Mitteilungen aus dem
Institut für Pflanzenökologie
der Justus-Liebig-Universität
Gießen

Heft 3

A Modular Structure for Models of Time-Evolving Systems

Johannes Hoffstadt
Gerd Esser

Justus-Liebig-Universität
D-35392 Gießen

Mitteilungen aus dem Institut für Pflanzenökologie der Justus-Liebig-Universität Gießen Gerd Esser (Series Editor)

Volume 3
Hoffstadt, Johannes; Esser, Gerd:
A Modular Structure for Models of Time-Evolving Systems.
Gießen 1997.

Address of the Authors:
Dipl.-Phys. Johannes Hoffstadt (`hoffstadt@bio.uni-giessen.de`)
Prof. Dr. Gerd Esser
Institut für Pflanzenökologie
Justus-Liebig-Universität
Heinrich-Buff-Ring 38
D-35392 Gießen.

ISSN 0946–2201

This document is a result of work which was sponsored by the German Ministry for Research and Technology under the Climate Programme, and the Commission of the European Union under the EPOCH and ENVIRONMENT programmes.

Contents

Preface

Purpose of this document. This paper documents a modular structure to be used for models that describe systems evolving in time. It is also intended as a manual for the modification of existing models. The structure has been implemented as a computer program. The gentle reader is invited to contact the authors for code and example material. Depending on the demand, introductory courses can be arranged. Please contact us using the following address:

```
Johannes Hoffstadt
Institut für Pflanzenökologie
der Justus-Liebig-Universität
Heinrich-Buff-Ring 38
D-35392 Gießen
Germany
```
E-mail: `johannes.hoffstadt@bio.uni-giessen.de`

Software requirements. For the current implementation, a Unix operating system, a FORTRAN 77 compiler and an ANSI C compiler is required. The code has been tested on the following platforms (i. e., hardware/software combinations):

machine type	processor / speed	operating system
No-Name PC	PentiumPro / 200 MHz	Linux 2.0
DEC 3000 AXP	Alpha RISC	OSF1 3.2
HP Apollo 735	PA 7100 / 99 MHz	HP-UX 9.0
IBM RS/6000	Power2 / 66 MHz	AIX 3.2
SGI Power Challenge	4 MIPS R8000 / 75 MHz	IRIX 6.1
SUN ES 3000	4 UltraSPARC / 250 MHz	SUN OS 5.1.1
Cray C90	16 vector processors	UNICOS 9.0
Fujitsu/SNI VPP300/6	6 vector processors	UXP/V 10

The code is equally well suited for scalar and vector computers. We did not try to adapt the code to parallel computing, because there is no standard language extension for parallel computers that works on several platforms, and can still be used on scalar or vector machines. Porting the code to standard PC systems is possible with some loss of minor features.

Programming languages. The core of the structure has been implemented in C. We have worked successfully with modules written in FORTRAN and C.

Other compiler languages can also be used for modules, if they support access to shared variables (like FORTRAN common blocks), and if the created object files contain the entry addresses of the modules.

We have chosen FORTRAN and C, because they offer shared as well as local variables, they are widely used, and the generated code is efficient. This combination is especially useful when running models on machines with a Unix operating system, like most state-of-the-art compute servers. They provide a C-oriented environment with easy access of system and graphics libraries, and FORTRAN for optimal speed. The compilers are generally prepared for mixing object code of both languages.

1 Introduction

While our knowledge about the terrestrial part of the global carbon cycle has increased in the last few years, concepts and questions have accumulated even quicker. Many processes are considered important, and we have to handle processes on rather different time scales within one model.

Accordingly, models become increasingly complicated and difficult to manage. Each model is unique in its philosophy and implementation, which is a major obstacle in model comparison. This is because the overall performance of models does not help to attribute differences in model behaviour to particular assumptions. True understanding can probably be achieved only on the level of underlying processes. To investigate the role of different mathematical formulations of a particular process, we need to test various descriptions of a single process within the framework of one model.

A modular model structure is therefore required, which supports easy exchange of different versions of a process module without changes to the rest of the model.

The modular structure as presented in this document has been designed for arbitrary models that describe systems evolving in time, typically using differential or finite difference equations. A central concept is the linear flow of time, during which the system state evolves continuously; interrupted only occasionally by the execution of modules. These module calls are due to a given schedule, and due to dependencies between modules; they may influence the dynamics by updating coefficients, or may simply output the current system state.

Generally, the system state must be treated and propagated as a whole. Regarding the special case of models of the global carbon budget, a common simplification has been to subdivide the system state vector into state vectors of the grid elements, which are then treated independently. Coupling of grid elements is then impossible.

In contrast, our structure strives to be generally applicable and must be open to any coupling between elements of the system state. Adaptation of the abovementioned models therefore involves a transformation of the code, so that the grid elements are always synchronized, and their state corresponds to the same current time.

What you will gain: The proposed scheme facilitates not only the exchange of modules, but also the coupling with other models. As a byproduct, your code will be suitable either for parallelisation or for vectorisation on supercomputers.

What you will lose: Any optimisation may be lost, that relies on modeling one grid element after another, if pre-calculated results cannot be stored for all grid elements anymore. This applies especially when bringing the system into equilibrium for some constant year, and can increase CPU time dramatically. On the other hand, the demand for CPU time for transient model runs will roughly stay the same. Memory requirements are bound to increase depending on the number of grid elements.

Consequently, one can employ this structure only with some reserves in CPU time and memory left.

2 The modular model structure

2.1 Modules and interfaces

A module is a subprogram that performs a specific task. For its task it usually needs some input data and produces some output data (or takes other actions, according to the particular task). Input and output together form the interface definition of the module. Task and output are closely connected, and determine if modules are exchangeable (given that we provide the necessary input).

Otherwise, a module is an entirely independent entity. It can be used in any context, and will simply do its job. The inner functioning does not matter. Modules can be treated as black boxes.

A complete interface description of a module consists of stating the input and output data structures, and it is accompanied by the task description. If modules from different workers should be interchangeable, then there must be some agreement on the layout, interpretation, and physical units of these data. Documentation on this is part of the interface specification.

Input and output data can be internal (shared variables) or external (files). Therefore, one has at least three classes of modules: initialisation modules reading input data from files; processing modules that perform calculations on internal data; output modules writing results or diagnostics to files.

Models usually have some kind of spatial resolution, e. g. a surface grid or box-shaped cells in a volume. Internal data per element are most conveniently stored in arrays. Modules operate on whole arrays. This keeps the state of the elements synchronized, and is necessary for exchange of modules and coupling of submodels.

2.2 The configuration file

A guiding idea in the design has been to be able to 'switch' modules on or off as required by a particular model run. There should be every conceivable freedom in the choice of the time points and/or intervals when a module should be called. Imagine a singular historical event that influences the system through an input pulse, like ^{14}C production by an atomic bomb test, or imagine a period of time with a different radiative forcing due to volcanic eruptions, after which the driving forces return to normal values. These examples could be handled by modules that are called once at a specific point in time, or regularly during a time interval.

Also, if there are dependencies between modules, such as that module B always requires module A to run beforehand, it should be possible to state them directly, rather than having to 'hard-code' the calling sequence. Finally, one should be able to control parameters of a model run without re-compiling. These ideas led us to the formulation of a configuration file.

The configuration file describes a model run completely, by containing instructions for a unique, reproducible sequence of modules, and all relevant control parameters.

2.3 The master module

A master module is required which interprets the configuration file, resolves given dependencies, sets control parameter values, and creates and executes the calling sequence.

Obviously, the master module needs to know about the memory address of the entrance point of each module that is linked into the model. It also needs to know the location of each variable that can be set through an entry in the configuration file in order to transfer the value. Lists of pointers to memory addresses are not supported by standard FORTRAN. Thus, the master module is written in C. This is no disadvantage for the execution speed, since no lengthy calculations are necessary. Apart from address information, which is provided by separate files, the master module is independent from the particular model.

2.4 The concept of time

The usual loop structure of models (Fig. 1) consists of (possibly hierarchical) time loops, and subroutine calls at the beginning and end of a loop. The passage of time happens at the center of the loops. For the modular structure, the above arrangement of loops and calls had to be transformed into a clear concept of time, which is: *Module calls become associated with points in time. The continuous flow of time, realized by integration, is interrupted according to the schedule of module calls. The alternating calling and integrating is controlled by the master module.*

One may view this mechanism as some kind of alarm clock, which activates modules according to a schedule during the otherwise undisturbed passage of model time. For model time, one can choose between a simple calendar with 30-day months and 360-day years, or a better calendar with 365-day years (but no leap years). 10^9 years of model time can be spanned, and the resolution of time points is one second. However, this can be easily adapted to different needs.

```
c     some initialisation           e. g. reading input data
      do 10 iannum = start, end     for every year:
         call annprp                  annual preparation
         do 20 imonth = 1, 12         for every month:
            call monprp                 monthly preparation
c           some inner time loop        (containing integration)
            call outmon                 output at end of month
20       continue
         call outann                  output at end of year
10    continue
```

Figure 1: An example for the usual loop structure of models. Subroutine calls are placed at the beginning or end of a loop. Time actually passes in the innermost loop.

