Simulation Reports CABO-TT

Subprograms in simulation models

D.W.G. van Kraalingen and C. Rappoldt[*]

Simulation Report CABO-TT nr. 18

[*]Author names are in alphabetical order

A joint publication of

Centre for Agrobiological Research (CABO)

and

Department of Theoretical Production Ecology, Agricultural University

Wageningen 1989

Simulation Reports CABO-TT

Address

Simulation Reports CABO-TT
P.O. Box 14
6700 AA Wageningen
Netherlands

Authors affiliation

(Alphabetical order)
D.W.G. van Kraalingen and C.Rappoldt:
Department of Theoretical Production Ecology
P.O. Box 430, 6700 AK Wageningen, The Netherlands

Preface

The simulation programs in the early days of modelling were relatively small and easy to understand. The amount of existing knowledge was less than nowadays and much of the existing knowledge simply had not yet been put into the model. By including more and more information into existing simulation models, often much of the original structure was retained leading to unstructured programs that were error prone. With the advent of structured and modular programming, many people realized that this provides a way out.

A general programming language like FORTRAN leaves the user an enormous amount of freedom. A simulation program which is structured and modular in the technical sense is not automatically a readable and adaptable model. Many different solutions to problems as task definition, initialization and time control are possible and they are not equally clear. We have tried to give simple solutions to a series of problems known to everyone writing crop growth simulation models in FORTRAN. The basis of this report is a classification of subprograms according to their function in a simulation model. Writing a simulation model, a clear insight in these matters is very useful.

Through our experience with models we are convinced that the spin-off from structured and modular programming can be increased considerably by adopting standard solutions. After lengthy discussions, and taking into account pro's and con's of a particular solution we decided to write down our conclusions in this report.

This report is not meant as a guide for writing neat, syntactically correct FORTRAN programs. There are very good books serving that purpose and you may find some in the list of references. Writing a modular simulation model using the guidelines we provide in this report may speed up program writing, reduce errors and stimulate the exchange of subprograms among scientists. We hope this report contributes to the improvement of simulation models.

Daniel van Kraalingen and Kees Rappoldt

Contents

Samenvatting

Een toenemend aantal simulatiemodellen maakt gebruik van FORTRAN subprogramma's of is helemaal geschreven in modulair opgezet FORTRAN. Er bestaat echter geen overeenstemming over de wijze waarop subprogramma's gebruikt moeten worden. Dit rapport geeft aanwijzingen, richtlijnen en oplossingen voor problemen bij het ontwikkelen van modellen. In de eerste twee hoofdstukken worden de voordelen besproken van modulair programmeren en wordt een klassifikatie gegeven van subprogramma's. In de hoofdstukken 3-9, worden oplossingen gegeven voor problemen die bij het schrijven van subprogramma's optreden. In hoofdstuk 10 wordt een eenvoudig gewasgroeimodel besproken. Hoofdstuk 11 handelt over het compileren en linken van programma's. In hoofdstuk 12 worden voorbeelden van subprogramma's gegeven die volgens de in dit rapport gegeven richtlijnen zijn geschreven.

Summary

A growing number of simulation models has incorporated FORTRAN subprograms or is written entirely in modular FORTRAN. There is however no concensus on how to use subprograms. This report provides hints, guidelines and solutions to problems that occur during model development. In the first two chapters, the advantages of modular programming and the different types of subprograms that can be recognized are discussed (Chapter 1 and 2). In the Chapters 3-9, hints are given and solutions presented to problems that normally arise during subprogram writing such as file handling and initialization. In Chapter 10, a very simple FORTRAN crop growth simulation program is discussed. Chapter 11 deals with compilation and linking. In Chapter 12 examples are given of subprograms that have been written according to the guidelines of the report.

1 Introduction

A large model is born as a small model. Initially only few processes are modelled. During development of the model, processes and interrelations are included. Often additions describing new aspects are dispersed throughout the program. In this way the description of subprocesses may easily become inaccessible for anyone but the author. The more the program structure deviates from the structure of the system, the harder it becomes to understand the program. This practice results in a vicious circle, existing programs lack a structure and are unsuitable to extract parts from, as a consequence everybody starts from scratch which in turn leads to yet another model of bad design. The formulation of well-defined subprograms and the use of a few conventions is a way to overcome this difficulty.

Advantages of the use of subprograms are :
- transportability with respect to other computers, models, programming languages,
- accessibility of the information for the author (improvements are simple to introduce), and other users,
- subprograms can be maintained and tested individually,
- simplification of the development of large models by making use of well-documented existing subprograms,
- speed of execution (PC) is improved and computer costs (VAX) are reduced since subprograms need no repeated compilation,
- students who have a limited amount of time, may work on the description of a subprocess without having to deal with the text of a large model.

To fully benefit from the use of subprograms, in this report some guidelines are proposed. Some topics are general, like declaration of variable type or the use of comment lines, and others are typical for simulation models, like initialization. Subprograms are classified according to their function in a simulation model. This classification makes it easier to recognize the different parts of a simulation program.

2 Classification of subprograms

SUMMARY - subprograms are classified according to their function,

I. Service routines
II. Generation of driving variables,
III. Rate calculations,
IV. Fast subprocess description,
V. Simulating subprograms,

- subprograms can only call other subprograms with equal or lower number.

Subprograms may be classified with respect to their function within a simulation model. Five classes of subprograms are distinguished. In order of increasing complexity these are:

I. Service routines

Service routines are independent of the described system. They are used to perform well defined tasks such as linear interpolation (see Example I, Chapter 12), generation of random numbers, curve-fitting, sorting of data, calculation of the roots of a function etc. This type of function will often appear in libraries.

II. Generation of driving variables

Driving variables are variables that are part of the system but are not determined by the system. In other words, these variables are input of the simulation program and their time dependence is not simulated but prescribed. Weather variables (e.g. radiation, temperature), sometimes groundwater table or leaf area index may be driving variables. Examples of this type of routines are:

- reading weather data from file,
- estimation of hourly temperatures from daily maximum and minimum (see Example II, Chapter 12),
- reading irrigation data from file,

This type of subprogram creates an 'environment' for the actual simulation.

III. Rate calculations

Rates of change are determined by the state of the system and its parameters. Thus, state variables and parameters are input for this type of subprogram and one or more rates of change are output. Examples are:

- calculation of the CO_2 assimilation rate of a crop canopy,
- flow rate of water between adjacent soil layers,
- evapotranspiration rate of water from soil and crop (see Example III, Chapter 12),
- suction rate of aphids,
- number of predated aphids by ground beetles,
- mineralization rate of organic nitrogen.

IV. Fast subprocess description

A fast subprocess is always in equilibrium from the point of view of the main process. It may be convenient to calculate the equilibrium state in a subprogram. In that way the subprocess can be described independent of the main process described in the calling program.

About the calculation method nothing can be said in general. Sometimes equations can be solved algebraically. Sometimes an iterative procedure has to be used to find the equilibrium state. In complicated situations it may be necessary to simulate time course of the process until equilibrium is reached.

Contrary to the next type of subprogram (simulating subprograms), the subprocess time is not controlled by the calling program. The subprocess is assumed to be so fast that only equilibrium states are needed. So even in case a (local) subprocess time occurs explicitly in the subprogram, it runs freely until equilibrium is reached.

Examples are:
- stomatal behavior responding to CO_2 and humidity,
- generation of an average wind profile in the crop while the crop is simulated with a time step of one day.

V. Simulating subprograms

Writing a simulation model in FORTRAN one is tempted to write a main program. There is an initial section in which initial calculations are carried out, initial settings are done, input files opened etc. The central part of the model is the integration loop in which rates of change are calculated and integrated to find the new state variables. Possibly a terminal section is executed after finishing the integration.

There is however, no good reason for writing a simulation model as a main program. One could equally well write a subroutine in which the above described sections are all present. A call to this subroutine causes the execution of the simulation model.

The use of this kind of simulating subroutines already has distinct advantages. Sensitivity analysis, for instance, becomes easier since the model parameters concerned can be passed to the subroutine as arguments. There are other types of model use that require more than a single simulation run. It is always advantageous then to have the simulation model available as a subroutine.

Sofar a call to a simulating subroutine leads to a complete simulation run. The subroutine is free running in the sense that the calling program does not control the integration process. The simulation goes on until some finish condition is reached. Now suppose one has available two models in the form of two subroutines. One model, for instance, simulates crop growth as a function of weather data and soil water content. A second model simulates the soil water content as a function of weather data and crop condition. The two models may have different authors and may originate from different research groups. Improved versions of both may appear from time to time.

