SWAN

PROGRAMMING RULES

Version 1.3

SWAN PROGRAMMING RULES

by	:	The SWAN team
mail address	:	Delft University of Technology Faculty of Civil Engineering and Geosciences Environmental Fluid Mechanics Section P.O. Box 5048 2600 GA Delft The Netherlands
e-mail	:	swan-info-citg@tudelft.nl
home page	:	*http://www.fluidmechanics.tudelft.nl/swan/index.htm*

Contents

Chapter 1

Introduction

This document describes the programming rules for developing shallow water wave model SWAN. It is essential for all programmers to strictly observe the rules set out in this document. The programming rules should be suitable to support the programmer to make reliable, stable, readable and structured source code. It is furthermore recommended that the adjacent programs also comply with these rules, as this results in a considerable simplification of maintenance, management and error tracking in particular.

Chapter 2 discusses several general FORTRAN 90 standards, while programming of FORTRAN control statements is dealt with in Chapter 3. Use of modules is outlined in Chapter 4. The layouts of subroutines and modules are the subject of Chapter 5, and Chapter 6 discusses input and output. Instructions concerning errors are covered in Chapter 7. The pseudo code mentioned in Chapter 5, is further discussed in Chapter 8. Chapter 9 gives some advices on the improvement of the code, while Chapter 10 briefly considers machine dependency. Chapter 11 describes exceptions to the FORTRAN 90 standards which goes against programming ethics, but is nevertheless acceptable in certain prescribed cases because it strongly improves user-friendliness. Chapter 12 mentions several agreements on defining names for SWAN subroutines and modules. Chapter 13 provides examples of a subroutine layout and module layout, written according to the above-mentioned standards.

This document has no final status, but will be developed and modified as experience and insight are being gained. Please refer to the Log sheet for a summary of this document's modifications.

The document uses different fonts for plain text and FORTRAN text. The latter is in a sanserif `typewriter` font.

Chapter 2

FORTRAN 90 standards

When programming in any program language it is important to avoid obsolete, undesired and unnecssarily complicated constructions, to have well documented codes and codes with a well-defined layout to support easy understanding and desk checking. By satisfying the proposed rules as outlined below, the code will be easier to transfer to other programmers. Rules will help the programmer to develop and maintain the source code.

The following rules should be consider for developing and extending the SWAN source code.

- Code needs to be written in accordance with ANSI FORTRAN 90 standards. Good texts on programming in FORTRAN 90 are [1] and [2].
- Modular programming is strongly recommended.
- The program code is subdivided into program units − or segments − of a maximum of 500 source statements.
- Only one statement per line is allowed.
- The first line of every program unit is a `PROGRAM`, `SUBROUTINE`, `FUNCTION` or `MODULE`. The `PROGRAM` statement is obligatory for a main program, and is followed by the comments as described in Chapter 5.
- Every new program unit needs to be written in free format. Fixed format in the existing program units is acceptable for the time being. Note that an include file in free format cannot be included in a fixed form subroutine.
- New program units are given the '.f90' file extension and existing program units retain the '.f' extension (Linux/Unix) or the '.for' extension (Windows/DOS).
- An exclamation mark (!) is used in the 1^{st} column to indicate a comment line.
- It is preferably to use columns 1 to 5 for labels and columns 7 to 72 for statements (cf. FORTRAN 77).

- Every line contains a maximum of 80 characters. Insert an ampersand (&) as a follow-on mark in the 73rd column, and on the following line in the 6th column.

- Parameter transfer between subroutines takes place preferably via a parameter list and otherwise via modules. Common blocks are not acceptable.

- All variables in a program unit need to be declared explicitly and be given the `INTENT` attribute − with the exception of pointers. The first declaration always needs to be the mandatory statement `IMPLICIT NONE`. The implicit FORTRAN standard is still used for the purpose of clarity, i.e. integer names start with i up to and including n, complex variables with c or z, and all remaining letters are reserved for reals.

- Each variable must be declared in a separate statement.

- Always use the sign :: in the declaration list.

- Use the $*$ option as little as possible for array declaration, and define the actual length by means of parameters wherever possible. This enables good functioning of array bound checkers.