Two-faced time points. We want to be able to call modules either at the beginning of an integration interval (e. g. at the beginning of a month), or at the end of the interval (e. g. for output of the system state at the end of a month). This cannot be described using single time points. Rather, we have to split a time point in two adjacent parts, one of which faces the end of the previous interval, the other faces the beginning of the next interval. We speak of module calls that happen just before a given point in time, or at ('just after').

For example, one can draw the sequence of module calls according to the loops in Fig. 1, and add a horizontal time axis. This is shown in Fig. 2 for the beginning of the model run (first line), and for the end of each year (second line).

The model run begins *at* midnight, January 1st of the first year with the initialisation. The annual preparation happens *at* January 1st of each year, the monthly preparation happens *at* the first of each month. Then comes time integration, which spans an interval, and happens between two points in time. Afterwards, current time is *just before* the first of the next month. This is the point where the monthly output has to be called: *just before* the first of the next month. And similarly, annual output has to be called *just before* January 1st of the next year. Lastly, the model run itself ends in the final year, *just before* January 1st of the following year.
When the calling list for the current time point has been worked through, the master module has to calculate the time interval until the next call of a module. This interval can be of varying length, depending on the superposition of calling 'rhythms' of the modules. If the interval is not zero, the master module calls a special integrator module, which propagates the system state through this time interval.

Time integration. The state of the system is represented by one or more arrays of variables. Models using finite differences add the calculated changes to the state variables. With continuous models, an integrator has to solve the system of first-order differential equations that represents the changes of the state variables. A numerical integration method is preferred, because only a few models can be solved analytically.

We have equipped our model with a numerical solver of differential equations, using the 4th order Runge-Kutta method with fixed time steps, *but any*

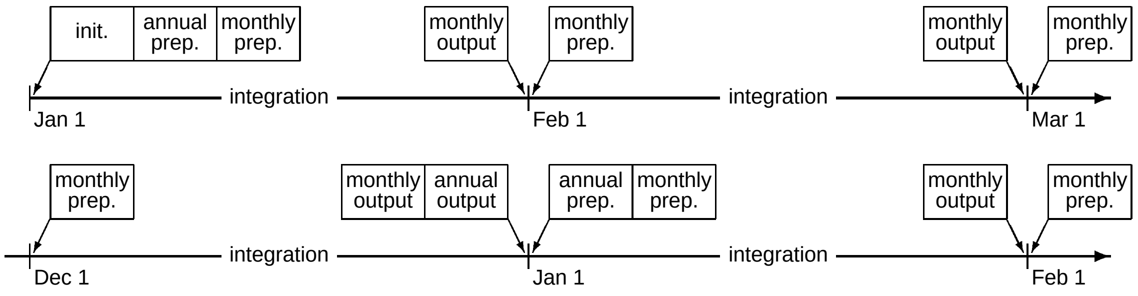


Figure 2: Calling sequences of modules as they interrupt the passage of time. Module calls happen just before a time point (indicated by the date), or just after.

other method can be used instead. As with all modules, the integrator treats all cells in parallel, not sequentially, with the side-effect that temporary results can occupy large amounts of memory (since always the whole arrays are needed). The chosen method needs only two temporary results per state variable and cell, which we consider the best compromise regarding speed (due to vectorisation) vs. memory usage.

Adaptive time steps and predictor-corrector methods could speed up the integration of individual grid cells with less pronounced dynamics, but this spoils vectorisability (which depends on parallel treatment of lots of cells).

The system of differential equations. The numerical solver needs to evaluate the left-hand sides of the differential equations, which give the derivatives (or 'slopes') of the state variables with respect to time. These differential equations are conveniently grouped within a second special module. Only the integrator calls this module, and the integrator itself is called only by the master module (because the master 'clockwork' controls time). These three modules are special in that they cannot be specified in the configuration file like ordinary modules.

2.5 Files involved, and their relationships

The master module is contained in the file `master.c`, which includes definitions from `master.h`. Information about the particular model is included from three files:

- `variables.h` contains a table of variables that can be set within the configuration file. Those variables commonly are: switches for the run-time behaviour, file names for input and output files, settings for the master module itself.
- `modules.h` contains the declaration and the table of all modules.
- `data.h` contains the declaration of all shared data between modules.

The file `variables.h` can easily be modified for a particular model. This is described in section 4.6. You are not supposed to write the files `modules.h` and `data.h` manually, but use a generator as described below.

One can simply compile the master module and the modules of a particular model and link them together. But there are some system-dependent problems which require specific compiler flags. Therefore, we have provided a rather generic `Makefile` containing settings for various systems. In the `Makefile`, one has to modify the list of file names that are to be compiled.

We recommend that you first try out the example model and make sure that it works, as described in section 3. In the case of problems, we can probably help if you provide access to your machine.

Automatically generated files. We have provided an auxiliary C program called `autogen`, that generates several include files, including `modules.h` and `data.h`, and an initialisation module `init.f`, from very simple text files. All automatically generated files are created in the subdirectory `auto`, where the master module expects to find these files.

We did this for two reasons: first, information about modules and shared variables should appear only once, not several times in identical copies, where consistency can become difficult to ensure; secondly, the file `modules.h` consists of several sections with different structure, which appear obscure for people without knowledge of C.

The input files used by `autogen` are

- `common.def`: a list of shared (common) variables, including type and dimension(s);
- `const.def`: a list of constants (e. g. for array dimensioning, or array access), including type and their FORTRAN and C value (which can be different);
- `modules.def`: a list of modules, including argument types (which is usually '`void`'), and optional pre-processor symbols that can be used for conditional compilation.

The layout of these files is described in sections 4.3, 4.4 and 4.5. Using them, `autogen` creates the following files in the sub-directory `auto`:

- From `common.def`, two include files per entry (i. e., per variable) are generated. One is the header file with a C-style declaration of the shared variable. The other file contains the FORTRAN common block definition, where the common block has the same name as the variable. The filenames also correspond to the name of the variable and end with `.h` and `.f`, respectively.

  Additionally, a file `data.h` is created, which contains `#include` statements for all `.h` files, and enables the master module to 'see' every shared variable. Even if you do not have C modules that use data from the FORTRAN parts, you might have an easier time when debugging on some machines.

  Finally, a new module `init.f` is created which contains initialisation code for all variables. Numeric variables are set to zero, character arrays are set to empty strings. This module is not callable by the configuration file, for the simple reason that it effectively disrupts any model run, and deletes any assignments in the configuration file. It is optionally called by the master module at the beginning (see appendix A), before the configuration file is parsed. Afterwards, one can safely assume that all variables have initial values. Alternatively, one can modify this module to set variables to some special values that are easily spotted (like `88888888` or `NaN`), in order to detect forgotten assignments.

- From `const.def`, two include files are generated: `arrays.h` contains C pre-processor definitions of the given symbols, using the C values, while `arrays.f` contains FORTRAN parameter definitions with the FORTRAN values. Every module should include these array settings at its top.
- From `modules.def`, the file `modules.h` is generated.

3 The provided 'package' and its use

We have put together a package, `modular.tar`, containing the parts described in the previous section. This file can be unpacked using the Unix command

```
tar xf modular.tar
```
,

which creates the following directory structure in the current directory:

`modular/doc/`	contains this documentation in printable form
`modular/conf/`	contains a no-op model with empty modules for tests of configuration files (see section 5.5)
`modular/kbm/`	the example model, which is discussed below
`modular/tests/`	contains a simple test for compatibility of FORTRAN and C, discussed in appendix B.1

The following refers to the example model, that can be used as a template. It comes complete with initialisation and output modules, the special differential equations module, a time integration module using the fixed step 4th order Runge-Kutta method, the master module, written in ANSI C, and some configuration files.

3.1 The example model

The example model is a model of the global carbon cycle on a yearly basis, which is used for courses in system analysis for graduate students, and is known as 'Kurs-Biosphärenmodell', or short: KBM. It uses a 10° by 10° grid for the terrestrial biosphere (158 grid elements, four compartments), and contains a one-dimensional ocean submodel (Oeschger et al., 1975, *A box diffusion model to study the carbon dioxide exchange in nature*, Tellus 27, 168–192). It needs about 150 KB RAM. The directory structure is as follows:

`kbm/`	the uppermost directory, which is the working directory for model runs; contains config files
`kbm/input/`	contains input files (read-only)
`kbm/output/`	can be used to store output files
`kbm/src/`	contains the source code of the modules and a Makefile
`kbm/src/auto/`	contains automatically generated include files

Some conventions. We use standard FORTRAN 77 with the extension of lower-case code and `include` statements, supported by all compilers that we know of. FORTRAN source code may not contain platform-dependent code. (C source can, but must use conditional compilation.) All variables must be declared, but we don't use `implicit none`, since not all compilers support this non-standard directive. We use compiler flags instead (e. g. `-u` on several systems). We also don't use `enddo`, but numerical labels on `do` loops.

