortran_env, only: real32, real64, real128

implicit none

private

! define the generic name cbrt
interface cbrt
?? if( real32 > 0 )then
   module procedure real32_cbrt
?? endif
?? if( real64 > 0 )then
   module procedure real64_cbrt
?? endif
?? if( real128 > 0 )then
   module procedure real128_cbrt
?? endif
end interface

public :: cbrt

contains
```

```
?? if( real32 > 0 )then
?? copy :: cbrt( real32)
?? endif

?? if( real64 > 0 )then
?? copy :: cbrt( real64)
?? endif

?? if( real128 > 0 )then
?? copy :: cbrt( real128)
?? endif

end module cube_root
```

Note that this time, the source for the cbrt() function need be specified only once, the text-copy mechanism performs the copy-paste-substitute operation as needed.

One advantage of using the values of processor kind values as CoCo integers is that the value may be used within the program (perhaps in a Fortran comment). Below, a CoCo logical constant is made from one of the integer values found in the fortran_env.inc include file. Of course, this may be done with any of them, if this style is desired.

```
?? logical, parameter :: has_real128 = real128 > 0
```

Chapter 8

An Example Using a *freeze-file*

This example involves two executions of CoCo, the first to prepare the freeze file, the second to use it. The first execution reads an input file with no source lines. The input file simply includes the fortran_env.inc file, which may be modified by command line options and/or set file directives. Other declarations and processing logic may be present as well, but for simplicity, very little is shown.

```
?? ! prepare the freeze-file

?? ! fortran_env.inc was made by env2inc
?? ! env2inc must have been compiled by the same compiler
?? ! to be used to compile this program !!

?? include 'fortran_env.inc'

?? ! what to do when no 128-bit real

?? if( real128 <= 0 )then
?? !  setup for life without real128
?? end if

?? ! debugging is enabled via the command line or set file
?? ! the actual value of debug at the time of the freeze statement
?? ! will be written to the freeze-file configure.inc

?? logical, parameter :: debug = .false.
?? freeze 'configure.inc'
```

In the second execution of CoCo, the source file intended to be compiled is preprocessed. It merely includes the freeze file, containing the declaration of all symbols needed. Using this technique, CoCo may taylor one configuration of definitions for use in many source files.

```
?? ! use the prepared freeze-file

?? include 'configure.inc'