The combination of the two subroutines into a single program tends to require a large programming effort. Therefore it is worthwhile to consider solutions to this problem that leave the different models in their original form as much as possible.

Clearly the simulation of the whole system consists of the simulation of the two subsystems. The two simulating subroutines, however, cannot be left free running in this case. The calling program needs to control time progress in the two subroutines in order to synchronize the two processes taking place. This can be done by switching between the subsystems. Both subsystems are simulated for a short, prescribed timestep. This is easy to realize when the timesteps of the two submodels are equal. The coupling is then called synchronous. The case of unequal timesteps is called asynchronous coupling and is more difficult to handle. The solution will depend on the importance of the interaction between the systems.

Here only synchronous coupling is further discussed. Submodels using equal timesteps are coupled by controlling the integration process from a calling program. A 'task' variable is passed to the submodels that controls the action that should take place: initialization, calculation of rates of change, integration (=return new status) or terminal calculations. In doing so, everything can be controlled from a short main program describing the system as a whole (See the example in Chapter 12).

The difference between a free running simulation model and a simulating subroutine coupled with others is rather small. Both consist of sections for model initialization, rate calculation etc. In a simulating subroutine the execution of these sections is controlled from the calling program by means of the task control variable. When several well defined submodels are coupled, the structure of the program reflects the structure of the system.

3 Subprograms in FORTRAN

SUMMARY
- use passed length array declarations,
- write the array length as integer variable ILxxxx, where xxxx is the first part of the array name it belongs to,
- use COMMON blocks only where they are necessary.
- use blank SAVE,

In FORTRAN two different subprogram types exist nl. subroutines and functions. In cases where only one calculated result is required from a subprogram, the function type of subprogram is chosen (e.g. the LINT function, Example 1, Chapter 12). It is however not illegal to use a subroutine if only one calculated result is required. A subprogram is called from the calling program which may be the main program or another subprogram. During subprogram execution, the execution of the calling program is temporarily suspended. Returning from the subprogram, the execution from the calling program resumes.

FORTRAN subprograms are complete separate programs. They are separately compiled and may be separately stored in memory. Main program variables or variables from another subprogram are not automatically known. Of course, subprograms cannot work together without some form of communication among them. There are two ways to communicate variables from one subprogram to another. Subprograms may have arguments (sometimes called formal parameters), and subprograms may share variables with other subprograms by means of common blocks of memory. This will be discussed in more detail below.

Local variables

All variables that are not communicated with other subprograms are called local. Consider:

```
SUBROUTINE SUB1
REAL C
SAVE

C = 10.
RETURN
END

SUBROUTINE SUB2
REAL C
SAVE

C = 20.
RETURN
END
```

The variable C in SUB1 will thus not interfere with the variable C in SUB2. This helps to understand why subroutines and functions are called subprograms and not for instance subsections or substatements.

The definition of FORTRAN is such that local variables do not retain their values between successive calls.

Example:

```
* variable not saved between calls
        SUBROUTINE SUB1
        INTEGER I

        WRITE (*,*) I
        I = 10

        RETURN
        END
```

The variable I is written to the screen without having a value, consequently, the result is unpredictable. This is because, following the FORTRAN definition, the value of any type of variable is not defined before the first assignment. After the WRITE statement, a value is assigned to I. Subsequent calls to the subroutine SUB1 thus gives the following output:

```
<unpredictable>
<unpredictable>
<unpredictable>
etc.
```

In order to achieve that, in a subsequent call, I still has this value of 10, I should be 'saved' by means of a special SAVE declaration:

```
*       variable saved between calls
        SUBROUTINE SUB2
        INTEGER I
        SAVE

        WRITE (*,*) I
        I = 10

        RETURN
        END
```

This subprogram will now give the following output:

```
<unpredictable>
10
10
10
etc.
```

Note: With VAX-FORTRAN and Microsoft FORTRAN compilers, variables retain their values without SAVE statements. This is however an extension to the standard rule. With MacFortran on the Apple Macintosh, local variables only retain their values when using a SAVE statement.

By means of a SAVE statement without variable names behind the other declarations (a so called blank SAVE), all local variables of the subprogram will retain their values. The use of blank SAVE is suggested here, even if only a few variables actually need to be saved for the following reasons:

Safety:

1) if one variable is by accident not listed behind the SAVE statement, the subprogram will not work properly,
2) if subprograms share COMMON blocks among each other, the retaining of variables may become unpredictable if they are not saved explicitly by means of a blank SAVE.

Speed:

1) subprograms in which only essential variables are saved will run much slower because at each call to the subprogram some memory has to be assigned during calculations to hold the 'volatile' local variables and released on return to the calling program. In subprograms in which all the variables are saved, allocation of memory is done only once during the execution of the whole program.

Arguments

To understand the function of arguments it is necessary to have some understanding of the representation of variables in a compiled program.

The compiler replaces variable names by memory addresses, e.g.

```
A = B * C
```

will be translated into:

```
A1 = A2 * A3
```

(Where A1, A2 and A3 stand for the contents of the addresses of the memory locations).

Each location as it is defined here can hold an integer or a real number (coded in very different ways in FORTRAN).

As stated previously, local variables in a subprogram are given explicit memory locations during subprogram compilation. For arguments appearing in an argument list, a different approach is followed: its memory location is provided by the calling program. A subprogram expects for each of its arguments an actual address provided by the calling program during execution. The subprogram does not use copies of the variables in the argument list, as is sometimes believed. The subprogram makes use of memory locations declared in the calling program and may directly change their contents.

This has the following implications:

- Names of variables do not have to be the same. It is advised however to use the same variable names in calling program and subprogram whenever possible.
- The number of arguments in the 'call' should be the same as the number of arguments of the subprogram.
- The variable types should match. Otherwise the location of, for instance an integer variable is used by the subprogram as if it contains a real value. The result is unpredictable.
- If an argument is an array name, only the location of the first element is passed to the subroutine. In the subroutine, the array should be dimensioned again to make use of the other elements. An error may occur if the subprogram array is declared longer than it actually is, the subprogram may then change values of other variables that do not even appear in the argument list. As nothing is wrong with the array bounds within the routine, this error is hard to find. The programmer is tempted to use this knowledge for tricks that we consider very bad programming, e.g. declaring an array X as X(40) in the calling program and as X(2,20) in a subprogram. Although this kind of practice is standard FORTRAN the above mentioned type of errors will most likely result.

Array declaration in a subroutine

A flexible and relatively safe way of handling array-type arguments is the use of passed-length declaration:

```
      SUBROUTINE EXAMPL (X,ILX)
      IMPLICIT REAL (A-Z)
      INTEGER ILX
      DIMENSION X(ILX)
```

Both the array and its length are arguments of the subprogram. In the calling program an array has to be declared as a local variable in order to reserve memory locations. Close to the actual declaration, also the array length should be given its value. Using a variable for the array length, one should be aware of value changes due to possible errors in the program code. There is a nice way to avoid this problem by the PARAMETER statement (see also Chapter 7):

```
      PROGRAM MAIN
      IMPLICIT REAL (A-Z)
      INTEGER ILX
      PARAMETER (ILX=20)
      REAL X(ILX)
      ..
      CALL EXAMPL (X,ILX)
      ..
```

The parameter statement, as used in this example, instructs the compiler to first change all occurrences of ILX into 20 before compiling the code. As 20 is a constant and not a variable, its value cannot be changed during execution.

Sequence of subprogram arguments

We propose the following general form of the list of arguments of a subprogram:

```
SUBROUTINE EXAMPL (control variables,
                   variables used for initialization,
                   time control for type V routines only,
                   input variables used for normal operation,
                   output variables)
```

Not all types need to be present in a subprogram. In case of a large number of arguments the different types should be written on different program lines. Explanation of the five types:

1. control variables are for instance:
   RESET or ITASK, to initialize the subprogram (see Chapter 4),
   IUNIT, file unit numbers (see Chapter 5),
   IOUT, suppresses output,
   IWAR, returns warnings to the calling program (see Chapter 6),

2. variables used for initialization:
   variables used only during a reset of the subprogram. (see Chapter 4). Mostly these variables will be input only.

3. time control for routines of type V (see example in Chapter 12)

4. input variables:
   single variables and arrays. The array lengths are separately given in the list as single integer variables written directly behind the array name. The proposed array length variable name is ILxxxx.

5. output variables:
   similar to 4. Note that the array lengths of output arrays, are input variables.

Common blocks of memory

Variables that appear in a COMMON statement undergo a special treatment by the compiler. Their addresses are situated in a memory block that may be shared with other subprograms.