All floating-point variables in the model code should be declared as `real`, except for those where you need at least double precision. Most compilers have switches to promote single precision to double precision. For C variables, we use a preprocessor macro named `real`, which is defined to either `float` or `double`. This way we have kept the decision between single and double precision for both languages in one place, the `Makefile`. Shared variables are accessed from C by appending an underscore to the name.

Files. From the file `modules.def`, `autogen` builds the table of modules that is needed by the master module. The generated file is `auto/modules.h` (listed in appendix B.4). The modules of the KBM are given in Table 1. Each module is kept in a separate file with the same name. Those files are:

arcalc.f	de.f	ocean.f	plot.f	cglosum.c
cld.f	filred.f	outann.f	prread.f	xdview.c
clp.f	glosum.f	peq.f	prwrit.f	
cnpp.f	idummy.f	pinit.f	timint.f	

Task and interface description are included as comments. The modules use `include` to access the constant dimensions required to declare the arrays (from the generated files `arrays.f` and `arrays.h`), and the declarations of the shared variables. The included files exist twice per variable, as FORTRAN-style and as C-style declarations, and are also automatically generated:

`common.def` is used to produce include files for the shared variables as seen from FORTRAN and from C. The variables are given in Table 2 on page 11. Additionally, the files `auto/data.h` and `init.f` are created as described above.

From `const.def`, the files `auto/arrays.f` and `auto/arrays.h` are produced. The constants defined are:

(1)		(2)		(3)	
ngrid	158	ph	1 (0)	atm	5 (4)
npool	8	pw	2 (1)	npp	6 (5)
nbiome	17	lh	3 (2)	lp	7 (6)
		lw	4 (3)	ld	8 (7)

Constants in column (1) define sizes for dimensioning array variables. Columns (2) and (3) define FORTRAN index values (C index values in parentheses).

This is used for consistent access: we try to avoid 'magic numbers' in the source and use symbolic names instead, meaning that we write `pool(i, lw)` to get the pool 'litter, woody' on grid element `i`. Later, we can expand and rearrange the pool array without restrictions; if we choose to have the woody litter at some other index than 4, only this file has to be changed. The same is true for model runs with different resolution and therefore a different number of grid elements

Table 1: Modules used in the KBM (in alphabetical order).

`arcalc`	calculates area of grid elements based on latitude of their center
`cglosum`	C module that calculates global sums of pools and fluxes
`cld`	calculates litter depletion coefficients based on climate
`clp`	calculates litter production coefficients based on stand age
`cnpp`	calculates NPP
`de`	calculates time derivatives of the system state (does not appear in `modules.def`, since only `timint` needs to know)
`filred`	reads input data from files
`glosum`	FORTRAN version of `cglosum`
`idummy`	sets pools to zero, which are used for integrating annual fluxes
`infnan`	is used to test the treatment of exceptions, and produces overflow, not-a-number, and illegal memory access
`ocean`	calculates CO_2 uptake by the ocean and the atmospheric CO_2 concentration, using an ocean sub-model that contains its own time integration (Oeschger et al., 1975)
`outann`	writes global pools and annual fluxes to standard output
`peq`	calculates the equilibrium system state
`pinit`	sets the system state to zero (array initialisation)
`plot`	creates maps of data during a model run. Maps are displayed in X11 windows (using C functions defined in `xdview.c`)
`prread`	reads a previously saved system state
`prwrit`	writes the current system state to a file
`timint`	numerical integrator for the system of differential equations

— just change `ngrid` (and the input files, of course). An additional advantage of having such parameters as loop bounds comes from optimisation. Compilers know in advance if loop unrolling is useful, or which vector length to use on vector hardware.

The file `variables.h` is given in appendix B.5. It contains the definitions of variables that are required by the master module itself (see section 4.6), the initial value of the atmospheric CO_2 concentration, and some filename variables.

3.2 Compiling the model

Change to the source directory and enter the Unix command

```
make
```

which will examine the rules and definitions given in the file `Makefile`. The environment variable `HOSTTYPE` is used to decide which platform-specific instructions should be applied. If the variable is not yet set, you will see instructions how to do this.

These are the currently recognized `HOSTTYPE` codes along with the tested platforms:

`aix`	IBM RS/6000 running AIX 3.2
`cray`	Cray C90 running UNICOS (vector-parallel)
`dec`	DEC Alpha running OSF1

`hp-ux`	HP Apollo 735 running HP-UX 9.0
`linux`	No-name PPro 200 running Linux 2.0 and g77
`sgi`	SGI Power Challenge running IRIX 6.1
`sun`	SUN Enterprise Server 3000 running SunOS 5.1.1
`vpp`	Fujitsu/SNI VPP300 running VPUX 10 (vector-parallel)

`make` performs the following: building the automatic generator `autogen` from its C source, then generating the include files, then compiling the modules and linking the object files into an executable. Single precision (`REAL*4`) is the default. For most platforms, there is also a double precision target (`REAL*8`). For this, say '`make double`' instead of '`make`'. Integer size is not affected. Note that not all platforms support both: on Linux, `g77` lacks an option for automatic promotion to double precision. And the Cray vectorizes operations only on double precision data.

We use our HP machines for development and short model runs, and the vector computers for large model runs. Compiler settings are rather elaborate on these platforms. On the other platforms, the model has been made to work, but just with standard optimisation. If you use different platforms than the above, you will have to edit the Makefile. This is discussed in appendix B.

3.3 Running the model

Change to the `kbm` directory. Just type `kbm` to get a short summary how to use the program. Essentially, you have to specify at least the configuration file that you want to use. Eight configuration files are provided:

`kbmpre.cfg`	The model starts with empty pools and runs for 860 years to reach equilibrium state, which is written to a file.
`kbmpost.cfg`	The model starts from the previous results and runs from 1860 to 1986 with fossil emissions and ocean turned on.
`kbmeq.cfg`	Same as before, but calculating equilibrium directly. This is a lot faster than combining the first two.
`kbmeqc.cfg`	Like `kbmeq.cfg`, but the FORTRAN module `glosum` is exchanged for an equivalent C module, `cglosum`. Use this to test array access (especially with double precision) from C.
`kbmplot.cfg`	Like `kbmeqc.cfg`, but adds two X11 windows showing online maps of NPP and NEP. Just a little gimmick. Check that your `DISPLAY` variable is correctly set.

For instance, use the command

```
kbm kbmeq.cfg
```

to see the development of the global carbon pools during the industrial period, ending with about 334 ppm in December 1986.

If you want to examine the calling of the modules, you can use option `-v` during the run, or `-n` if you just want to see the sequence without actually calculating anything. You can also activate a stop watch for the modules: option `-t` shows the time needed for each module call, and at the end a summary is shown as a table.

Table 2: Variables shared between modules of the KBM (for the constants used for array dimensioning see text). NPP: net primary production; LP: litter production; LD: litter depletion.

input data		
`real`	`t(ngrid)`	annual mean temperature (° C)
`real`	`pp(ngrid)`	annual sum of precipitation (mm)
`real`	`fsoil(ngrid)`	correction factor for NPP from soil quality
`integer`	`ibiome(ngrid)`	biome code (1–17)
`real`	`lat(ngrid)`	latitude of SW corner of grid element (°)
`real`	`lon(ngrid)`	longitude of SW corner of grid element (°)
`real`	`ageh(nbiome)`	mean stand age (herbaceous material)
`real`	`agew(nbiome)`	mean stand age (woody material)
`real`	`h(nbiome)`	share factor of NPP to herbaceous material
`real`	`fossc(1860:1986)`	C emissions from fossil sources (Gt C/year)
intermediate results		
`real`	`area(ngrid)`	area of grid element (m^2)
`real`	`npph(ngrid)`	share of NPP to herbaceous material
`real`	`nppw(ngrid)`	share of NPP to woody material
`real`	`klph(ngrid)`	LP coefficient for herbaceous material
`real`	`klpw(ngrid)`	LP coefficient for woody material
`real`	`kldh(ngrid)`	LD coefficient for herbaceous material
`real`	`kldw(ngrid)`	LD coefficient for woody material
`real`	`pool(ngrid,npool)`	system state (g C/m^2)
`real`	`pglobal(npool)`	global sums (g C)
`real`	`co2`	atmospheric CO_2 concentration (µl/l)
`real`	`mixsm`	C stored in mixed ocean layer since 1860 (g C)
`real`	`deepsm`	C stored in the deep sea since 1860 (g C)
current time information		
`integer`	`iannum`	current year
`integer`	`imonth`	current month
`integer`	`idoy`	current day of year (doy)
`integer`	`iday`	current day
`integer`	`ihour`	current hour
`integer`	`iminut`	current minute
`integer`	`isecon`	current second
file names		
`string`	`fgrid`	input file for data related to grid elements
`string`	`fbiome`	input file for data related to biomes
`string`	`ffoss`	input file for yearly emissions from fossil C
`string`	`fpoolr`	file to read system state from
`string`	`fpoolw`	file to write system state to
time integration control		
`integer`	`maxdt`	maximum step size (sec)

4 Using the structure with your own model

Now we are going to show the steps one would have to take in using our package with an existing model.