?? ! no more coco variables are needed
?? freeze

?? ! the rest of the program to be compiled follows
```

The FREEZE directive in the second file does not write a freeze file, it only serves to prevent further declarations. Auxiliary variables must be declared before this directive appears. A FREEZE directive appearing in the set file would not allow further declarations in any input source file.

Chapter 9

Some Known Bugs and Limitations

CoCo does not read Fortran source beyond the minimum necessary for macro replacement. This is done both for efficiency, and to increase CoCo's reliability by allowing a simpler design. However, some limitations are a result of this choice.

During source line editing, CoCo makes several passes through its list of macro names for each line to be edited. A symbol to be replaced by its value must appear entirely on one line, the name cannot be broken across a continuation line. CoCo checks that a macro value does not contain a reference to a previously-declared macro (this tries to prevents recursion). Thus, a macro may appear in the value of another macro if the macro to appear is declared later than the macro in whose value it is to appear. One may conditionally define a macro to have the desired macro value. The integers, logical, and predefined macros are replaced after the explicitly declared macros. When editing a line causes the line to become too long, it is broken into several lines and each new line is output. This may cause line numbers to disagree between the input file and the output file, if so, enabling line numbering may help trace compiler error messages back to the original source line. Better, when tracing a compiler error message read the file compiled rather than the file input to the preprocessor.

The actual arguments supplied to macro expansion and text block copying should contain either no parenthesis, or balanced parenthesis. CoCo attempts to skip commas intended as part of format specifications, array subscript sets, and procedure argument lists by skipping commas appearing between parentheses. Unbalanced parentheses within actual arguments can foil this process.

CoCo attempts to glean the directory separation character by scanning the value of the `?cwd?` macro, which is gotten from the PWD environment variable. If this fails, the directory separation character must be appended to directory names specified on `?? DIRECTORY` set file directives and on `-I` command line options. If a simple directory separator is inappropriate (for example, with VMS), the directory names must be specified so that simply appending the include file name produces a value that will allow the include file to be used.

CoCo accesses its command line via the usual methods. Since file indirection is ordinarily handled by the shell as part of program startup processing, CoCo cannot distinguish the case where files have been redirected to stdin and stdout from the case where there are no source files on the command line. Therefore, CoCo will honor input and output directives in its set file even in the case where files have been redirected from the command line. Care should be taken to use a set file without input or output directives when file indirection is intended, and, for clarity, to shun file indirection when input or output set file directives are intended. To document CoCo's idea of its command line, see the ?? CMDLINE directive, the ?cmdline? predefined macro, and the the ?? DOCUMENT directive. Also, if there are no input or output file names on the command line (either because stdin and stdout are redirected, or because ?? INPUT and ?? OUTPUT set file directives are being used, and there is no set file named on the command line (see Section 4.2), the only set file that can be read is coco.set, the default.

If verbose mode is on, and a log file is named, the closing of the log file is reported on stderr, because after the log file is closed the log file can no longer receive comments. Likewise, if verbose mode is switched on in the set file, the opening of the set file will not be reported, unless verbose mode is also switched on by the command line.

As with all CoCo statements and inactive source lines, the set file, its standard-defined separation line, and lines marking subsequent input files are invisible when the alter state is set to delete or blank. If you want to see these lines, set the alter state to one of the shift values.

Chapter 10

Downloads

To download the source code for the CoCo program, click CoCo preprocessor. As CoCo has grown in size and complexity, it is now desirable to maintain CoCo as several modules rather than one large program. Thus, CoCo is now a gzipped tarfile containing the source files and a makefile. The makefile is set to use gfortran as the compiler, but also has macros for ifort and nagfor. You should be able to configure it for other compilers as well. You may want to use the free F2KCLI Module from I.S.S. Ltd. to compile the CoCo program if your compiler doesn't support the Fortran 2003 command line access intrinsic procedures. Download the restore program which can undo CoCo processing under limited circumstances. The env2inc program is also available.

For more information about the `fortran_env.inc` file, click here (link does not work).

Chapter 11

Using CoCo with NONMEM

The main utility of a preprocessor for NM-TRAN control files is that various versions may exist with only one master file where changes are made. The various versions may differ only in (possibly important) details. It often happens that when a modification is desired in a common part of the code, all existing versions must be edited. Such a modification may be a change of the items in `$INPUT` or `$TABLE`, something simple as the `$PROBLEM` name, or some typo or mistake.

Also, different modeling options may be evaluated, which result in some NM-TRAN input lines commented out in various combinations in different control files. With a preprocessor, these options may be selected using IF statements.

The Fortran preprocessor or conditional compilation utility CoCo has been adapted to work with NONMEM. Fortunately, because the NM-TRAN input resembles Fortran, this required only some minor changes.

The present chapter illustrates how many CoCo directives may help in writing one master NONMEM control file. The chapter was added to the (mostly unmodified) original CoCo documentation for easy referencing to the relevant sections using hyperlinks.

The following changes were made to the original CoCo program:

- The comment character was changed from '!' to ';' as is required by NM-TRAN.
- The default input file extension was changed from ".fpp" to ".nmt" and the output file extension from ".f90" (or ".f") to ".ctl" (not to ".mod" for PsN because that file may contain additional information; however, see

below at Section 11.3). These defaults may be overridden by giving both the complete input and output file names on the command line.

- The code lines generated by the ?? ASSERT directive are now preceded with the double quotation (‘"’) character for verbatim code.
- Finally, two extensions were added:
  - the predefined macro `?EVAL?` (see Section 11.1.6)
  - the command line option `-U name` (see Section 11.1.8)

11.1 Example Snippets

Various examples using directives, macros, and command-line options follow.

11.1.1 ?? DOCUMENT

First of all, the ?? DOCUMENT directive is very useful to document within the control file how it was generated:

```
../src/coco -a1 -p -DDRUG=1 < ../examples/document.nmt
```

```
;?? DOCUMENT
;
; Preprocessor executed: 2023/08/12 23:57:13.682
;
; Preprocessor command line: ../src/coco -a1 -p -DDRUG=1
; Preprocessor set file:
; Preprocessor log file:
; Preprocessor version: $Id: coco.f90,v 2.11 for NM-TRAN 2023/08/09 14:00:00 dan/erik
;
; Source file: <stdin> line: 1
; Compile file:
; Include path: .
; Freeze file:
;
; User: olofsen
; Current directory: /usr2/cocononmem/book
;
STOP coco normal exit
```

At one of the first output lines, the options at CoCo’s command line are shown for reference.

For the purposes of this document, option `-a1` was used to show all lines, possibly commented out, and `-p` to hide information about a SET file (see Section 4.2).

`?? DOCUMENT` implicitly uses many of CoCo’s predefined macros.

11.1.2 ?? IF

When modeling two drugs in parallel, one obvious difference in the control file is the name provided with `$PROBLEM`. It is easy to forget to change it when copying and editing an earlier control file. The ?? IF directive may be used to select which name should be given to the problem. At this point, it may also be notified that something is not meaningful by using ?? MESSAGE and ?? STOP.

```
../src/coco -a1 -p -DDRUG=1 < ../examples/problem.nmt
```

```
;?? INTEGER :: DRUG

;?? IF (DRUG.EQ.1) THEN
$PROBLEM DRUG1
;?? ELSE IF (DRUG.EQ.2) THEN
;$PROBLEM DRUG2
;?? ELSE
;?? MESSAGE 'DRUG should be 1 or 2'
;?? STOP
;?? ENDIF
STOP coco normal exit
```

Because the integer DRUG does not have an initial value, it must be defined by `-DDRUG` on the command line (or in a SET file).

While the name of the `$PROBLEM` may not be that important or it may be given a more generic one, one main difference between the desired control files is the data file used with `$DATA`:

```
../src/coco -a1 -p -DDRUG=2 < ../examples/datafile.nmt
```

```
;?? INTEGER :: DRUG

;?? IF (DRUG.EQ.1) THEN
;$DATA drug1.csv
;?? ELSE
$DATA drug2.csv
;?? ENDIF
STOP coco normal exit
```

If all data are in one file and `ACCEPT` or `IGNORE` is used, their parameters may be selected with `?? IF` as well, or by using `$DATA ... ACCEPT=(DRUG.EQ.?DRUG?)`, where `DRUG` is a column name in the data file as mentioned in `$INPUT`.

11.1.3 ?? GETENV

String variables are not available in CoCo, but MACROs have names which can be used. With ?? GETENV a macro may be obtained from the environment, such as merely a file name:

```
DATAFILE=drug.csv ../src/coco -a1 -p < ../examples/getenv.nmt
```

```
;?? GETENV :: DATAFILE = DATAFILE
;?? ; now using $DATA ?DATAFILE?
$DATA drug.csv
STOP coco normal exit
```

11.1.4 ?? INCLUDE

The ?? INCLUDE may be useful for control files that differ essentially between versions, like the specifications for `$OMEGA` blocks, where different runs need for example 0 FIXes. Otherwise, the master control file gets too cluttered with IF statements.

11.1.5 ?? MACRO

Often, with MU modeling and interoccasion variability, a model parameter is written as follows

```
V1 = EXP(THETA(1)+ETA(1))*EXP(EOCC1)
```

Time consuming correction of typos may be avoided by using ?? MACRO directives:

```
../src/coco -a1 -p < ../examples/macro.nmt
```

```
;?? MACRO :: REPLETA(I,J) = $ABBR REPLACE ETA(OCC_P?I?)=ETA(,?J? to ?EVAL?(?J?+3))
;?? MACRO :: MPAR(I) = EXP(MU_?I?+ETA(?I?))*EXP(EOCC?I?)
;?? ; now using ?REPLETA?(1,9)
$ABBR REPLACE ETA(OCC_P1)=ETA(,9 to 12)
EOCC1 = ETA(OCC_P1)
;?? ; now using ?MPAR(1)?
V1 = EXP(MU_1+ETA(1))*EXP(EOCC1)
STOP coco normal exit
```

Here EOCC1 is a variable that may also be used as an item in `$TABLE`. With `?? TEXT` (see below), it would be possible to generate both the `$ABBR` and the `EOCC1 = ETA(OCC_P1)` lines, but these are needed at separate occasions in the control file.

11.1.6 ?EVAL? predefined macro

In the previous example, `?EVAL?(?J?+3)` was used to actually calculate `9+3 = 12`, because otherwise the macro expansion would give `9+3` which NM-TRAN does not accept. It would, however, be possible to do this with the ?? TEXT directive and an intermediate variable I and calculation `?? I = I+3` statement which also allows for expression evaluation. `?EVAL?` handles only integer expressions.

11.1.7 ?? TEXT

?? TEXT is like a multi-line macro; for an example see below at section 11.3.

11.1.8 The -U command line option

It may be logical to define a LOGICAL with default value .TRUE.. However, with the `-D` command line option a logical may only be defined to be true and the original CoCo provided no way to set it to .FALSE. (except by using a SET file). Therefore, the `-U` option was added to the CoCo program, as a way to unset the variable.

```
../src/coco -a1 -p -UTEST < ../examples/undefine.nmt
```

```
;?? LOGICAL :: TEST = .TRUE.
;?? IF (TEST) THEN
;" PRINT *, 'TRUE'
;?? ELSE
" PRINT *, 'FALSE'
;?? ENDIF
STOP coco normal exit
```

11.2 A Complete Example

`CONTROL5` with the theophylline data set is the “hello world” of NONMEM. The following adaptation shows the possibility of testing various modeling choices.

```
?? DOCUMENT
?? INTEGER :: RUNNO = 1
?? LOGICAL :: OBSATZERO = .TRUE.
?? LOGICAL :: LOGNORM = .FALSE.
?? LOGICAL :: PROPERR = .FALSE.
?? LOGICAL :: DIAGOMEGA = .FALSE.
?? LOGICAL :: FOCE = .FALSE.
?? LOGICAL :: TABLE = .FALSE.

?? IF (RUNNO==2) THEN
?? PROPERR = .TRUE.
?? OBSATZERO = .FALSE.
?? ENDIF

$PROB THEOPHYLLINE POPULATION DATA

$INPUT ID DOSE=AMT TIME CP=DV WT

?? IF (OBSATZERO) THEN
$DATA THEOPP
```

```
?? ELSE
$DATA THEOPP ACCEPT=(AMT>0,TIME>0)
?? ENDIF

$SUBROUTINES ADVAN2

$PK
 CALLFL = 1
?? IF (LOGNORM) THEN
 KA = THETA(1)*EXP(ETA(1))
 K = THETA(2)*EXP(ETA(2))
 CL = THETA(3)*WT*EXP(ETA(3))
?? ELSE
 KA = THETA(1)+ETA(1)
 K = THETA(2)+ETA(2)
 CL = THETA(3)*WT+ETA(3)
?? ENDIF
 SC = CL/K/WT

$ERROR
?? IF (PROPERR) THEN
 Y=F*(1+EPS(1))
?? ELSE
 Y=F+EPS(1)
?? ENDIF

$THETA
 (.1,3,5)
 (.008,.08,.5)
 (.004,.04,.9)