Schematically:

```
      PROGRAM EX
      REAL A,B,C
      COMMON /EXAMPL/ A,B,C

      CALL SUB1

      STOP
      END

      SUBROUTINE SUB1
      REAL X,Y,Z
      COMMON /EXAMPL/ X,Y,Z
      SAVE

      A = 10.
      Z = A*X+Z

      RETURN
      END
```

The variables A, B and C in the calling program are situated in the same memory locations as the variables X, Y and Z in the subroutine SUB1. The variable A in SUB1 is local ! Note that only the start address of the common block of memory is shared among the different subprograms. A construction we consider very bad programming is the subdivision of a block into variables in different ways by different routines, e.g. a second subroutine using the same common block could be:

```
      SUBROUTINE SUB2
      REAL X,Y
      COMMON /EXAMPL/ A(2)
      SAVE

      X = A(1)+A(2)

      RETURN
      END
```

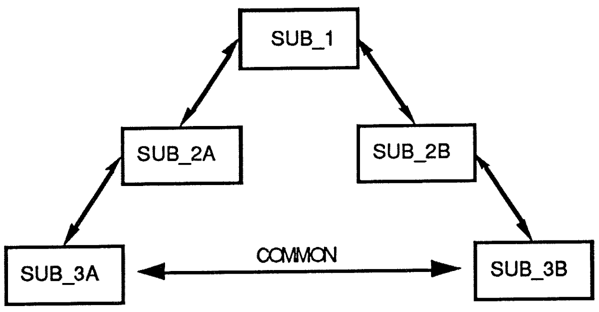
Often COMMON blocks are used to have everything everywhere available. This makes programs unreadable. It is then hard to trace where the value of a specific variable is changed. It is strongly recommended to use COMMON blocks for the tasks they are meant for: necessary communication of variables between subprograms that is hard to accomplish with arguments. Do not hesitate to use a long list of arguments for a subprogram fulfilling a complicated task.

The use of COMMON blocks

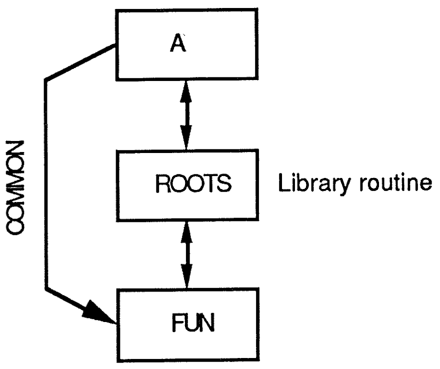
Scheme 1 below represents a hierarchical structure of subprograms in which two low level subprograms share certain variables. Accomplishing this by means of argument lists leads to the inclusion of superfluous arguments in higher level subprograms. It may be much clearer to use a COMMON block of variables shared by the low level routines.

A typical situation when using routines from large numerical libraries such as IMSL, LINPACK, ODEPACK and others is given in Scheme 2. A library subroutine ROOTS is used to find roots of a user-defined function FUN. The argument list of FUN is prescribed by the supplier of ROOTS. There is no way then to communicate (extra) function parameters between A and FUN by means of arguments. A COMMON block shared by FUN and A is the solution.

Scheme 1



Scheme 2



In most cases COMMON blocks can be kept small. If for instance three pairs of subprograms need to communicate, use three small COMMON blocks with a unique name. The three pairs of subprograms may then be used and maintained independently.

4 Initialization of subprograms

SUMMARY - use implicit initialization with LOGICAL INIT,
- or explicit initialization with LOGICAL RESET.

Many subprograms can only perform their defined task after some form of initialization, e.g. reading weather data from file, setting up mathematical functions, reading initial amounts of integrals (subprogram type 5) etc. When making reruns of a model, sometimes repeated initialization is desired.

The initialization of a subprogram A is often done by a call to a different subprogram B that initializes the data from A. In practice this will need communication of the shared variables between the subprograms A and B. This problem will generally be solved by using a COMMON block consisting of variables used in both subprograms. In many cases a more elegant solution is to program the initialization of A into A itself without using a second subprogram B. This can be accomplished by two different methods, that may be called implicit and explicit initialization.

The simplest way is to write a program section that is executed only during the first of several subprogram executions (implicit initialization). This is done as follows:

```
      subprogram A (.....)
      declarations
      LOGICAL INIT
      SAVE
      DATA INIT/.FALSE./

      IF (.NOT.INIT) THEN
*        initialization procedure
         .
         .
*        set initialization flag
         INIT = .TRUE.
      ENDIF

*     rest of subprogram
      .
      .
      RETURN
      END
```

Note that, using this structure, the user of the subprogram has nothing to do with the initialization. It is done automatically during the first time the subprogram is executed. A disadvantage is that initialization can be done only once since it is not controlled by the calling program. This implies that there is no possibility to repeat the initial calculations for a different parameter setting or so. The use of this method in simulation models will generally be limited to low level (service) routines.

To control the initialization of a subprogram in the calling program, a reset parameter has to be supplied as an argument of the subprogram (explicit initialization). The subprogram now can have more than two states, an initialization state at which variables are set to zero, files are opened and closed etc. and no results are returned, a computation state i.e. the normal operation state at which the main task is done, and e.g. a terminal state at which summary statistics are calculated.

If there are only two states, the action of the subprogram can be controlled by means of a LOGICAL RESET. In case more states are possible the INTEGER ITASK is recommended. The control parameter (RESETor ITASK) is the first argument in the argument list and invokes a reset (=initialization) of the subprogram when its value equals RESET=.TRUE. or ITASK=1, any other value will not cause initialization. At any time during simulation, sending RESET=.TRUE. or ITASK=1 will cause a reset of the subprogram and no results are generated. The subprogram should never change the value of the reset parameter. The control of the subprogram is left completely to the calling program.

In general the following structure can be followed:

```
subprogram (RESET,...,
            ...........)
LOGICAL RESET
SAVE

IF (RESET) THEN
   .
   .
   RETURN
ENDIF
.
RETURN
END
```

Four examples will be given. In the first example, implicit initialization with a LOGICAL INIT is used. A calling program is assumed in which a very large number of function values is required, for instance in complicated iteration loops, that has to be combined with a function that is costly to evaluate. The program below reduces the number of function evaluations to e.g. 200. Initially, these 200 function values is calculated. The array of X and Y values is used later to calculate required function values by linear interpolation (using the LINT function, see Chapter 12, Example 1).

```
      REAL FUNCTION FUN (X)
      IMPLICIT REAL (A-Z)
      INTEGER ILBUF,I
      PARAMETER (ILBUF=400,XMAX=100.0)
      DIMENSION BUFFER(ILBUF)

      LOGICAL INIT
      SAVE
      DATA INIT/.FALSE./

*     initialization of buffer with function values
      IF (.NOT.INIT) THEN
*        calculate function values between X=0 and X=XMAX
         XSTEP = 0.5 * XMAX / FLOAT(ILBUF)
         XLOC = 0.0
         DO 10 I=1,ILBUF,2
            XLOC = XLOC + XSTEP
            BUFFER(I) = XLOC
*           costly function evaluation:
            .
            .
            BUFFER(I+1) = ...
10       CONTINUE
         INIT = .TRUE.
      ENDIF

*     evaluation of function value by linear interpolation
      FUN = LINT (BUFFER,ILBUF,X)

      RETURN
      END
```

In a second example a weather file containing radiation and temperature records is opened and read at RESET=.TRUE.. Only the relevant statements and parameters for the reset are shown.

```
      SUBROUTINE WEATHR (RESET,IUNIT,
     $                  FILE,
     $                  IDAY,
     $                  TMPA,AVRAD)
      IMPLICIT REAL (A-Z)
      REAL TMPA1(365), AVRAD1(365)
      INTEGER IUNIT, IDATA, IDAY
      CHARACTER*(*) FILE
      LOGICAL RESET
      SAVE
```

```
*     initialization section
      IF (RESET) THEN
         OPEN (IUNIT,FILE=FILE,STATUS='OLD')
         IDATA = 0
10       READ (IUNIT,'(2F8.2)',END=100) VAR1,VAR2
            IDATA = IDATA+1
            TMPA1(IDATA) = VAR1
            AVRAD1(IDATA) = VAR2
         GOTO 10
100      CLOSE (IUNIT)
         RETURN
      ENDIF

*-----normal operation
      TMPA = TMPA1(IDAY)
      AVRAD= AVRAD1(IDAY)

      RETURN
      END
```