4.1 Prerequisites

You have to have a partly modular model to begin with. Ideally, your code should consist of several subroutines, each one dedicated to a particular task, and a main program that coordinates the calling. What you have to do is to shift all inter-module communication down to the subroutine level, treat grid elements in parallel, and to remove most, if not all, knowledge of model details in the main program. Then you can replace the main program by the master module.

4.2 Change subroutines to modules

For every subroutine, take down its interface definition. What is the task of the subroutine? What does it expect as input data? What is its output? It is useful to add this information as commentary lines to the code. Look at the arguments that the subroutine is called with, if there are any. All of them should be accounted for in the interface description.

If your main program contains other loops (e. g. grid element loops) than time loops, change this to have those loops down in the subroutines. — This is usually the main part of the work. It also has the disadvantage of having to store intermediate results (that are used by the next module, for instance) for all grid elements in arrays, which is memory-intensive. As already mentioned in the introduction, you will lose much of the code optimisation, that relied on modeling one grid element at a time, with precalculating invariant expressions, or whole time courses of fluxes or driving forces; since in general you won't be able to store the precalculated information for all the grid elements at once.

When you are clear about the interfaces of all subroutines, each item (array or scalar) of input or output data has to be noted in the file `common.def` (see below), from which common block and C header declarations are generated later. Still, you have to fill your subroutines with the necessary `include` statements to gain access to the data.

In the end you should have subroutines that are called without parameters, and that share their data according to their interface specification without the main program seeing anything of this. These subroutines will be called modules from now on. The main program then contains only calls to modules and the loop structure for time evolution of the system, and probably some initialisation of variables at the beginning.

4.3 Create a list of shared variables (`common.def`)

This file may contain comments (starting with '#') and empty lines. Other lines are taken as entries for variables and have the form

Variable *Type* [*Dimension-1* [*Dimension-2* …]]

Variable is a valid name for FORTRAN and C variables. *Type* is a valid FORTRAN type. It may not contain blanks. If you use '`real`', '`integer`', or '`string`' (meaning `CHARACTER*80`), `autogen` will produce a corresponding C declaration, otherwise you cannot use this shared variable in C modules.

No dimensions are given for scalar variables and for strings. Otherwise, *Dimension-n* is a valid expression for dimensioning, and may not contain blanks or tabs. Example (excerpt from the KBM):

```
# name   type    dimension(s)
# -------------------------
area     real    ngrid
fossc    real    1860:1986
pool     real    ngrid       npool
iannum   integer
fgrid    string
```

leads to, for instance, the file `auto/fossc.f` containing

```
c     auto/fossc.f (generated by autogen)
      real fossc(1860:1986)
      common /fossc/ fossc
```

and `auto/fossc.h` containing

```
/* auto/fossc.h (generated by autogen) */
#ifdef UCASE_OBJECT_NAMES
#define fossc_ FOSSC
#endif
#ifdef LCASE_OBJECT_NAMES
#define fossc_ fossc
#endif
EXTERN real fossc_[1986-1860+1];
```

which is explained in appendix B.

4.4 Create a list of constants (`const.def`)

This file may contain comments (starting with '#') and empty lines. Other lines are taken as entries for constants and have the form

Constant   *Type*   *FORTRAN-Value*   *C-Value*

Constant is a valid name for FORTRAN and C variables. *Type* is either '`real`' or '`integer`'. *FORTRAN-Value* is the numerical value to be used for `arrays.f`, *C-Value* is the value to be used for `arrays.h`. The C value will be equal to the FORTRAN value if it gives the size of the arrays. It will be one less if it is used as array index.

Example from the KBM:

```
# name  type    fvalue  cvalue
npool   integer    8       8
ngrid   integer  158     158
nbiome  integer   17      17
ph      integer    1       0
pw      integer    2       1
```

4.5 Create a list of modules (`modules.def`)

This file may contain comments (starting with '#') and empty lines. Other lines are taken as entries for modules and have the form

Module *Type* [*Switch*]

Module is a valid name for FORTRAN and C subroutines. *Type* is the argument type in C notation, and may not contain blanks. *Switch* is an optional preprocessor symbol, which has to be defined in order to include this module.

The sequence of entries does not matter. There is no limit to the number of modules, since the array is dimensioned at its initialisation and therefore has always the correct size.

Example from the KBM:

```
# name  arg     preprocessor-switch
arcalc  void
cnpp    void
filred  void
timint  int*
plot    void    __WANT_X11__
```

Ordinary modules have no arguments (the type is '`void`'). The only exception is `timint`, which takes the number of seconds of integration as a pointer to `int`, since it is a FORTRAN subroutine, where arguments are passed by reference, not by value.

The module `plot` requires the X Window System libraries, and is only included if the pre-processor symbol `__WANT_X11__` is defined during compilation (see the `Makefile` of the KBM).

From `modules.def`, the file `auto/modules.h` is created, which provides the address information about all modules for the master module. No corresponding FORTRAN file is necessary.

4.6 Create a list of variables (`variables.h`) for initialisation purposes

`variables.h` contains a table of variables that can be set from the configuration file. What variables are these? They are typically those that were initially assigned in the former main program, like switches that control model behaviour, or simply some initial values. They have to be *scalar* variables.

You cannot initialize arrays from the configuration file; for this, one has to use modules (like reading driving forces from files, or reading a temporarily saved system state for continuing a model run).

It is also possible to put all remaining assignments for scalar variables into a 'parameter file', that is read by some module at the start of the run. On the other hand, it is desirable to have some assignments located in the configuration file for a particular run, so that all the information necessary to reproduce the run remains visible in one place, especially when one considers switches, or input/output file names.

You cannot completely omit variables, since the master module and the time integration requires some variables to be set (these entries are discussed below). The layout of the file is rather complex, since it is a complete C declaration file. An example from the KBM is given in appendix B.

The first fixed part of the file looks like this:

```
struct {
   char  name[80];
   int   type;
   int   required;
   void *addr;
} variable[] = {
  { "", 0, 0, NULL },
```

This defines a structure that is made up from four components:

- `name`: a name of a variable, up to 80 characters long. As with modules, there needs to be a mapping from a string name to the actual memory location of the variable. The string name may not contain the five characters `:=<>"` but is otherwise unrestricted.
- `type`: an integer that represents a data type, where the valid data types for variables are defined as symbolic constants in the file `master.h`. They are:

`INTEGER_TYPE`	meaning `INTEGER`,
`REAL_TYPE`	meaning `REAL`,
`STRING_TYPE`	meaning `CHARACTER*80`.

  The latter is somewhat arbitrary, of course, but it is useful enough for the single purpose of defining input and output filenames in the configuration file.
- `required`: an integer, being either `REQUIRED` or `OPTIONAL`.
- `addr`: a pointer to the memory location of the variable.

The C variable named `variable` is declared as an array `[]` of this structure, and it can be directly initialized. Its size is not specified and depends on the number of elements that follow in the initialisation.

The first element (with array index 0) is an empty entry. The following array elements define the variables. After the last entry, the file ends with the second fixed part

```
};
#define NUMBER_OF_VARIABLES (sizeof(variable)/sizeof(variable[0])-1)
```

which closes the array initializer, and defines a preprocessor macro which gives the actual size of the array.

Now to the variable part of `variables.h`: The master module requires two variables, and offers several optional settings that are turned off by default. The provided time integration module `timint` needs the maximum step size for integration. Table 3 lists these variables. Their declaration can be found in `master.h`; only the step size is a shared variable, `maxdt_` (requires a trailing underscore), and thus appears in the file `common.def`. The table in `variables.h` must contain at least the following lines:

```
{"First year", INTEGER_TYPE, REQUIRED, &first_year},
{"Final year", INTEGER_TYPE, REQUIRED, &final_year},
{"Write core", INTEGER_TYPE, OPTIONAL, &write_core},
{"Trap FPE", INTEGER_TYPE, OPTIONAL, &trap_fpe},
{"MM.DD date format", INTEGER_TYPE, OPTIONAL, &date_format},
{"Maximum step size", INTEGER_TYPE, REQUIRED, &maxdt_},
```

The first column is again the literal (string) name to be used in the configuration file, the second column contains the data type, the third controls whether the variable is `REQUIRED` or `OPTIONAL`, and the fourth entry gives the address of the variable.