- Variable names have to be distinguishable in the first few characters.

- The meanings of names only apply within a program. They need to be as long as possible and have a logical structure, i.e. they should indicate their functions, preferably in English.

- Variables need to be given in alphabetical order.

- Use the type declaration instead of the dimension statement to declare local variables:

  ```
  INTEGER IARRAY (100)
  ```

  instead of

  ```
  DIMENSION IARRAY (100).
  ```

  Never use real expressions in array dimensions.

- Mixed-mode expressions (single-precision/double-precision) are not used.

- Always initialize pointers with the `NULLIFY` instruction.

- Use as many *allocatable* arrays as possible, and as little *automatic* arrays or fixed-length arrays as possible. The reason is that automatic arrays are placed on the stack, which has a limited size, so that large arrays can easily cause *stack overflow.* By contrast, allocatable arrays are always stored in the heap memory.

- If allocatable arrays are no longer used, they must always be deallocated at the end of the subroutine.

- For validity checks, preference is given to the `STAT=` attribute for the purpose of array allocation/deallocation.
- Use `POINTER` and `TARGET` for very large data sets or linked lists for which the length is not known beforehand.
- Always end a do loop with `END DO` instead of `CONTINUE`. In the case of nested do loops, every do loop must end with its own `END DO` statement.
- Use labels to identify the structure of do loops since, this will improve the readability of the code.
- A combination of upper case and lower case may improve readability, but is not a standard for fixed-format (cf. FORTRAN 77). In general small letters are recommended.
- Make ample use of spaces, comment blocks and extra space between lines.
- Use multi-dimensional arrays where they match the logical structure, unless this leads to a considerable increase in memory required. Do not use special auxiliary variables for the purpose of efficient programming. The compilers achieve this anyway, and the special variables do not improve readability.
- Use the `SAVE` attribute (not the `SAVE` statement!) when the value of a variable must be retained in the next call of a subroutine.
- Use the parameter statement to define values for constants. This should be done for the logical unit numbers, array dimensions and physical and numerical constants. The numbers are defined on just one place in the program (e.g. module), so that changing the value can be done once. Some constants should not be parameterised, such as π which can be calculated with an intrinsic function: `PI = 4.*ATAN(1.)`.
- Statement labels need to meet the following conditions:
  - They are used exclusively for `FORMAT` statements.
  - They are used in ascending order.
- Character strings need to meet the following conditions:
  - A string which enters a subroutine via the argument list, must be dimensioned as a variable in that subroutine: `CHARACTER (LEN=*)`.
  - The string length may not be transferred via the argument list. The length is determined via intrinsic function `LEN`.
  - Strings should not be used as operands in relational operators, such as `.LT.`, `.GT.`, `.EQ.`, `.NE.`, `.LE.`, `.GE.`. Use FORTRAN functions `LGE`, `LGT`, `LLE`, `LLT` instead.

  - `ICHAR` and `CHAR` must not be used in a way that makes the code dependent on the underlying character set (ASCII, EBCDIC). Use the `IACHAR` and `ACHAR` statements to obtain an ASCII number of a specific character – or to do the reverse.

- Do not use free format output, i.e. $*$, but always use `FORMAT` statements. This facilitates the comparison of the output obtained on various computers. The use of a format in the `WRITE` statement itself is not recommended, since it does not improve the readableness.

- Use arrays as units in expressions to improve readability. This means that the arrays can be treated as scalar quantities, and can therefore be subject to practically all scalar operations and functions.

- Avoid data-dependency in array expressions.

- Initialize every variable in a subroutine and never assume the compiler sets any value at 0. Avoid saving a local variable value as much as possible. If this value does need to be saved, then a `SAVE` attribute must be used.

- Do not use the following, obsolete, standard FORTRAN 77 statements:

  - Statement functions, i.e. a function in one line within a different routine.
  - Assigned goto.
  - Named common en blank common.
  - Arithmetic if (`IF ... 100, 200, 300`).
  - `EQUIVALENCE`
  - `ENTRY`
  - `ERR=` and `END=` in read instructions; use `IOSTAT` specifier instead.
  - `PAUSE`
  - More than one entry or output per module (`ENTRY`, alternate `RETURN`).
  - Jump to `ENDIF` from outside `IF` block.
  - Non-integer index in a do loop.
  - The H edit descriptor.