Example for the initialization of an integrating subroutine:

```
      SUBROUTINE WEIGHT (ITASK,
                         WLVI,
                         DELT,TIME,
                         .......)
      IMPLICIT REAL (A-Z)
      INTEGER ITASK
      SAVE

      IF (ITASK.EQ.1) THEN
*        initialization of state and rate
         WLV = WLVI
         GLV = 0.
         RETURN
      ELSE IF (ITASK.EQ.2) THEN
*        calculation of the growth rate
         GLV = ........
      ELSE IF (ITASK.EQ.3) THEN
*        new weight is old weight plus the growth during the stime
*        step
         WLV = WLV+GLV*DELT
      ELSE
         ....
      ENDIF

      RETURN
      END
```

In this example, a hypothetical output routine is shown. It is defined in such a way that at RESET=.TRUE. a new output file by the name of OUTPUT.DAT is opened and thus, an old output file is closed and deleted. This means that at the very first call to the routine (also with RESET=.TRUE.) there are no files to be closed and deleted. The presence of an opened file thus has to be known. This is done with the logical FOPEN which is given the initial value of .FALSE. by the DATA statement.

```
SUBROUTINE OUTPUT (RESET,
                   FILE,..............)
IMPLICIT REAL (A-Z)
LOGICAL RESET
CHARACTER*(*) FILE
LOGICAL FOPEN
SAVE
DATA FOPEN /.FALSE./

IF (RESET) THEN
   IF (FOPEN) CLOSE (UNIT=99)
   OPEN (UNIT=99, FILE=FILE, STATUS='NEW')
   FOPEN = .TRUE.
   .
   .
   RETURN
ENDIF
.
WRITE (99,.....).....
.
RETURN
END
```

5 Unit numbers and file handling

SUMMARY
- do not overrule default unit numbers,
- always open files explicitly,
- use file unit numbers of value 40 or higher,
- pass unit numbers as arguments (part of the control line),
- immediately close files if possible,
- do not use TYPE and ACCEPT for screen input and output but READ and WRITE.

The input and output operations (I/O) from a FORTRAN program (mostly reading and writing to file or screen) are always done by referencing a unit number. The (integer) value of a unit number is between 0 and 255. Some unit numbers like 0, 3, 5, 6,13, 14, and 15, may have a preassigned use. It is considered bad programming practice to overrule preassigned unit number for purposes other than the default ones (e.g. opening a file in VAX-FORTRAN with UNIT=5). This may cause problems and overruling the default assignments should thus be avoided.

In VAX-FORTRAN, files don't have to be opened explicitly. The file is then created or opened at the first read or write operation. This is however not standard FORTRAN and therefore file I/O must be preceded by an OPEN statement.

Unit numbers of files are not local to a module !! In other words, if in a subroutine a file is opened with UNIT = 40, another subroutine cannot open another file also with UNIT = 40 unless the first one is closed. Modules thus can interfere with each other by referencing the same units.

Modules can access files in two ways:

1) A file is opened and closed during the same call to the subroutine. This will occur mostly in situations where data are read from a file. The unit number is thus released at the return from the module.
2) A file is opened during the first call to the subroutine and remains open during subsequent calls. This will occur mostly when data are output to a file during simulation. The unit number thus remains occupied after return from the module.

To avoid interference between modules that use file I/O, the unit number(s) should be passed as arguments in the argument list (part of the control line) instead of defined in the subroutine itself. In that way the calling program keeps control over the used unit numbers. The integer value of the units should be 40 or higher so as not to overrule the default assignments.

In case a subroutine uses unit numbers, this should be indicated in the comment header (see

Chapter 8), it should also be indicated if the unit number is used only once (the file is opened and closed during one call) or if the unit number is used at each call (the file remains open).

I/O operations to the screen

It is often desirable to read and write information from and to the screen. With VAX-FORTRAN this can be done with the TYPE and ACCEPT statements. These statements however are not standard FORTRAN and MS-FORTRAN will not recognize them. The standard FORTRAN alternatives to TYPE and ACCEPT are READ and WRITE from UNIT = * (see example).

Example:

```
      TYPE '(A)',' How many iterations:'
      ACCEPT '(I)',ITER
```

should be:

```
      WRITE (*,'(A)') ' How many iterations:'
      READ (*,*) ITER
```

Note: There seems to be no standard FORTRAN way to hold the cursor at the end of a question. Depending on the compiler, a dollar ($), a backslash sign (\), or nothing at all behind the A FORMAT is used (e.g. '(A,$)').

6 Warnings and errors as generated by modules

SUMMARY
- errors in case input or output values are impossible,
- errors terminate programs with STOP 'message'
- warnings in case input or output values are improbable,
- warnings return a non-zero IWAR, negative for underflow, positive for overflow.

In some cases it can be useful to check if the value of an input variable is within a range that can be handled by the subprogram. Some combinations of input values however may give wrong results (e.g. A can be less than 1 and B can be less than 1 but not A and B at the same time). In some cases it may be useful to check also output variables. In case some variable has exceeded its input or output range, either a warning or an error will occur, dependent on the severity.

Errors:

An error occurs when the value of an input or output variable is impossible (e.g. negative soil moisture content, minimum temperature greater than maximum temperature).

An error should result in the termination of the run by means of a STOP statement (note: CALL EXIT () is not standard FORTRAN). A message is then displayed on the screen. For instance:

```
IF (LAI.LT.0.) STOP 'ERROR in EXAMPL: LAI < 0'
```

Note: on some machines the message may not be readable since the output screen disappears on termination of the program. In the source text library TTUTIL (available on disk on request), a routine is provided to handle this:

```
IF (LAI.LT.0.) CALL ERROR ('EXAMPL','LAI < 0')
```

Warnings:

A warning occurs when the value of an input variable is very unlikely but not impossible. A warning however will not terminate the execution.

A warning results in one of the return arguments (IWAR) set to a non-zero value. IWAR is an argument of the I/O list (see Chapter 3) and can be used in the calling program to see if any warnings have occurred. An underflow of the valid range will result in a negative value of IWAR, an overflow in a positive value of IWAR. The absolute value is different for each variable and must be documented in the comment header of the subprograms.

Example:

```
IWAR = 0
IF (LAI.GT.10.) IWAR = 3
```

7 Type declarations of functions and variables

SUMMARY - use IMPLICIT REAL (A-Z),
- exclusively use Ixxxxx declarations for integer names,
- use the PARAMETER statement to define constants,

In CSMP all variables are REAL by default. In FORTRAN, the variables whose first characters are I,J,K,L,M,N default to INTEGER, others default to REAL. In dynamic simulation few integer variables are needed (mostly for counters in DO-loops). To avoid problems in interfacing CSMP with FORTRAN, all variables in FORTRAN modules are made REAL by the IMPLICIT REAL (A-Z) statement. However few integers are needed, reading a program is much easier if integers can be recognized without checking a list of integer declarations. Names of integers should therefore always begin with an 'I' and explicitly declared INTEGER (see the first example). For clarity, real variables should never begin with an 'I'.

The declaration of arrays that are used later as arguments in a call needs some special attention. To enable passed-length declaration in the subprogram, the array length should be available as a constant or variable. In FORTRAN the PARAMETER statement can be used to define a fixed constant. (For the PARAMETER statement also see Chapter 3).

Example:

```
IMPLICIT REAL (A-Z)
INTEGER PGO(20), DB
DATA DB /20/
CALL EXAMPL (PGO,DB)
```

should be (taking into account the recommendations of this chapter):

```
IMPLICIT REAL (A-Z)
INTEGER IPGO, ILPGO
PARAMETER (ILPGO=20)
DIMENSION IPGO(ILPGO)
CALL EXAMPL (IPGO,ILPGO)
```

8 Header and comment lines

SUMMARY
- write comment in lowercase characters except FORTRAN names,
- write comment above the statement it refers to,
- write program headers containing at least author name and information on the arguments,
- write extensive header for library routines.

Different authors tend to write very different amounts of comments in their programs. It seems therefore rather useless to develop detailed rules for form and content of comment lines. It should be noted, however, that reading a program written by someone else often means reading comment lines. This appears to be much easier if comment lines are written in lowercase characters (except FORTRAN names). Further it is proposed here to write comment lines always above the statements they refer to. It is useful to divide a (sub)program into functional blocks of statements by means of open lines. These blocks may or may not coincide with DO-loops, IF-THEN-ELSE structures etc. A comment line above each functional block greatly simplifies program reading. Comment lines are coded usually with a 'C' in the first column. The '*' also is a valid character and improves readability. The exclamation mark '!' cannot be used as this character only applies to VAX-FORTRAN.