?? IF (DIAGOMEGA) THEN
$OMEGA
 6
 0.0002
 .4
?? ELSE
$OMEGA BLOCK(3)
 6
 .005 .0002
 .3 .006 .4
?? ENDIF

$SIGMA
 0.4
```

```
$EST
?? IF (FOCE) THEN
?? IF (PROPERR) THEN
 METHOD=1 INTERACTION
?? ELSE
 METHOD=1
?? ENDIF
?? ENDIF
 MAXEVAL=9999 PRINT=5

?? IF (TABLE) THEN
$TABLE ID AMT TIME DV WT
?? ENDIF
```

Running coco without any `-D` command line options, results in an output that gives the same results as when using the original `CONTROL5`. Using `-DPROPERR` will result in an early stop of NONMEM because of zero concentrations. These may be excluded by using `-UOBSATZERO` as well.

The possibilities that will be explored at different run numbers might be predefined by selecting the options depending on a `RUNNO` integer.

11.3 PsN

CoCo may help PsN in various ways, for example:

- `?? INCLUDE` : An advantage is that with ?? INCLUDE the included file is, well, included, so the resulting control file may be used with PsN, which as of date does not handle NONMEM's `$INCLUDE` record.
- `?? IF` to modify a source file for some utilities that won't accept the control file which was used for the fit, or to omit the `$COV` and/or `$TABLE` steps when these are not needed and take computing time and disk space.
- To generate the information for the `runrecord` utility.

```
../src/coco -a1 -p -UTEST < ../examples/runrecord.nmt
```

```
;?? TEXT :: RUNRECORD(BASEDON,DESCR,LABEL,STRUCTMOD,COVMOD,IIV,IOV,RV,EST)
;;; 1. Based on: ?BASEDON?
;;; 2. Description:
;;; ?DESCR?
;;; 3. Label:
;;; ?LABEL?
;;; 4. Structural model:
;;; ?STRUCTMOD?
;;; 5. Covariate model:
```

```
;;; ?COVMOD?
;;; 6. Inter-individual variability:
;;; ?IIV?
;;; 7. Inter-occasion variability:
;;; ?IOV?
;;; 8. Residual variability:
;;; ?RV?
;;; 9. Estimation:
;;; ?EST?
;?? END TEXT

;?? COPY :: RUNRECORD(,initial,base,2abs1disp,none,diagonal,diagonal,propadd,FOCE)
;?? ! text runrecord
;; 1. Based on:
;; 2. Description:
;; initial
;; 3. Label:
;; base
;; 4. Structural model:
;; 2abs1disp
;; 5. Covariate model:
;; none
;; 6. Inter-individual variability:
;; diagonal
;; 7. Inter-occasion variability:
;; diagonal
;; 8. Residual variability:
;; propadd
;; 9. Estimation:
;; FOCE
;?? ! end text runrecord
STOP coco normal exit
```

11.4 Issues

- There is no ?EVAL? for logical expressions.
- Fixed form options for older versions of Fortran and so possibly for older versions of NONMEM have not been tested.
- Line wrapping has not been tested.

Chapter 12

List of Directives and Macros

Because the description of the available directives and macros is scattered across the earlier chapters, an index is given here.

12.1 Directives

- ?? NUMBER (Set file extension)
- ?? OPEN (File handling extension)
- ?? OPTIONS (Diagnostic extension)
- ?? OUTPUT (File handling extension), ?? OUTPUT (Set file extension)
- ?? POST (Set file extension)
- ?? REPORT (Diagnostic extension), ?? REPORT (Set file extension)
- ?? STOP (Standard CoCo)
- ?? SYMBOLS (Diagnostic extension)
- ?? TEXT (Extension)
- ?? VERBOSE (Set file extension)
- ?? WARNING (Set file extension)
- ?? WRAP (Set file extension)

12.2 Macros

- Predefined Macros
- New: ?EVAL? predefined macro