- Always use the generic name for intrinsic functions, so `ABS` instead of `CABS` or `DABS`.

- Use the following operators in subroutines to be generated: `>`, `>=`, `<`, `=<`, `==` and `/=`. Using the former style, such as `.GT.` and `.NE.`, is acceptable in existing subroutines.

- Do not use a `STOP` except in the following cases:

  - an error subroutine after printing of a fatal error,

  - in an explicit lockup routine

- Use the `SELECT CASE` construct in the case of more than two mutually exclusive choices.
- Do not use `WHERE` statements, as these generally have a negative effect on performance.
- Do not use BLAS routines.
- Do not use a `GOTO` statement except for error handling. Jump to a `CONTINUE` statement with a numerical label (usually 9999), after which the error can be dealt with.
- Develop the code such that it may improve the performance. This means that clean do loops should be used and, that `IF THEN ELSE` constructions should be placed outside the do loops, if appropriate. See also Chapter 9 for more details on how to optimize the code.
- Do not write intermediate results to background memory unless there is a very good reason to do so. I/O is very expensive and must therefore be avoided wherever possible. It is often much more favorable to save the intermediate results in an array or, if not enough memory is available, to recompute them.
- In the case of multi-dimensional arrays applied in nested do loops, it is preferred to use the inner loop for the first index. On vector computers and machines containing a cache memory, this approach results in a considerably higher efficiency.

Chapter 3

Control statements

In principle, a net program contains three control structure:

- sequence,
- selection and
- iteration.

These program statements for FORTRAN are discussed below. Other constructions and FORTRAN control statements should not be applied.

- Sequence.
  This is a series of statements and/or references to a process elsewhere (via `CALL`). Exit a subroutine only via `END`.

- Selection.
  There are two constructions allowed:

  – IF - THEN - ELSE
  – SELECT CASE

  The first construction enables a choice to be made between two options and is programmed as follows:

```
IF (condition) THEN
   statements
ELSE
   andere statements
ENDIF
```

The second construction is programmed as follows:

```
SELECT CASE (variable)
   CASE (value1)
       statements
   CASE (value2)
       statements
   ...
   CASE default
       other statements
END SELECT
```

An alternative for the latter construction is the construction `IF - THEN - ELSEIF - ELSE`. A major disadvantage of this construction is that all alternatives need to be tested, even if it takes until the end of the list to find that an option in fact applies.

- Iteration.
  There are various iteration options, but only the following three are allowed:
  - DO - ENDDO
  - DO WHILE
  - REPEAT UNTIL

  The first construction is a counter-controlled loop:

```
DO i = istart, iend, istep
   statements
ENDDO
```

If necessary, use the `CYCLE` and `EXIT` statements to immediately start the next iteration step of the loop or to exit the loop, respectively.

Although, FORTRAN 90 does recognize the `DO WHILE` statement, it is recommended to simulate this construction by using the `EXIT` statement:

```
DO
   IF (.NOT.condition) EXIT
   statements
ENDDO
```

instead of

```
DO WHILE (condition)
   statements
ENDDO
```

The third construction is programmed as follows:

```
DO
   statements
   IF (condition) EXIT
ENDDO
```

Chapter 4

Use of modules

The idea of modules is data hiding and locality. Place groups of related subroutines and data together in one module. Common parts of the software architectures of the system are candidate to be placed in modules.

Due to the availability of modules, common blocks are prohibited.

Explicit interfaces should be use. The interfaces are automatically known in the calling routines by means of the `USE` statement. Two ways are possible:

- Place the whole subroutines in the module.
  The drawback is, however, that a small change in a module procedure, even it is declared private and the interfaces are not changed, can cause a cascade of compilations in all the places where the module is used.

- Place only the interfaces in the module.
  The drawback is that the interfaces appear twice: in the subroutine and in the module.