Somewhat more can be said about the form of a program header above a subprogram. This is a block of comment lines describing the program. A header is not meant as full documentation. For a discussion of scientific aspects of the calculations, the subprogram header may refer to the literature. A subprogram header should contain:

1) the subprogram name,
2) the name of the author,
3) a version number incremented by one each time an improvement is made to the subprogram, (so that a user can see how many improvements have been made to the subprogram since he got his copy),
4) a short function description,
4a) optional: known bugs, if a bug in the subprogram has been detected that awaits improvement,
5) a list of arguments and their meaning,
6) a description of error conditions, the meaning of returned warnings, information on file access and the names of called subprograms.

Below is a possible form for a full subprogram header. This header is available from the source-text library TTLIB (available on disk on request). Some 'example lines' are included. They simplify header writing and should be deleted after use.

```
*----------------------------------------------------------------------*
* SUBROUTINE name                                                      *
* Authors: N. Nonsense, I. Irrelevant                                  *
*          based on an earlier version written by:                     *
* Version: n                                                           *
* Purpose: This subroutine is meant to                                 *
*          (references to literature if possible) ...                  *
*          .........................................                   *
* Keywords:Utility, Header,                                            *
* (Known Bugs: optional)                                               *
*                                                                      *
* FORMAL PARAMETERS:  (I=input,O=output,C=control,IN=init,T=time)      *
* name   type meaning                                    units  class *
* ----   ---- -------                                    -----  ----- *
* RESET   L4  resets the routine when .TRUE.              -      C,I  *
* IUNIT   I4  unit number of used file                    -      C,I  *
* FILE    C*  file name                                   -      I    *
* IWAR    I4  output, when .NE.0  warning !!!             -      C,O  *
* XXX     R4  initial value of ......                    xxx     IN,I  *
* BBB     R4  calculated during initialization           xxx     IN,O  *
* ILBBB   I4  array length                                -      I    *
* TIME    R4  current time                                s      T,I  *
* DEAD    L4  death flag ...                             xxx     O    *
*                                                                      *
* FATAL ERROR CHECKS (execution terminated, message):                  *
* XXX < 0.                                                             *
* XXXXX1 > XXXXX2                                                      *
*                                                                      *
* WARNINGS:                                value of IWAR returned      *
* XXX > 100.                                           1               *
* XXXXX2-XXXXX1 > 10.                                  2               *
* AAA < 2.                                            -3               *
*                                                                      *
* SUBROUTINES and FUNCTIONS called :                                   *
*                                                                      *
* FILE usage :                                                         *
*----------------------------------------------------------------------*
```

9 Programming style

SUMMARY
- comment is written above the statements,
- don't use the exclamation mark '!',
- comment lines in lower case characters,
- leave space between continuation character and rest of line,

This chapter is meant to give hints to write more readable programs. FORTRAN is a language that does not force the programmer to write structured programs and it needs some discipline to write programs that don't discourage other readers to study them.

Flow control statements

Flow control statements specify the order in which the computations are done. They comprise the skeleton of a program. FORTRAN-77 control structures are DO-loops and IF-THEN-ELSE. Control structures like DO-WHILE, DO-UNTIL and CASE, are not standard FORTRAN and can be mimicked by functionally equivalent structures in standard FORTRAN. We recommend the methods given by the NNI manual (see references) because it provides 1) simple standard structures and 2) if some time in the future these control structures become part of standard FORTRAN, these structures are very easy to convert.

Example of DO-WHILE loop:

```
*-----do while loop
10    IF (TIME.LE.FINTIM) THEN
         TIME = TIME+1.
         .
         .
      GOTO 10
      ENDIF
```

Use of logical variables

Sometimes the use of logicals improves the readability of IF statements. Complicated decision rules are programmed using logical expressions.

Example:

```
      LOGICAL DEAD, YELLOW
      .
      .
      DEAD = DVS.GT.2. .OR. TMIN.LT.-50.
      YELLOW = DVS.GT.1.8

      IF (.NOT.DEAD .AND. YELLOW) THEN
      .
      .
      ENDIF
```

Indentation

Indentation can be used to spot DO-loops and IF-THEN-ELSE constructs. An indentation of three spaces at each level in the program text is sufficient in most cases. Never use tabs in the program text as the interpretation of tabs by compilers and text editors is not standardized.

Formats

A standard feature of FORTRAN-77 is that FORMAT statements are no longer necessary. FORMAT strings can be included in the READ of WRITE statements directly. This makes programs a lot more readable as the line numbers of the FORMAT statements tend to be very confusing.

Example of old-style FORMAT:

```
      WRITE(20,100) TMIN,TMAX,RAIN
100   FORMAT('Minimum temperature=',F8.2,/,
     & 'Maximum temperature=',F8.2,/,'Rainfall=',F8.2,/)
```

Can now be written as:

```
       WRITE (20,'(3(A,F8.2,/))')
     &       ' Minimum temperature=',TMIN,
     &       ' Maximum temperature=',TMAX,
     &       ' Rain=',RAIN
```

Continuation of lines

Characters beyond column 72 are ignored by the compiler (also not signalled !) and can be put on a second line, preceded by a continuation character. This continuation character can be put in the sixth column (preceded by 5 spaces, the use of tabs is not recommended),

Programs can become very confusing when the program text is typed immediately behind the continuation character:

the following line:

```
*234567
      A = 3.2*B*C*7.413
```

can be written (legally) as follows:

```
*234567
      A = 3.2*B*C*
     17.413
```

or as:

```
*234567
      A = 3.2*B*C
     **7.413
```

It would be less confusing if written like this:

```
*234567
      A = 3.2*B*C*
     &     7.413
```

10 Simulation by Euler integration in FORTRAN

The previous chapters dealt with subprograms meant to be combined with CSMP or FORTRAN main programs. Since FORTRAN is not a simulation language, some attention should be given to the structure of a simulation loop. The following sequence shows the correct order in which the different tasks should be put when using the rectangular integration method.

```
      state = initial, rate = 0
10    IF (TIME.LE.FINTIM) THEN
          integration
          generation of driving variables
          rate calculations
          output of simulation results
          TIME = TIME+DELT
      GOTO 10
      ENDIF
      STOP
```

The example below shows this sequence for a simple crop growth simulation model. After initialization of the state variables, the integration statements follow. Tthe first integration is dummy, since the rates have been set to zero in the initial part of the program. Subsequently, driving variables and rates are calculated. At this point in the program, the state of the system and the corresponding rates of change may be sent to an output device. The sequence of the statements implies that during the first time step the initial state is written to the output device.

```
      PROGRAM EXAMPL
      IMPLICIT REAL (A-Z)
      INTEGER IDAY, INT, IWAR
      CHARACTER STAT*20
      REAL FLVT(4), FSTT(4), FRTT(4)

*-----species parameters
      PARAMETER (AMAX=30., EFF=0.45, KDIF=0.7)
      PARAMETER (MAINLV=0.025, MAINST=0.015, MAINRT=0.01)
      PARAMETER (SLA=0.002)

*-----dry matter distribution functions
      DATA FLVT /0.,0.85, 1.,0.45/
      DATA FSTT /0.,0.00, 1.,0.40/
      DATA FRTT /0.,0.15, 1.,0.15/

*-----initial amounts of state variables
      DATA WLV /5./, GLV /0./
      DATA WST /0./, GST /0./
      DATA WRT /5./, GRT /0./
      DATA DVS /0./, DVR /0./

*-----timer parameters
      DATA TIME /120./, FINTIM /200./, DELT /1./

*-----initialization of weather routine
      DATA STAT /'WAG'/

10    IF (TIME.LE.FINTIM.AND.DVS.LE.1.) THEN

*-----integration
      WLV = WLV+GLV*DELT
      WST = WST+GST*DELT
      WRT = WRT+GRT*DELT
      DVS = DVS+DVR*DELT
      LAI = WLV*SLA

*-----driving variable generation and rate calculation
      CALL WEATHR (40,IWAR,
     &             STAT,1983,NINT(TIME),1,
     &             LONGIE, LATIN, ALTI,
     &             TMIN, TMAX, AVRAD, RAIN, VAP, WIND)
      TMPA = (TMIN+TMAX) / 2.0

      CALL ASTRO (TIME,52.,DAYL,SINLD,COSLD)
      CALL TOTASS (TIME,DAYL,AMAX,EFF,LAI,AVRAD,SINLD,COSLD,DTGA)
      GPHOT = DTGA*30./44.

      TEFF = 2.**((TMPA-25.)/10.)
      MAINT = TEFF*(MAINLV*WLV+MAINST*WST+MAINRT*WRT)
      AVASS = MAX (0.,GPHOT-MAINT)

*-----linear interpolation
      FLV = LINT (FLVT,4,DVS)
      FST = LINT (FSTT,4,DVS)
      FRT = LINT (FRTT,4,DVS)

*-----rate calculations
      GLV = FLV*AVASS/1.4
      GST = FST*AVASS/1.4
      GRT = FRT*AVASS/1.4
      DVR = 0.02*(TMPA-12.)

*-----output during simulation
      WRITE (*,*) TIME,IWAR,WLV,WST,WRT,GLV,GST,GRT,GPHOT

*-----time updating and jump to time loop control
```

```
TIME = TIME+DELT

GOTO 10
ENDIF

STOP 'End of simulation'
END
```

11 Execution of programs

SUMMARY
- use array bound checks,
- use standard FORTRAN compiler option,
- consider the use of object libraries,
- screen debuggers are useful.