Care must be taken to give optional variables an initial value. This is not part of the lookup table, but happens either at the initialisation in C (as with `data_format = 0;` in the master module — before parsing the configuration file, of course), or using the `block data` modules in FORTRAN (because `data` assignments are usually not allowed for variables in common blocks, except from the main program).

4.7 Provide modules for time integration.

At the heart of your main program's loops usually resides the code that propagates the system forward through time. This code consists of two parts: the system of differential or difference equations, and the implementation of a numerical or analytical integration method (probably using library calls). With finite differences, this distinction is not necessary.

For continuous systems, it is useful to separate system-dependent information from the system-independent integrator method, especially if the method needs to evaluate changes more than once. But this is a recommendation only, since the subroutine that calculates the system equations is not treated like the other modules, but is exclusively called by the integrator.

Your job is either to modify your dynamics to work with our `timint`, or to put your integrator or finite difference routine into your own `timint`. Regarding the interface, `timint` receives a single argument, namely the time interval in seconds that is to be covered (but see also appendix B.3). The argument is of type `integer`, and since FORTRAN passes arguments by reference, C programmers have to use a pointer, `int*`. The input data is the state of the system prior to time integration, and the output is the state afterwards.

Table 3: Variables of the master module and the integrator module, that can be initialized from the configuration file.

required by `master`:	
`First year`	First year of the model run. The run starts at January 1 on this year.
`Final year`	Final year of the model run. The master module will not process events later than 'just before' January 1 of the following year. But the run may stop earlier if there are no further calling entries of modules.
optional by `master`:	
`Write core`	controls whether program crashes should produce a `core` file. The signal handler catches signals resulting from an illegal instruction (`SIGILL`), a bus error (`SIGBUS`), and a segmentation violation (`SIGSEGV`). It is not possible to continue the run, but the handler can exit cleanly. If you turn on this flag, the handler calls `abort()` to produce a core dump for post-mortem debugging. By default, no core is produced, since the `core` file is usually very large and not too useful (at least not in batch runs on supercomputer hosts).
`Trap FPE`	controls whether floating-point exceptions (`SIGFPE`) should get caught by the signal handler, which gives a warning message and stops the run. The default behaviour is to leave handling to the FORTRAN runtime library. Depending on the platform and compiler settings, these exceptions stop the run with a traceback, or are silently treated as `INF` (infinity) or `NaN` ('not-a-number' as defined by the IEEE standard).
`MM.DD date format`	if non-zero, changes the interpretation and output of dates to 'months first'. Standard behaviour is 'days first'.
emphrequired by `timint`:	
`Maximum step size`	gives the maximum allowed size of a single time step in seconds, which has to be determined using trial and error, or experience. `timint` needs this since it does not use adaptive step size control, but uses fixed steps.

4.8 Finally: Compilation.

Compile your model without the original main program, but with the C master module instead. Your main program will be replaced by a suitable configuration file.

5 The configuration file

The configuration file may contain:

- initial values for some variables
- dependencies for modules
- explicit time entries for modules

Empty lines, or lines containing only white space (blanks, tabs) and comment lines (starting with the character '#' in the first column) are ignored.

The file is scanned in two passes: first, all assignments are processed; then dependencies and explicit time entries are considered. This ensures that `START` and `END` are set before looking at explicit time entries.

The general idea is to use dependencies as much as possible (wherever they occur). If you have to call modules A, B, C, D in advance to calculating coefficients for the equations with module E, then make E dependent on A, B, C, and D. Further on, if C requires B to be called first, make C dependent on B. The master module will sort out the dependencies. You just have to say explicitly, that E is to be called once a month, and all others will be called accordingly.

A given sequence in the configuration file determines the calling sequence, which is modified through dependencies. The calling list is not created completely (for the whole model run) in advance, but a module re-registers itself when it is called. The calling algorithm is explained below.

5.1 Assignments for variables

Assignments for variables have the form

String = *Value*

The string up to the '=' character, not counting leading or trailing white space, represents a variable and can consist of a description text. White space inside the text is allowed. The file `variables.h` defines the connection between text string and memory location of the variable. The text string may differ from the name of the variable, but entries in the config file must match the definition in `variables.h` literally.

The master module requires that four variables must always be set (see previous section). The place of the assignment does not matter.

5.2 Explicit time entries for modules

Explicit time entries have the form

Module name: *Start spec. [End spec. Interval spec.]*

Either you set a single time point, or you give start, end, and interval specification. It is not possible to have start and end only.

A start or end specification can use three different formats:

`YEAR`	implies January 1st, at midnight
`DD.MM.YEAR`	implies midnight
`HHMMSS.DD.MM.YEAR`	

You can switch from DD.MM notation to MM.DD notation using the variable '`MM.DD date format`', see above.

The `YEAR` can be any signed integer, or it can contain `START` or `END`, optionally followed by *+int* or *-int*.

If the time entry is followed by an asterisk (`*`), this means 'just before' the given time, as discussed in section 2.4. For example, you can write annual output at the end of each year using the entry

```
outann: 01.01.START+1* END 1y
```

Older versions of the master module used 'illegal' time entries with a day of 31, or a month of 13, as 'just before' entries. This is detected if one uses the simple calendar, and the user is reminded to change these old-style entries.

The interval specification is an integer larger than zero, followed immediately by a unit (no space allowed). Available units are

`y`	year	`h`	hour
`m`	month	`min`	minute
`mon`	month	`s`	second
`d`	day	`sec`	second

For example, you may not write `1.5h` or `1h30min`, but you can write `90min` instead.

5.3 Dependency entries

Dependency entries have the form

Depending module : *Independent module [Independent module ...]*

This is a similar notation as used in Makefiles. You can specify multiple independent modules on the same line, but note that white space is used as delimiting character. If module names contain white space, they must be enclosed in double quotes.

It is not possible to specify that a module should always be *followed* by another module; the implemented kind of dependency accounts only for the case that the independent module is called beforehand.

Redundant dependencies are admitted: at a given point in time, each module can only be called once, and repeated registration of modules through dependencies is ignored.

Dependencies leading to recursion will not be detected, so that the process will eventually crash due to stack growth failure.

5.4 The calling algorithm

In this section you will get an impression how the machinery within the master program works. This knowledge is necessary to write complex configuration files.

The calling sequence is created in two steps. First, only the explicit time entries are considered. They are rearranged to give a preliminary calling sequence (called `sorted_index`) sorted by time stamps. Entries with the same time stamp are sorted according to the order of appearance in the config file. In the second step, only the entries belonging to the first time stamp are taken. They are 'registered' in the final calling sequence for this point of time.

Registration is a recursive process that checks first if the module is already registered; if not, it registers any modules the current one depends on. Finally the current module is registered and moved in the sorted index to the adequate position for the next call (which is again sorted by time, and by order of appearance). If the entry does not specify a next call, a time stamp of January 1st, (final year + 2) is used, which is not reached.

When all entries at the current time are registered, the calling sequence is complete, and it is executed. Afterwards, the next time stamp in the sorted index is examined. If it lies before the final time stamp January 1st, (final year + 1), the run continues, the time interval is calculated, and if non-zero, integration is called.

Example. Consider the following configuration file, which resembles the example in section 2.4:

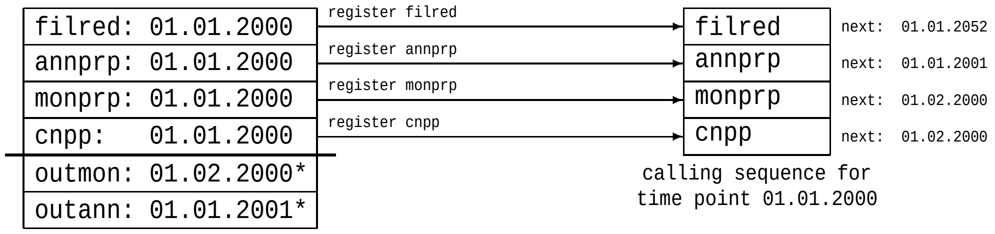
```
First year = 2000                          run from 2000 to 2050
Final year = 2050
filred:  START                             read files first
annprp:  START END 1y                      annual preparation
monprp:  START END 1m                      monthly preparation
cnpp:   START END 1m                       calculate monthly NPP
outmon:  01.02.START* END 1m               output at end of month
outmon:  glosum                            needs global sums
outann:  START+1* END 1y                   output at end of year
outann:  glosum                            needs global sums
```

The resulting table of explicit entries is

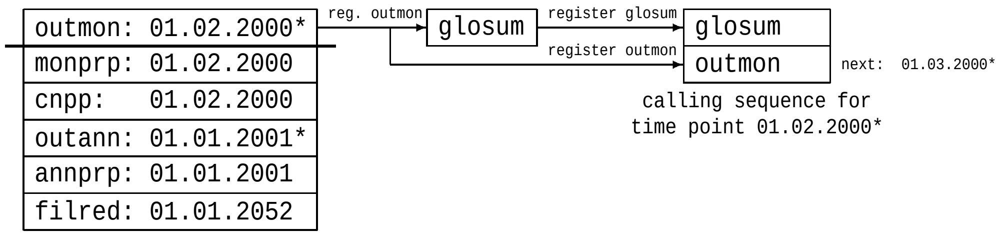
name	first	last	delta
filred	01.01.2000	01.01.2052	0y 0m
annprp	01.01.2000	01.01.2051	1y 0m
monprp	01.01.2000	01.01.2051	0y 1m
cnpp	01.01.2000	01.01.2051	0y 1m
outmon	01.02.2000*	01.01.2051*	0y 1m
outann	01.01.2001*	01.01.2051*	1y 0m

(Observe that single entries have an interval of zero and a last time entry of the final year + 2; `END` entries are replaced by the final year + 1, and the 'just before' flag of the first time entry is preserved.)