Both choices are allowed.

Make the interfaces of the subroutines explicit. Place several modules where the interfaces and subroutines are programmed in the same file. Redefinition of functions is only allowed when a naming conflict occures, otherwise redefinition of functions is prohibited.

All entities in a module must be declared private by default, and thereafter the entities are declared public which have to be known outside the module. Modules may not contain data that is shared by different subroutines of a program. Modules may contain private data.

Chapter 5

Program layout

The following general rules apply to the layout:

- In the case of nesting, indent by three positions.
- Separate the sub program units by brief comments lines.
- Insert a extra space before and after every comment block.
- Never use *tabs*, as this causes problems with transport to other machines.

The standard layout of a program or subroutine is described below.

1. Program unit statement (`PROGRAM`, `SUBROUTINE`, `FUNCTION`, `MODULE`).
2. Comment block consisting of:
   - Programmer's name.
   - Version number followed by the date and machine on which it was developed
   - Version numbers of the modifications including dates, followed by a brief description of the modification and the reason for modifying.
   - Any applicable copyright, preferably GNU Public License (see *http://www.gnu.org*).
   - Description of the subroutine function, if necessary with reference to a detailed design.

   The version number consists of several digits separated by a decimal point, and has a start value of 1.0. Major modifications result in an increase by a value 1 of digit 1, and in the decimal being set at 0. The modified digits are filled in on the next line to retain the previous information, and the modification is briefly described in brackets. Minor modifications result in the decimal being increased by a value 1 and, if required, a brief description of the modification.

   Routines which are clearly sensitive to machine accuracy contain a warning in the comments.

3. Declaration of input and output variables, followed by a comment block with a description of these variables marked with *i* (input) and/or *o* (output). Use the lexicographic order, and always use the `INTENT` attribute.
4. Local variables:
   - Declaration of variables.
   - Any `SAVE` attributes, parameter and data statements.
   - A comment block with a description of all local variables. Use the lexicographical order.
5. Comment block consisting of:
   - I/O description including unit numbers and files used.
   - Description of all subroutines used.
   - Description of all subroutines which call the current subroutine.
   - Error messages.
   - Description of the module as a pseudo code (see Chapter 8).
6. Program text.
   Format statements are given at the end of the program unit.

A template format of a subroutine is given in Chapter 13. No layout must be used for SWAN other than the layout of that example.

Chapter 6

Input en output

- Do not use the `PRINT` statement, but only the i/o statement keywords `READ` and `WRITE`. Never use $*$ as a device number because the device linked to $*$ is machine-dependent.
- File unit numbers should be provided by a generic function or subroutine.
- Use variables for logical unit numbers in `READ` and `WRITE` statements.
- Logical unit numbers smaller than 11 are prohibited.
- Always use `IOSTAT=` instead of `ERR=` or `END=` for testing on i/o status.
- Only use a blank for carriage control.
- Create user-friendly error messages including the name of the module where the error occurred.
- Close files which are no longer used.
- Open statements are machine-dependent (see Chapter 10).
- The input should not differentiate between upper case and lower case.
- Input of numerical data should always be free format.

Chapter 7

Error messages

The standard option selected for errors is the *hard* option (imperative). This means that a message appears when an error occurs and, if necessary, the activity is aborted. The *soft* option (non-imperative) is applied only in the case of a numerical method error, for example due to a lack of convergence. The activity may be continued, while an error number is added to the parameter list. The imperative option is set as default, so users need to define explicitly if they prefer the non-imperative option, by means of a parameter in the parameter list.

Differentiate between errors and warnings.

Always stop if too many, for example 10, errors occur and print a maximum of 10 warnings. The error subroutine entails a `STOP` only if the number of errors is exceedingly high. A `STOP` needs to be built into the calling subroutine after fatal errors. In the case of input errors, continue until all errors have been given.

Chapter 8

Pseudo code

8.1 Using the pseudo code

It is recommended that all programmers use the same conventions for the pseudo code. For a mathematical description of the algorithm, the algorithm needs to be written exactly as it would appear in an article. Use superscripts and subscripts for upper and lower indices respectively.