The concept of modular programming can only be fully implemented with some knowledge about the execution of programs. The execution of a program basically consists of two parts, compilation and linking. During compilation the program statements of the source text are translated into instructions of the computer's processor. During compilation, a new file is thus created called an object file. This file however does not contain all the information needed to execute the program. For instance the algorithms to compute mathematical functions and routines that are called and reside in a library (e.g. IMSL, KOMPLOT, or user- defined libraries) are not yet included in the program. These routines are attached to the object file during the linker phase, the result being an executable file.

It is not necessary to have all subprograms together in one large source text. Functions and/or subroutines can be compiled separately. They can then be linked with other routines to give an executable program or can be put into an object library. During development of a program or routine, it is advisable to execute the program from the debugger and to have array-bounds checked continuously during execution.

Standard FORTRAN compilation

Especially with compilers that recognize non-standard extensions of the language, it is important to not make use of these extensions. The portability of the source text to other machines is then maximized. Some compilers that are able to recognize extensions can be instructed to generate warnings on non-standard FORTRAN syntax.

An extremely useful program to check FORTRAN source texts in this respect is FORCHECK (see references). FORCHECK has much stronger syntax, variable declaration, argument passing and standard FORTRAN checking capacities than most compilers.

Checks on array bounds

The default compilation of FORTRAN is without checks on the (integer) value of array subscripts. It may lead to unforeseen situations if one is not aware of this. In the example below, the DO-loop that is assumed to run from I=1 to I=10 is in fact an infinite loop. The explanation for this is that IA(11) is mapped on the same memory location as I, the counter of the loop. When IA(11) is made equal to 1, also the counter of the loop is reset to 1.

```
        PROGRAM EXAMPL
        INTEGER IA(10),I,IN

        IN=10
        DO 10 I=0,IN
           IA(I+1)=1
10      CONTINUE

        END
```

A check on the array subscripts will catch these errors (see your compiler's manual how to check this).

Execution from a debugger:

The execution phase from a debugger normally brings about a number of extra commands. A debugger however is a very valuable tool to speed up program development and minimize the risk of program errors. Programs can be executed line by line with the source text displayed on the terminal, values of variables can be displayed and changed before the next program line is executed.

Working with libraries

A library is a set of programs that are kept in a file. Different types of libraries exist e.g. text libraries and object libraries. In text libraries, normal text files can be stored. In object libraries, compiled subroutines and functions can be stored. (An object library however cannot contain text files and vice versa!). Subroutines and functions that are created during program development and have been tested sufficiently can be put into an object library. The advantage is that compilation of every routine is not done each time the main program is compiled which speeds up program development and limits the costs. The routines that are put into the library are attached to the main program by the linker.

Example session using an object library

The commands in the following example apply to VAX-FORTRAN. With other compilers on other machines, the commands are different but the procedure is essentially the same.

Assume we have a main program called MAIN.FOR in which a potential evaporation rate is used. The rate is obtained by a call to the subroutine PENMAN. The program text of the subroutine PENMAN however is not in MAIN.FOR but in a separate file PENMAN.FOR. It is our intention to compile PENMAN.FOR and insert the object file permanently into an existing object library (MODULE.OLB) from which we will link it to the main program. The handling of libraries in this example is done with the VAX LIBRARY command).

```
$ FORTRAN/CHECK=BOUNDS/STANDARD PENMAN
                                       ;compiles PENMAN.FOR with array bound
                                       checks during execution.
$ LIBRARY/OBJECT/INSERT MODULES.OLB PENMAN.OBJ
                                       ;insert PENMAN into MODULES.
$ FORTRAN/CHECK=BOUNDS/STANDARD MAIN   ;compilation of MAIN.FOR
$ LINK MAIN,MODULES/LIBRARY            ;instructs the linker to link MAIN.OBJ with
                                       routines from MODULES.OLB that are
                                       called in MAIN.
$ RUN MAIN                             ;run MAIN.EXE
```

12 Examples

Example of module type 1

This LINT function is a linear interpolation function. Before returning the interpolated value, the function checks the ascending order of the X-values, and also if the X-value at which interpolation is to take place is within the range of the X- values of the data. When this happens, a message is printed on the screen that interpolation is outside the defined region. The returned value then is the Y-value of the nearest X-value.

The result from the interpolation is returned to the main program through the name of the function (LINT).

```
*----------------------------------------------------------------------*
* REAL FUNCTION LINT                                                   *
* Authors: Daniel van Kraalingen                                       *
* Version: 1                                                           *
* Purpose: This function is a linear interpolation function.  The      *
*          function does not extrapolate : in case of X below or       *
*          above the region defined by TABLE, the first                *
*          respectively the last Y-value is returned and a message     *
*          is generated.                                               *
* Keywords:Utility, linear interpolation                               *
*                                                                      *
* FORMAL PARAMETERS:  (I=input,O=output,C=control,IN=init,T=time)      *
* name   type meaning                                    units  class  *
* ----   ---- -------                                    -----  -----  *
* LINT    R4  function name, result of the interpolation   =      O    *
* TABLE   R4  A one-dimensional array with paired          =      I    *
*             data: x,y,x,y, etc.                                      *
* ILTAB   I4  The number of elements of the array          -      I    *
*             TABLE                                                    *
* X       R4  The value at which interpolation should      =      I    *
*             take place                                               *
*                                                                      *
* FATAL ERROR CHECKS (execution terminated, message):                  *
* TABLE(I) < TABLE(I-2) , for I odd                                    *
* ILTAB odd                                                            *
*                                                                      *
* No WARNINGS using the control variable IWAR are generated since      *
* nobody will check IWAR after each LINT  call ; instead an X-value    *
* below TABLE(1) or above TABLE(ILTAB-1) is reported on screen         *
* with a message containing the value of ILTAB and X. Further          *
* information on the error is not available within this function.      *
*                                                                      *
* No other SUBROUTINES and FUNCTIONS are called                        *
*  No FILE's are used (error message with WRITE(*,...)... )                *
*-----------------------------------------------------------------------*
      REAL FUNCTION LINT (TABLE,ILTAB,X)
      IMPLICIT REAL (A-Z)
      INTEGER I, IUP, ILTAB
      DIMENSION TABLE(ILTAB)
      SAVE

*     check on odd ILTAB
      IF (MOD(ILTAB,2).NE.0) THEN
         WRITE (*,'(A,I4/,A)')
     $      ' ERROR in function LINT: ILTAB=',ILTAB,
     $      ' ILTAB must be even !'
         PAUSE ' - Press RETURN to continue -'
         STOP
      ENDIF

      IUP = 0
      DO 10 I=3,ILTAB,2
*        check on ascending order of X-values in function
         IF (TABLE(I).LE.TABLE(I-2)) THEN
            WRITE (*,'(A,I4/,A,I4,A/,A)')
     $         ' X-coordinates not in ascending order at element',I,
     $         ' LINT-function contains',ILTAB,' points',
     $         ' Run deleted!'
            PAUSE ' - Press RETURN to continue -'
            STOP
         ENDIF
         IF (IUP.EQ.0.AND.TABLE(I).GE.X) IUP = I
10    CONTINUE

      IF (X.LT.TABLE(1)) THEN
         WRITE (*,'(A/A,I4,A/A,G12.4)')
     $         ' Interpolation below defined region!!',
     $         ' LINT-function contains ',ILTAB,' points,',
     $         ' Interpolation at X=',X
         LINT = TABLE(2)
         GOTO 40
      ENDIF

      IF (X.GT.TABLE(ILTAB-1)) THEN
         WRITE (*,'(A/A,I4,A/A,G12.4)')
     $         ' Interpolation above defined region!!',
     $         ' LINT-function contains ',ILTAB,' points,',
     $         ' Interpolation at X=',X
         LINT = TABLE(ILTAB)
         GO TO 40
      ENDIF
```

```
*  TMAX2 < TMIN3                                            -2              *
*                                                                           *
*  SUBROUTINES and FUNCTIONS called : ERROR                                 *
*  FILE usage : none                                                        *
*---------------------------------------------------------------------------*

      REAL FUNCTION TEMP (IWAR,
     $                    TMAX1,TMIN2,TMAX2,TMIN3,DAYL,HOUR)
      IMPLICIT REAL (A-Z)
      INTEGER IWAR
      SAVE

      PARAMETER (PI=3.14159, TAU=4.)

*     errors and warnings
      IWAR = 0
      IF (HOUR.LT.0.)  CALL ERROR ('TEMP','HOUR < 0')
      IF (HOUR.GT.24.) CALL ERROR ('TEMP','HOUR > 24')
      IF (TMIN2.GT.TMAX2) CALL ERROR ('TEMP','TMIN > TMAX')
      IF (TMIN2.GT.TMAX1) IWAR = +1
      IF (TMAX2.LT.TMIN3) IWAR = -2

      SUNRIS = 12.-0.5*DAYL
      SUNSET = 12.+0.5*DAYL

      IF (HOUR.LT.SUNRIS) THEN
*        hour between midnight and sunrise
         TSUNST = TMIN2+(TMAX1-TMIN2)*SIN(PI*(DAYL/(DAYL+3.)))
         NIGHTL = 24.-DAYL
         TEMP1  = (TMIN2-TSUNST*EXP(-NIGHTL/TAU)+
     $            (TSUNST-TMIN2)*EXP(-(HOUR+24.-SUNSET)/TAU))/
     $            (1.-EXP(-NIGHTL/TAU))
      ELSE IF (HOUR.LT.13.5) THEN
*        hour between sunrise and normal time of TMAX2
         TEMP1 = TMIN2+(TMAX2-TMIN2)*SIN(PI*(HOUR-SUNRIS)/(DAYL+3.))
      ELSE IF (HOUR.LT.SUNSET) THEN
*        hour between normal time of TMAX2 and sunset
         TEMP1 = TMIN3+(TMAX2-TMIN3)*SIN(PI*(HOUR-SUNRIS)/(DAYL+3.))
      ELSE
*        hour between sunset and midnight
         TSUNST = TMIN3+(TMAX2-TMIN3)*SIN(PI*(DAYL/(DAYL+3.)))
         NIGHTL = 24.-DAYL
         TEMP1  = (TMIN3-TSUNST*EXP(-NIGHTL/TAU)+
     $            (TSUNST-TMIN3)*EXP(-(HOUR-SUNSET)/TAU))/
     $            (1.-EXP(-NIGHTL/TAU))
      ENDIF

      TEMP = TEMP1
      RETURN
      END
```