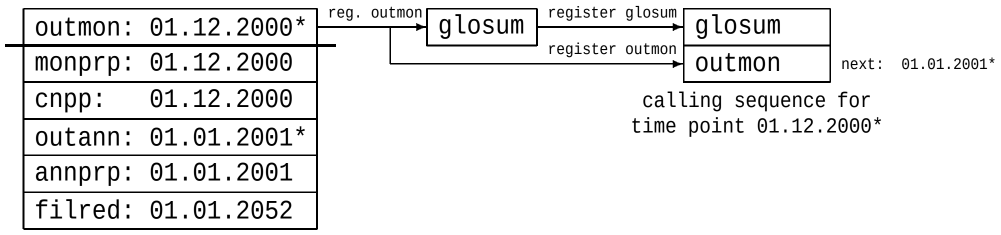
After creating the sorted index, current time is set to the first time entry (01.01.2000). From the sorted index and the dependencies, the following calling sequence is generated:



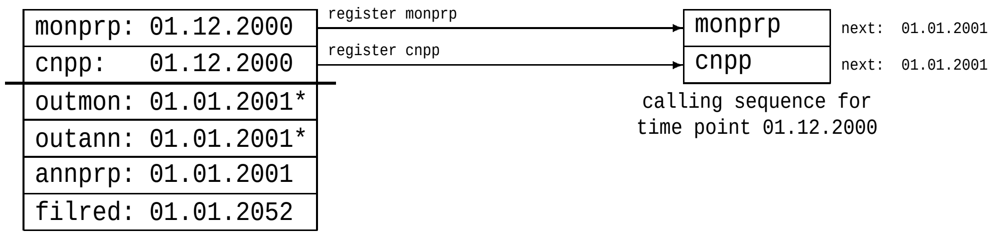
The four modules have been registered for the next call according to the interval specification. For a single call, as with `filred`, the next year is set to the final year + 2, which will never be reached. With each registration, the entry in the sorted index is removed from the top and inserted again at the appropriate position, which is 1) before all later entries, and 2) within entries for the same point in time, the order of first appearance in the config file matters. Therefore, `monprp` remains in front of `cnpp`, and both are inserted after `outmon`, and before `outann` (see below).



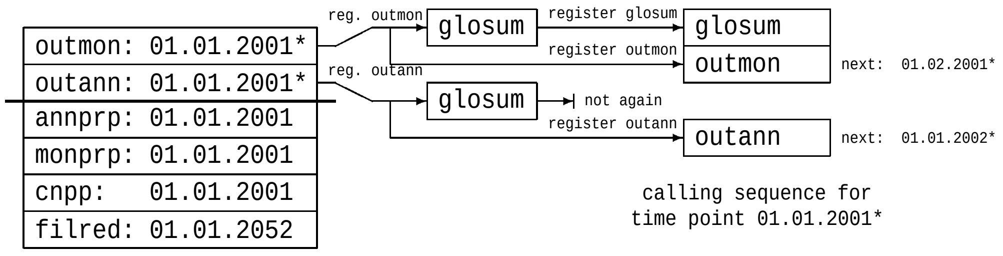
For this point of time, there is only a single entry in the sorted index. But when `outmon` is about to be registered in the calling sequence, the dependency is seen, and `glosum` is registered first. If in turn `glosum` depended on another module, this other one would be registered before `glosum`. Registration is a recursive process. — Now we skip until just before December 1st:



You can see that `outmon` is inserted before `outann`, which has been already registered for `01.01.2001*`, due to their order of appearance in the config file.



Likewise, `monprp` and `cnpp` are inserted after `annprp`, because `annprp` precedes them in the config file.



At this point in time, `glosum` is required by both `outmon` and `outann`. The rule is not to register twice, since the dependency of `outann` is already satisfied by the first registration of `glosum` in the calling sequence.

5.5 Playing with configuration files

You can create your own config files and try them. Change to the directory `conf/` and build the executable like you produced the `kbm`. The model `conf` contains no variables, but several empty modules, `a` to `e`, and those mentioned in the previous section (`filred`, ...).

When running the executable `conf`, you can either use the command line switch `-v` for verbose output of calls and the dummy integration in-between, or, if you are interested in following the internal tables as shown above, you can turn on debugging output with `-d`. The latter option causes lots of information to be written to the error output. Redirect it to a file, or use

```
conf -d whatever.cfg 2>&1 | more
```
(ksh or bash)

or

```
conf -d whatever.cfg >& | more
```
(csh)

to combine standard output and error output and pass both through the pager `more`.

A Command line switches

Four command line switches are recognized by the master module:

`-n` (no calling mode). Specifying this option prevents calling of modules. Otherwise, the model runs as usual. The reason for this option is to check the calling sequence that results from your configuration file without delay due to real computation.

`-t` (timing mode). With this option, a simple profiling is activated. At the end of the run you get a table that lists each module with its CPU time consumption both in milliseconds and as a percentage. There is also additional diagnostic output during the model run whenever a module is called.

`-v` (verbose mode). Using this option, the master module informs about the current time, calls of modules, and time integration calls. Can be useful to watch the progress.

`-d` (debug mode). This requires a preprocessor switch to be set. With debug mode one gets several tables of information while the master module is scanning the configuration file, and when it established the calling sequence, and registers modules for the next call. This option produces lots of output!

`-i` (init shared variables). With this option, the master module calls the automatically generated module `init` which initializes all shared variables at the beginning of the run.

B Internals

This section contains bits and pieces that you don't really want to know, if you can avoid it.

B.1 Porting to a new platform

All that is necessary to add a new platform are a few lines in the `Makefile`. Suppose your platform is called 'HAL'. In the first section which lists platforms, add two new Make variables to store your favourite FORTRAN compiler flags. For single precision, call the variable `SP_HAL`, for double precision, `DP_HAL`. Leave their values empty. Later on, you have to define a new target architecture and the rules for compilation, called `hal-single` and `hal-double`. It's easiest to copy an existing entry and modify it afterwards. Don't bother with optimisation and double precision for the beginning.

External names. First, you have to check the external names that are used by FORTRAN. Change to the directory `tests/` outside of the `kbm/` tree. It contains a FORTRAN subroutine `sub1` and a C main program,

```
subroutine sub1               extern float r;
real r                        extern int i;
integer i                     extern void sub1();
                              void main()
common /r/ r                  {
common /i/ i                     sub1();
                                 i = 12345;
write(*,*) 'i=',i                r = 67890.;
write(*,*) 'r=',r                sub1();
end                           }
```

(where the `()` indicate a C function without parameters) and a `Makefile`. If your FORTRAN compiler is not called `f77`, change the Makefile accordingly. Type `make`, which compiles both files separately (should work ok) and then tries to link them to an executable called `a.out` (this may fail). If you don't get any errors, and if running `a.out` produces

```
i= 0
r= .0
i= 12345
r= 67890.0
```

then FORTRAN and C go very well along. They even use the same names for external (visible) symbols in the object files. Otherwise, check the names used in the object files with `nm sub1.o` and `nm main.o`. Look for '`sub1`', '`r`', and '`i`'. Here is an excerpt of sample output produced on our HP. This may look totally different on your machine, *but what matters is the literal appearance of these symbols*.

```
Symbols from sub1.o:            Symbols from main.o:

Name    Scope  Type             Name    Scope  Type
sub1   |extern|entry  |         sub1   |undef |code   |
r      |undef |common |         i      |undef |data   |
i      |undef |common |         r      |undef |data   |
```

Obviously, naming in both object files is identical. With C compilers, symbol names in object files are the same as in the source code, where mixed case is allowed and distinguished. With FORTRAN compilers, we know of three cases:

1) the names appear in lower-case as shown above,
2) the names are in lower-case with trailing '_',
3) the names are in upper-case (Cray only).