Provide clear translations for unavailable symbols, for example *sqrt* (square root) or *sum* (sum).

The structures used are those resembling a structural diagram, except that indentations are used instead of lines. An alternative is Jackson Structured Programming (JSP), but its readability is slightly less than that of a structural diagram.
The three control structures result in:

- sequence
  Statements are described one below the other.

- selection

  - IF - THEN - ELSE:

    ```
    if ... then
       statements
    else
       statements
    ```

  - SELECT - CASE:

    ```
    select ...
      case ..
        statements
    ```

```
case ..
  statements
...
else
  statements
```

- iteration
  The three allowed structures are written as follows:
  - DO ENDDO

    ```
    for i = i1 (i2) i3
       statements
    ```

  - DO WHILE

    ```
    while ...
       statements
    ```

  - REPEAT UNTIL

    ```
    repeat until ...
       statements
    ```

8.2 Input and output

Input and output can be given in descriptive text, which needs to include the source file or source machine of the input and output. Considering the fact that many computers have some standard input, such as batch data or keyboard/terminal, and some standard output, such as a screen or printer, the abbreviations STDIN and STDOUT can be used as standard files. All other files need to be named by means of an `OPEN`, after which they need to be closed again.

8.3 Further operations

In general, it is useful to give an outline in descriptive text and to avoid irrelevant details. Use as little programming-language dependent phrases as possible. If more details need to be given, a FORTRAN/PASCAL type code can be used, for example formulas including +, −, *, /, and (). For assigning, := is preferred. Conditions may include =, >, <, etc., but not as logical expressions.

In pseudo code no type distinction is made between integers and reals. Numbers are considered real with infinite accuracy. If necessary, although rarely, PASCAL operators DIV (integer division) and MOD (remainder) can be used.

Notes:

- The purpose of the pseudo code is to give a clear description of the algorithm. It is not the purpose to describe each statement in the pseudo code. However, the pseudo code must be such that an arbitrary programmer may code the subroutine immediately from the pseudo code.
- If the formulas or the algorithms are so complicated that there is no point in including them into the source codes, it is sufficient to refer to a document giving the full details. This document must be mentioned next to the source code.

Chapter 9

Some advices on the improvement of performance

Below we shall give some advices of how to improve the performance of a code.

- Use only clean loops without `IF`, `GOTO` or `CALL` statements. For example:

```
DO i = 1, n
   IF ( var > 0 ) THEN
      a(i) = 0.
   ELSE
      a(i) = b(i)
   ENDIF
ENDDO
```

  should be replaced by

```
IF ( var > 0 ) THEN
   a(:) = 0.
ELSE
   a(:) = b(:)
ENDIF
```

- Make sure that the inner loop is the longest one of a series of nested do loops.
- Use the inner loop for the first index of multi-dimensional arrays in case of nested loops. This may increase the efficiency, provided the inner loop is long enough.
- Avoid indirect addressing, if possible.
- Avoid division in a do loop.
- Avoid (seemingly) recursions in a loop.

Chapter 10

Machine dependency

Machine dependency can be found in various cases, in particular:

- Precision used for calculating.
- Open statements.
- Machine-dependent constants.
- Plot instructions.

These cases are discussed below.

- There are computers which calculate in 64 bits, for example several super computers such as CRAY. This means that large-scale numerical applications need to be calculated in double-precision. The following rules need to be observed for easy distillation of a single-precision version from such a double-precision version:
  - Declare explicitly with `DOUBLE PRECISION` or `COMPLEX * 16`, repeat the declaration with `REAL` and `COMPLEX`, but include the letter combination *ce* into the relevant lines in columns 1 and 2. Always write `DOUBLE PRECISION` with one space and always write `COMPLEX * 16` with one space on either side of the $*$. Always write these statements in lower case.
  - Always use generic names for intrinsic functions.

  These rules allow the simple conversion of a double-precision into a single-precision version and vice versa.