Example of subprogram type 3

```
*----------------------------------------------------------------------*
*  SUBROUTINE PENMAN                                                   *
*  Authors: Daniel van Kraalingen                                      *
*           based on an earlier version written by: Kees van Diepen    *
*  Version: 1                                                          *
*  Purpose: This subroutine calculates potential evaporation           *
*           according to Penman (1948).                                *
*  Keywords:Simulation, potential evapotranspiration                   *
*                                                                      *
*  FORMAL PARAMETERS:  (I=input,O=output,C=control,IN=init,T=time)     *
*  name   type meaning                                    units  class *
*  ----   ---- -------                                    -----  ----- *
*  IWAR    I4  output, when .NE.0  warning !!!              -     C,O  *
*  ELEV    R4  Elevation of site                            m      I   *
*  A       R4  Coefficient of Angstrom formula              -      I   *
*  B       R4  Coefficient of Angstrom formula              -      I   *
*  ATMTR   R4  Atmospheric transmission                     -      I   *
*  TMIN    R4  Minimum temperature during day               C      I   *
*  TMAX    R4  Maximum temperature during day               C      I   *
*  WIND    R4  Average windspeed                           m/s     I   *
*  E0      R4  Potential evaporation of open water        cm/d     O   *
*  ES0     R4  Potential evaporation of soil              cm/d     O   *
*  ET0     R4  Potential evapotranspiration of crop       cm/d     O   *
*                                                                      *
*  FATAL ERROR CHECKS (execution terminated, message)                  *
*  ATMTR < 0  or  ATMTR > 1                                            *
*  TMIN > TMAX                                                         *
*  WIND < 0                                                            *
*  AVRAD < 0                                                           *
*  VAP > SVAP * 1.01 (entered vapour pressure > theor. saturated)      *
*                                                                      *
*  WARNINGS                                 value of IWAR returned     *
*  AVRAD > 40,000,000 J m-2 d-1                         1              *
*  AVRAD <    100,000 J m-2 d-1                        -1              *
*                                                                      *
*  SUBROUTINES and FUNCTIONS called : LIMIT, ERROR                     *
*                                                                      *
*  FILE usage : none                                                   *
*----------------------------------------------------------------------*
      SUBROUTINE PENMAN (IWAR,
     $                   ELEV,A,B,ATMTR,TMIN,TMAX,AVRAD,WIND,VAP,
     $                       E0,ES0,ET0)
      IMPLICIT REAL (A-Z)
      INTEGER IWAR
      SAVE

*-----Albedo for water surface, soil surface and canopy
      PARAMETER (REFCFW = 0.05)
      PARAMETER (REFCFS = 0.15)
      PARAMETER (REFCFC = 0.25)

*-----Latent heat of evaporation of water (J/kg=J/mm) and
*     Stefan Boltzmann constant (J/m2/d/K) Psychrometric
*     instrument constant (K-1)
      PARAMETER (LHVAP = 2.45 E6)
      PARAMETER (STBC  = 4.9 E-3)
      PARAMETER (PSYCON= 0.000662)

*-----errors and warnings on some input variable ranges
      IWAR = 0
      IF (ATMTR.LT.0..OR.ATMTR.GT.1.)
     &    CALL ERROR ('PENMAN','ATMTR<0 or >1')
      IF (TMIN.GT.TMAX) CALL ERROR ('PENMAN','TMIN > TMAX')
      IF (WIND.LT.0.)   CALL ERROR ('PENMAN','WIND < 0')
      IF (AVRAD.LT.0.)  CALL ERROR ('PENMAN','AVRAD < 0')
      IF (AVRAD.LT.100. E3) IWAR=-1
      IF (AVRAD.GT.40. E6)  IWAR=+1

*-----Mean daily temperature and temperature difference (Celsius)
      TMPA = (TMIN+TMAX)/2.
      TDIF = TMAX-TMIN

*-----Coefficient Bu in wind function, dependent on
*     temperature difference
      BU = 0.54+0.35*LIMIT(0.,1.,(TDIF-12.)/4.)

*-----Barometric pressure (mbar), Psychrometric constant (mbar/K)
      PBAR = 1013.*EXP(-0.034*ELEV/(TMPA+273.))
      GAMMA = PSYCON*PBAR

*-----Saturated vapour pressure according to equation
*     of Goudriaan (1977)
      SVAP = 6.11*EXP(17.4*TMPA/(TMPA+239.))

      IF (VAP.GT.SVAP*1.01) CALL ERROR ('PENMAN','VAP > SVAP')

*-----Derivative of SVAP with respect to temperature, i.e. slope of the
*     SVAP-temperature curve (mbar/K)
      DELTA = 239.*17.4*SVAP/(TMPA+239.)**2
```

change. In that way S1 represents the coupling between the two subsystems. Other state variables do not influence subprocess 2 and remain local variables inside subroutine MODEL1.

A large subroutine header has been omitted. Also warnings and errors have not been programmed into this schematic example. Integration could be blocked, for instance, without previously calculated rates of change .

```
      SUBROUTINE MODEL1 (ITASK,IOUT,IUNIT,
     $                   TIME,DELT,
     $                   .....,S1)
*     Simulates subsystem 1 described with status
*     variables STAT1 and STAT2. Uses Euler integration.

*     ITASK - input, controls action taking place
*     IOUT  - input, output control
*     IUNIT - input, unit number to be used
*     TIME  - input, global system time
*     DELT  - input, timestep
*     S1    - output, copy of local status variable STAT1
*             used for coupling with MODEL2

      IMPLICIT REAL (A-Z)
*     declaration of subroutine arguments
      INTEGER ITASK,IOUT,IUNIT
      REAL TIME,DELT,S1

*     local variables ; system description
      REAL LTIME,STAT1,STAT2,RATE1,RATE2
      LOGICAL LOUT
      SAVE

      IF (ITASK.EQ.1) THEN
*        initialize system
         STAT1 = 0.0
         STAT2 = 1.0
*        set initial rates to zero
         RATE1 = 0.0
         RATE2 = 0.0
*        initialize local time
         LTIME = TIME
*        output when enabled
         LOUT = IOUT.GT.0
         IF (LOUT) THEN
*           open output file by call to utility routine
            CALL FOPEN (IUNIT,'MODEL1.OUT','NEW','UNK')
            WRITE (IUNIT,'(A/)') ' time,   states,    rates'
          END IF
```

change. In that way S1 represents the coupling between the two subsystems. Other state variables do not influence subprocess 2 and remain local variables inside subroutine MODEL1.