Some FORTRAN compilers of the first category have switches to add an underscore to external names:

compiler	switch
`f77` (HP-UX)	`+ppu`
`f77` (SunOS)	`-ext_names=underscores` (newer compilers only)
`xlf` (AIX)	`-qextname`

The behaviour of the other categories is usually fixed.

A simple solution is to use the C preprocessor for mapping names. We use names with trailing underscore as default and try to produce these names using compiler options. If this doesn't work, we define

```
LCASE_OBJECT_NAMES
```

in case 1), and

```
UCASE_OBJECT_NAMES
```

in case 3). The include files that are automatically generated contain conversion code for this mapping. In C modules, you have to write it explicitly. For example, `cglosum.c` starts like this:

```
#ifdef UCASE_OBJECT_NAMES
#define cglosum_ CGLOSUM
#endif
#ifdef LCASE_OBJECT_NAMES
#define cglosum_ cglosum
#endif
void cglosum_(void)
{
  ...
```

You may also check the names used by C, but they should appear in the object file unmodified. — Having done this, go back to the `kbm/src/` directory, and edit the `Makefile`. Enter the correct name of your FORTRAN compiler, if not `f77`, as `"FC=frt"` or the like (see target `vpp-single`); and the compiler switch that produces underscores goes to the first section of the Makefile, where you have created entries for your 'HAL' machine (see the HP entry, for example). If you need conversion, add the preprocessor symbol to the `CFLAGS` as in the target `cray-double`.

C `main()` and FORTRAN runtime library. The second step is to check if you can link FORTRAN object files with a C object file that contains a `main()` function. Usually, you have to use the FORTRAN compiler for linking, because you need the FORTRAN run-time libraries, some of which unfortunately expect a main program in FORTRAN. But as stated in the preface, mixed-language software development is not uncommon, and compilers are usually prepared for this (but the information can be hidden in some arcane manuals).

Go back to the `tests/` directory. Most linkers produce the `a.out` executable without errors. Only two of the platforms listed above refused to link at first attempt: On the DEC Alpha, the `f77` manpage lists the linker flag `-nofor_main`. On the VPP300, the C `main()` has to be called `MAIN__()` instead to make it work (which is done using the preprocessor compiler option `-Dmain=MAIN__`). On the SGI, the linker issued a warning that the C `main` replaces an earlier definition, but this is exactly what should happen, and it works.

Memory locations. The third step is to check the linker output whether the C and FORTRAN modules access the same variables in memory. Some C compilers require declaring the FORTRAN common blocks as external C variables (on SGI and VPP300), others require the opposite. This peculiar situation arises because there is no C equivalent to a common block. The linker has to choose a replacement.

The solution was to use the preprocessor again. In the C header files, these variables are declared as `EXTERN`, which is defined to be either `extern` or empty. When using the command line, be sure to use `-DEXTERN=` and not `-DEXTERN`, because the latter sets `EXTERN` to 1, which produces strange compiler errors).

Double precision. At this stage, single precision should compile, and you can continue with running the model. For double precision, it should suffice to enter the correct compiler flag into your `DP_HAL` variable. Not all compilers have flags to change the precision. Others let you choose the precision of variables and of constants independently.

An important issue connected to this precision change is the matter of alignment. On 64-bit machines, performances is generally better when double precision elements reside on 8-byte boundaries. Some compilers offer separate flags for padding the elements of a common block to various boundaries, but others just do it silently!

So, if you mix real and integer variables in common blocks, and use automatic double precision, you are in danger: the FORTRAN compiler might silently pad all variables to a 8-byte boundary. This introduces 'holes' in the FORTRAN common block that are not present in the corresponding C structure, and FORTRAN and C variables point to different memory locations.

This is a very good reason for placing each variable in its own common block, as it is done by our automatic common block generator.

Advanced topics. In the `modular/conf/` directory, you can test exception handling with `nan1.cfg` (no core, no special floating-point handling), `nan2.cfg` (no core, but trapping floating-point exceptions), and `nan3.cfg` (drop a core for post-mortem debugging).

Sometimes, the standard handling of floating-point exceptions is adequate, resulting in a traceback and source line numbers. Other times, exceptions are silently ignored, and the machine happily continues calculation with values of `INF` or `NaN`; the only indication is the threefold decrease in execution speed due to taking the exception branches in the mathematical functions. In this case, the settings in `nan2.cfg` should notify you that an exception has occurred; unfortunately they cannot produce real debugging information.

Later on, you may want to try the X11 online plots by uncommenting `FMODX` and the subsequent definitions in the Makefile. Probably you must provide an additional include path for the X11 include files. (This, and other problems, can be discussed in detail via private E-mail.)

B.2 Access of variables in FORTRAN and C

Scalar variables. Consider the scalar FORTRAN variable `CO2`, which contains the current atmospheric CO_2 concentration and resides in the common block named `CO2`. The variable is declared in `auto/co2.f` like

```
real co2
common /co2/ co2
```

(upper or lower case do not matter), and in `auto/co2.h` like this:

```
#ifdef UCASE_OBJECT_NAMES
#define co2_ CO2
#endif
#ifdef LCASE_OBJECT_NAMES
#define co2_ co2
#endif
EXTERN real co2_;
```

(Remember that `real`, which is no valid C data type, is `#define`'d to be either `float` or `double` when calling the C compiler from the Makefile.)

If you want to access the variable from a FORTRAN module, you only need to include the declaration, like

```
subroutine fmod
include 'auto/co2.f'
...
write (*,*) 'co2 = ',co2
end
```

From a C module, you have to use the name with trailing underscore,

```
#include "auto/co2.h"
...
(substitute name mapping as shown on page 25)
void cmod_(void)
{
   printf("co2 = %f\n", co2_);
#ifdef DBL
   scanf("%lf", \&co2_);
#else
   scanf("%f", \&co2_);
#endif
}
```

which is either taken 'as is' or mapped to `CO2` or `co2` by the preprocessor. Note that `co2_` is either declared as single precision ('`float`') or a double precision ('`double`') depending on the definition of the preprocessor symbol `real`. This is no problem for printing, since the format specification '`%f`' applies to double precision variables, and single precision variables are promoted. But for input via `scanf`, you have to use '`%f`' for single precision and '`%lf`' for double precision.

Array variables. Consider `FOSSC(1860:1986)`, a one-dimensional array that contains the annual emissions of fossil C for the respective year, and the two-dimensional array `POOL(NGRID,NPOOL)`, which gives the contents of all pools on all grid elements. The arrays and the common blocks of the same name are declared in `auto/fossc.f` and `auto/pool.f` like

```
real fossc(1860:1986)
common /fossc/ fossc
```

and

```
real pool(ngrid,npool)
common /pool/ pool
```

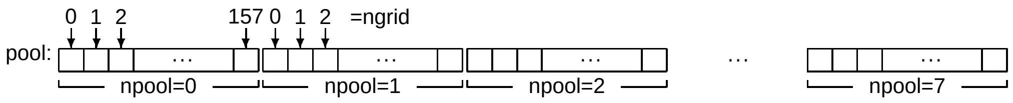
(where `const.f` defines `ngrid=158` and `npool=8`). The corresponding C declarations are kept in `auto/fossc.h` and `auto/pool.h`:

```
#ifdef UCASE_OBJECT_NAMES
#define fossc_ FOSSC
#endif
#ifdef LCASE_OBJECT_NAMES
#define fossc_ fossc
#endif
EXTERN real fossc_[1986-1860+1];
```

(in C, the array index always starts with 0, therefore we get 127 elements with indices 0. . . 126), and

```
#ifdef UCASE_OBJECT_NAMES
#define pool_ POOL
#endif
#ifdef LCASE_OBJECT_NAMES
#define pool_ pool
#endif
EXTERN real pool_[npool][ngrid];
```

where the sequence of indices is reversed, because C uses row-major array storage. Or in other words, `pool_` is an array of `npool` (8) elements, each of which is an array of `ngrid` (158) elements. In memory, the grid index runs fastest, as shown for the C values of the indices (FORTRAN values are one larger). A box represents one value, or four bytes.



Again, access from FORTRAN is straightforward:

```
subroutine fmod
include 'auto/fossc.f'
include 'auto/pool.f'
...
write (*,*) 'fossc(1975) = ',fossc(1975)
write (*,*) 'pool(29,ph) = ',pool(29,ph)
end
```

This prints the emissions of 1975 and the contents of the pool ‘phytomass, herbaceous’ (pool index 1) on grid elment 29. An equivalent C module has to account for the index shift and the different order of indices:

```
#include "auto/fossc.h"
#include "auto/pool.h"
...
(substitute name mapping as shown on page 25)
void cmod_(void)
{
   printf("fossc(1975) = %f\n", fossc_[1975-1860]);
   printf("pool(29,ph) = %f\n", pool_[ph][29-1]);
}
```

where ph is already defined to be '0', not '1' as in the FORTRAN case.