- Unfortunately, open statements are machine-dependent. The filename presents the biggest problem. According to the official standard the filename can be incorporated into the open statement, but an IBM mainframe rejects this. It is therefore necessary to include as many open statements as possible in one machine-dependent subroutine. Additionally, different computers can then accept different filenames.

- Certain machine-dependent constants, such as machine accuracy and largest real number, are used in many programs. It is recommended to initialize these constants in a general machine-dependent subroutine and to include them in a module.

- Plot instructions are not only machine-dependent, but also device-dependent and package-dependent. Using real plot calls in programming should be avoided. This can be done by creating a separate plot routine which simulates plot calls – at any level. This plot routine can contain the machine-dependent calls or, if necessary, can generate a local neutral file. It is recommended to use as few separate machine-dependent plot routines as possible.

Chapter 11

Exceptions

There is a situation where an explicit deviation from the FORTRAN standard is acceptable, namely array transfer to underlying subroutines. On the one hand, it is accepted that the dimension number and size in a current array are different to the number and size in an associated dummy argument array declarator, while on the other hand it is accepted that arrays are transferred by using the start address. In both situations explicit use is therefore made of the FORTRAN characteristic that array transfer means no more than start address transfer.

It goes without saying that this option goes against programming ethics and should therefore be avoided wherever possible, although its use is inevitable in larger packages. Several reasons are:

- In certain subroutines, the option preferred from a structural point of view is to use an array as an n-dimensional array (e.g. for the matrix structure), whereas in other subroutines an m-dimensional array is preferred (e.g. for the linear solver). If the same dimension were used in both subroutines, the address calculations would need to be carried out time and again. This has a negative effect on efficiency and definitely on readability.
- Another reason for array transfer by using start addresses, is that it avoids the number of parameters in a routine at top level and therefore avoids errors. From a structural point of view, however, it is preferred to work with several arrays in the underlying subroutines.

It is clear that the above approach is not without its risks and is therefore not allowed, except in the top layer of subroutines the user needs to handle. All underlying subroutines must be neat and tidy. In addition, if these options are used, the array bounds need to be checked carefully, because neither checkers nor debuggers are then able to carry out the checks.

Chapter 12

Names for SWAN program units

SWAN subroutines at top level can be given any name, because the user is familiar with these routines. The names therefore need to be mnemonic. It is recommended to opt for specific combinations for underlying subroutines, however, so that users with their own programming do not run into name conflicts. The suggestion is to start all SWAN routines with the letter combination *sw*.

Unambiguous names are also necessary for the modules. For SWAN, the combination *swmod* could for example be followed by a 3-digit symbol.

In any case, it is essential that an ASCII file with applied subroutine names and modules is kept up-to-date, including a brief description of the functionalities.

Chapter 13

Examples

Please find below a SWAN subroutine layout example complying with the programming rules set out in this document.

```
!****************************************************************
!
      SUBROUTINE SWBODY ()
!
!****************************************************************
!
      USE OCPCOMM4
!
      IMPLICIT NONE
!
!
!   --|-----------------------------------------------------------|--
!     | Delft University of Technology                            |
!     | Faculty of Civil Engineering                              |
!     | Environmental Fluid Mechanics Section                     |
!     | P.O. Box 5048, 2600 GA  Delft, The Netherlands            |
!     |                                                           |
!     | Programmers: R.C. Ris, N. Booij,                          |
!     |              IJ.G. Haagsma, A.T.M.M. Kieftenburg,         |
!     |              M. Zijlema, E.E. Kriezi,                     |
!     |              R. Padilla-Hernandez, L.H. Holthuijsen       |
!   --|-----------------------------------------------------------|--
!
!
!     SWAN (Simulating WAves Nearshore); a third generation wave model
!     Copyright (C) 2005  Delft University of Technology
```

```
!
!     This program is free software; you can redistribute it and/or
!     modify it under the terms of the GNU General Public License as
!     published by the Free Software Foundation; either version 2 of
!     the License, or (at your option) any later version.
!
!     This program is distributed in the hope that it will be useful,
!     but WITHOUT ANY WARRANTY; without even the implied warranty of
!     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
!     GNU General Public License for more details.
!
!     A copy of the GNU General Public License is available at
!     http://www.gnu.org/copyleft/gpl.html#SEC3
!     or by writing to the Free Software Foundation, Inc.,
!     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
!
!
!  0. Authors
!
!     40.23: Marcel Zijlema
!
!  1. Updates
!
!     40.23, Aug. 02: New subroutine
!
!  2. Purpose
!
!     Description of the purpose of this subroutine
!
!  3. Method
!
!     Description of the method
!
!  4. Argument variables
!
!     VARIABLE    meaning
!
      INTEGER, INTENT(IN) :: VARIABLE
      REAL
      CHARACTER (LEN=n)
      LOGICAL
!
!  5. Parameter variables
```

```
!
!     ---
!
!  6. Local variables
!
!     I     :     counter
!     J     :     counter
!
      INTEGER :: I, J
!
!  8. Subroutines used
!
!     ---
!
!  9. Subroutines calling
!
!     ---
!
! 10. Error messages
!
!     ---
!
! 11. Remarks
!
!     ---
!
! 12. Structure
!
!     Description of the pseudo code
!
! 13. Source text
!
      SAVE IENT
      DATA IENT/0/
      IF (LTRACE) CALL STRACE (IENT,'SWBODY')

      ...
      RETURN
      END
```