A large subroutine header has been omitted. Also warnings and errors have not been programmed into this schematic example. Integration could be blocked, for instance, without previously calculated rates of change .

```
      SUBROUTINE MODEL1 (ITASK,IOUT,IUNIT,
     $                   TIME,DELT,
     $                   .....,S1)
*     Simulates subsystem 1 described with status
*     variables STAT1 and STAT2. Uses Euler integration.

*     ITASK - input, controls action taking place
*     IOUT  - input, output control
*     IUNIT - input, unit number to be used
*     TIME  - input, global system time
*     DELT  - input, timestep
*     S1    - output, copy of local status variable STAT1
*             used for coupling with MODEL2

      IMPLICIT REAL (A-Z)
*     declaration of subroutine arguments
      INTEGER ITASK,IOUT,IUNIT
      REAL TIME,DELT,S1

*     local variables ; system description
      REAL LTIME,STAT1,STAT2,RATE1,RATE2
      LOGICAL LOUT
      SAVE

      IF (ITASK.EQ.1) THEN
*        initialize system
         STAT1 = 0.0
         STAT2 = 1.0
*        set initial rates to zero
         RATE1 = 0.0
         RATE2 = 0.0
*        initialize local time
         LTIME = TIME
*        output when enabled
         LOUT = IOUT.GT.0
         IF (LOUT) THEN
*           open output file by call to utility routine
            CALL FOPEN (IUNIT,'MODEL1.OUT','NEW','UNK')
            WRITE (IUNIT,'(A/)') ' time,   states,    rates'
        END IF
      ELSE IF (ITASK.EQ.2) THEN
*        calculate rates of change ; during the calculation
*        of rates of change local time LTIME should be equal
*        to global time TIME. That could be checked here.
         RATE1 = function (LTIME,STAT1,STAT2)
         RATE2 = function (LTIME,STAT1,STAT2)
         IF (LOUT)
     $     WRITE (IUNIT,*) LTIME,STAT1,STAT2,RATE1,RATE2

      ELSE IF (ITASK.EQ.3) THEN
*        integrate ; system status a timestep DELT later
         STAT1 = STAT1 + RATE1 * DELT
         STAT2 = STAT2 + RATE2 * DELT
*        local time increase
         LTIME = LTIME + DELT

      ELSE IF (ITASK.EQ.4) THEN
*        terminal calculations (none) ; terminal output
         IF (LOUT) WRITE (IUNIT,'(A)') ' simulation halted'
      END IF

*     The coupling with MODEL2 requires STAT1 as an output
*     variable. So before leaving the routine the local
*     status variable STAT1 is copied to output variable S1
      S1 = STAT1
      RETURN
      END
```

The layout of MODEL2 is similar. There the input variable S1 is used in the calculation of rates of change when ITASK is 2. The actual coupling becomes very simple now. The calling program follows closely the structure proposed in Chapter 10 for Euler integration.

Initialization calls are done with ITASK=1. In the "program" below the value of logical OUT enables both models to write output to a file (unit numbers 40 and 41).

```
      ........
*     initialization ; set initial states and prepare
*     for a first (dummy) integration call in which
*     no states are changed
      TIME = 0.0
      DELT = 1.0
      OUT  = .TRUE.
      CALL MODEL1 (1,OUT,40,TIME,DELT,...,S1)
      CALL MODEL2 (1,OUT,41,TIME,DELT,S1,...)
```

Now the system is prepared for the integration loop. A problem here is that N timesteps require output at N+1 times (initial output plus output after each step). Further, the output statements should be located in such a way that rates of change belong to the current status and not to the previous one. A simple solution was used already in Chapter 10. At first the integration calls are done, then driving variables and rates of change are calculated. Using such a loop, however, the first integration call should be a dummy one. No status variables should be changed then, for instance, by setting rates of change to zero during initialization calls (see MODEL1 above).

```
*     integration loop
      IF (TIME.LE.FINTIM) THEN
*        integration ; get system into new status
         CALL MODEL1 (3,OUT,40,TIME,DELT,...,S1)
         CALL MODEL2 (3,OUT,41,TIME,DELT,S1,...)

*        generation of driving variables
         CALL WEATHR (......)

*        rates of change for subsystem 1
         CALL MODEL1 (2,OUT,40,TIME,DELT,...,S1)
*        rates of change for subsystem 2 using S1 as input
         CALL MODEL2 (2,OUT,41,TIME,DELT,S1,...)

*        output of combined model
         .......

         TIME = TIME + DELT
      END IF

*     terminal section
      CALL MODEL1 (4,OUT,40,TIME,DELT,...,S1)
      CALL MODEL2 (4,OUT,41,TIME,DELT,S1,...)
      ..........
```

A few further remarks are made now. The example MODEL1 is kept simple. In practice a few groups of subroutine arguments will occur. There are (time) control variables like ITASK, OUT , IWAR and DELT. There are input variables used during initialization only. The initialization call may produce results as well, certain constants for instance, needed in other modules. And finally the dynamic calls (ITASK=2,3) will require input and output variables.

In MODEL1, the status variable STAT1 itself was not returned to the calling program. Instead, its value was copied into an argument. That provides an easy way of adapting the program to other coupling requirements. A subroutine argument can be added or removed without the need to change the (carefully written) system status declarations etc. Further, checks are possible against external system changes.

A simulating subroutine may be the top level subroutine of a whole group. Within such a group of subroutines complicated structures may exist, large common blocks may be used, etc. These structures should be left intact as much as possible. They can be maintained then by the author of the model. It is important, however, that the top level routine is simple. It should be easy to use for people who cannot go into the details of the model. Preferably, only a number of calls are to be included in the calling program. Datafiles can be read from the initial section of the model. The need to include common blocks almost certainly leads to erroneous model use.

The integration method used inside each of the two (or more) simulating subroutines does not need to be the simple Euler method. During the integration call the Runge–Kutta method, for instance, could be used to simulate the subsystem over a time DELT. That method involves repeated rate calculations usually programmed into a separate subroutine. Omission of the first integration call is easily realized then by means of a logical variable, set during initialization. And the rate calculations for ITASK=2 are done for the current status of the subsystem. So the coupling method described above implies only that the interaction (the feedback from the other subsystems) takes place at regular time intervals.

The above method is easily applicable when all subsystems are simulated with equal timesteps since there are no specific coupling problems. When the subsystems determine their own, local, timesteps the coupling timestep can be held fixed when the interaction is weak. Otherwise more elaborate schemes are required or a complete 'fusion' of the models should be carried out.

References and Further Reading

Cate, J.A. ten, Akkersdijk, J.W.J., 1988. CABO FORTRAN kursus 1988.

FORCHECK: a FORTRAN verifier and programming aid, Erik, Kruyt, Vagroep Fysiologie and Fysiologische Fysica. Leiden University. The Netherlands.

Haar, L.G.J. ter, 1983. FORTRAN 77, programmers pocket guide. Nederlands Normalisatie Instituut. 47 pp.

Hahn, B.D., Problem solving with FORTRAN-77. Edward Arnold Ltd. London. 247 pp.

Leffelaar, P.A., Wolbeer, E.W. Dierkx, R.T., 1986. Some hints to write more readable simulation programs by the combined use of CSMP and FORTRAN-subroutines. Simulation Reports CABO-TT no. 9. 34 pp.

Meissner, L.P., Organick, E.I., 1984. FORTRAN 77, featuring structured programming. Addison-Wesley publishing company. 500 pp.

Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T., 1986. Numerical Recipes, the art of scientific computing. Cambridge University Press. 818 pp.

Tiktak, A., Programmeerrichtlijnen voor deterministische, dynamische simulatie-modellen. Fysisch Geografisch en Bodemkundig Laboratorium. Amsterdam. 24 pp.

Wagener, J.L., 1980. FORTRAN 77, principles of programming. John Wiley & sons. New York. 370 pp.