Please find below a SWAN module layout example complying with the programming rules set out in this document.

```
      MODULE MODBODY
!
!
!   --|-----------------------------------------------------------|--
!     | Delft University of Technology                            |
!     | Faculty of Civil Engineering                              |
!     | Environmental Fluid Mechanics Section                     |
!     | P.O. Box 5048, 2600 GA  Delft, The Netherlands            |
!     |                                                           |
!     | Programmers: R.C. Ris, N. Booij,                          |
!     |              IJ.G. Haagsma, A.T.M.M. Kieftenburg,         |
!     |              M. Zijlema, E.E. Kriezi,                     |
!     |              R. Padilla-Hernandez, L.H. Holthuijsen       |
!   --|-----------------------------------------------------------|--
!
!
!     SWAN (Simulating WAves Nearshore); a third generation wave model
!     Copyright (C) 2005  Delft University of Technology
!
!     This program is free software; you can redistribute it and/or
!     modify it under the terms of the GNU General Public License as
!     published by the Free Software Foundation; either version 2 of
!     the License, or (at your option) any later version.
!
!     This program is distributed in the hope that it will be useful,
!     but WITHOUT ANY WARRANTY; without even the implied warranty of
!     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
!     GNU General Public License for more details.
!
!     A copy of the GNU General Public License is available at
!     http://www.gnu.org/copyleft/gpl.html#SEC3
!     or by writing to the Free Software Foundation, Inc.,
!     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
!
!
!  0. Authors
!
!     40.41: Marcel Zijlema
!
```

```
!  1. Updates
!
!     40.41, Apr. 05: New Module
!
!  2. Purpose
!
!     ---
!
!  3. Method
!
!     MODULE construct
!
!  4. Modules used
!
!     ---
!
      IMPLICIT NONE
!
!  5. Argument variables
!
!     ---
!
!  6. Parameter variables
!
!     ---
!
!  7. Local variables
!
!     ---
!
!  8. Subroutines and functions used
!
!     ---
!
!  9. Subroutines and functions calling
!
!     ---
!
! 10. Error messages
!
!     ---
!
! 11. Remarks
```

```
!
!     ---
!
! 12. Structure
!
!     ---
!
! 13. Source text
!
      END MODULE MODBODY
```

Bibliography

[1] Chapman, S.J. (1998). *Fortran 90/95 for scientists and engineers*, WCB McGraw-Hill, Boston, USA.

[2] Morgan, J.S. and J.L. Schonfelder (1993). *Programming in Fortran 90*, Alfred Waller Ltd., Henley-on-Thames.

Log sheet

Document version	Date	Modification of previous version
1.0	April 28, 2005	Initial document
1.1	May 19, 2005	Minor modifications in response to comments by RWS − RIKZ
1.2	January 10, 2006	Translation into English
1.3	March 22, 2006	Some minor modifications and extensions