B.3 Internal representation of time

In the current version of the structure, time points are stored using five integer values. A single integer cannot hold the time range which is necessary for biogeochemical models: According to the ANSI standard, a C `unsigned int` is guaranteed to hold at least the value 2^{32}-1, or roughly 4 billion. If we assume that a granularity of time of one second is fine enough, this could represent 138 years from a reference point onwards, which is too limiting.

Therefore, we have reserved one signed integer for storing the year (this way, we can capture most of the Earth's history). Month (0–11) and day of the month (0–29) are also stored separately, if we ever need to have a precise calendar (Gregorian, for example). A fourth integer value is used to store the seconds of the day (0–63,999). If the resolution of one second is not enough, we could even use it to store milliseconds. The fifth value can be 0 or 1, meaning 'at' (or just after) the given point in time, or 'just before'.

Time intervals are measured in seconds, and they clearly do not fit into a single integer. We have split intervals in two parts, namely year difference and remaining seconds. The sign convention is that both are either positive or zero, or both negative.

The integrator module that accompanies the example model uses a single integer that specifies the interval, that is to be covered, in seconds. When called by the master module, this has to be taken into account; calls are split so that the integrator never has to cover more than 2 billion seconds, or roughly 69 years, in one time step.

For models with a large time scale, this is probably too short. As a remedy, the master module and the integrator could be modified to simply use years, or months, instead of seconds, as unit for the interval parameter.

B.4 The KBM's `modules.h`

This section contains the automatically generated file `modules.h`, with some omissions indicated by `[...]`. Note that the trailing comma after the last entry of the table is allowed according to the ANSI C standard.

```
/* auto/modules.h (generated by autogen)
 * Note:
 * - Names must be shorter than 32 characters.
 * - The Symbol NUMBER_OF_MODULES must always give the correct
 *   number of table entries.
 * - Index 0 is not used.
 */
```

```
/* first section: object name translation */

#ifdef UCASE_OBJECT_NAMES
#define arcalc_ ARCALC
#define cld_ CLD
#define clp_ CLP
#define cnpp_ CNPP
#define filred_ FILRED
#define cglosum_ CGLOSUM
#define glosum_ GLOSUM
#define idummy_ IDUMMY
#define ocean_ OCEAN
#define outann_ OUTANN
#define peq_ PEQ
#define pinit_ PINIT
#define prread_ PRREAD
#define prwrit_ PRWRIT
#define timint_ TIMINT
#define plot_ PLOT
#endif
#ifdef LCASE_OBJECT_NAMES
#define arcalc_ arcalc
[...]
#define plot_ plot
#endif

/* second section: external declaration */

extern void arcalc_(void);
[...]
extern void prwrit_(void);
extern void timint_(int*);
#ifdef __WANT_X11__
extern void plot_(void);
#endif

/* third section: table entries */

struct {
  char name[32];
  void (*addr)();
} module[] =
{
  {"", NULL },
  {"arcalc", arcalc_},
[...]
  {"timint", timint_},
#ifdef __WANT_X11__
  {"plot", plot_},
#endif
};

#define NUMBER_OF_MODULES (sizeof(module)/sizeof(module[0]) - 1)

/* NUMBER_OF_MODULES is *not* the number of elements in the
 * array, but one less!!! we use only elements [1] ...
 * [NUMBER_OF_MODULES]. Therefore, array declarations have to
 * be made with NUMBER_OF_MODULES+1 */
```

`auto/modules.h` consists of three parts. The first part maps object names to strings. The second part contains the *declaration* of all modules, stating that a module is external (not part of the master module's code), takes no argument and returns no value (what a function would normally do).

If you want create your `modules.h` directly, here is an example what you can do. Consider the following legal, if stupid, module

```
SUBROUTINE EMPTY
END
```

which one would declare in the second part of `modules.h` as

```
extern void empty_(void);
```

As you can see, the argument list is 'void', and the return value is also 'void', meaning there is nothing. The module name `empty_` is used like this, because we assume that module names in FORTRAN object files are in lower case and with trailing underscore.

After all modules have been declared, the third part of `modules.h` produces a look-up table for the master module. In this table (an array called `module[]`), the module name written as a character string is connected to the calling address of the module in memory. For the dumb module `EMPTY`, there would be a line reading

```
{"empty", &empty_},
```

(trailing comma included). This associates the string `"empty"` with the address of `empty_` declared in the second part above. Please note that the string given here is later used by the master module for comparison, and must be matched verbatim by entries in the configuration file. Module names may not contain the five characters `:=<>"` and may not start with a digit. They may contain spaces, but then you must use double quotes to enclose the name in the right-hand side of dependency entries in the config file, since spaces are also used as a separator for the independent modules. Otherwise, there is no restriction except for the length of the string, currently limited to 32 characters (including a final zero byte, so that strings may actually be only 31 characters long).

For example, if you like a more verbose description in the configuration file, you could easily write

```
{"Das leere Modul", &empty_},
```

and use a very German-looking configuration file entry

```
Das leere Modul : 02.05.1997
```

stating that the module is to be called on May 2, model year 1997. However, you should keep in mind that the string must match literally, except for leading or trailing white space. Given that the number of modules easily exceeds 50, it is surely better to use the name of the module also for the string, which will happen automatically if you use `modules.def` and `autogen`. With the latter, one would simply have a line in `modules.def` looking like

```
empty   void
```

B.5 The KBM's `variables.h`

This section contains the file `variables.h`, which is not automatically generated.

```
/* variables.h
 *
 * This file is part of the modular structure distribution.
 * Johannes Hoffstadt, 28-MAY-1997
 *
 * This file contains a list of the known variables. It defines a table
 * which translates names of variables to their types and addresses.
 * The table is alphabetically sorted.
 *
 * Note:
 * - A name can be any description, but must be shorter than 80 characters.
 * - The Symbol NUMBER_OF_VARIABLES must always give the correct number
 *   of table entries.
 * - Index 0 is not used, only elements [1] ... [NUMBER_OF_VARIABLES].
 * - Array declarations have to be made with [NUMBER_OF_VARIABLES + 1].
 */

struct
{
   char  name[80];
   int   type;
   int   required;
/*
   union {
      int  *i;
      real *r;
      char *s;
   } addr;
   */
   void *addr;
} variable[] =
{
  { "", 0, NULL },
  { "First year",             INTEGER_TYPE, REQUIRED, &first_year },
  { "Final year",             INTEGER_TYPE, REQUIRED, &final_year },
  { "Write core",             INTEGER_TYPE, OPTIONAL, &write_core },
  { "Trap FPE",               INTEGER_TYPE, OPTIONAL, &trap_fpe },
  { "MM.DD date format",      INTEGER_TYPE, OPTIONAL, &date_format },
  { "Maximum step size",      INTEGER_TYPE, REQUIRED, &maxdt_ },
  { "Initial value for CO2",  REAL_TYPE,    REQUIRED, &co2_ },
  { "Cell data file",         STRING_TYPE,  REQUIRED, &fgrid_ },
  { "Biome data file",        STRING_TYPE,  REQUIRED, &fbiome_ },
  { "Fossil emissions file",  STRING_TYPE,  REQUIRED, &ffoss_ },
  { "Read pools from file",   STRING_TYPE,  REQUIRED, &fpoolr_ },
  { "Write pools to file",    STRING_TYPE,  REQUIRED, &fpoolw_ },
};

#define NUMBER_OF_VARIABLES (sizeof(variable)/sizeof(variable[0]) - 1)
```

Note that `variables.h` refers both to internal variables of the master module, and to shared variables (listed in `common.def`), which have to be accessed from C using the trailing underscore.

C Short description of the master module

At the beginning, the master module creates a list of explicit time entries according to their sequence in the configuration file. This list is called `explicit_table`. It contains the module index (in the array `module[]` defined in `modules.h`), the first, next and last calling times, and the calling interval.

The master module keeps a so-called `sorted_index`, which is a permutation of the entries in the explicit table (it contains a sorted list of all entries in the explicit table). *This sorted index reflects the preliminary calling sequence* (without dependencies).

When created, the sorted index is filled with all explicit entries, which are sorted according to time. Entries belonging to the same time keep the initial sequence of explicit entries.

An additional table, `explicit_index`, stores for each module its first appearance in the explicit table. This information is later used for ordering in the sorted index.

Each time the calling sequence is generated, all entries of the sorted index are processed that belong to the first time point. Each entry is registered, which recursively registers dependencies, and moved in the sorted index to the position that corresponds to the next call.

The created calling sequence is invoked. Integration is called to arrive at the next time point until the model run finishes.

Description in more detail (empty `()` denote C functions):

1. `hello()`: Write starting information (master version, copyright) to stderr.

2. `catch_signals()`: If possible, use POSIX-compliant code to catch some signals: `SIGHUP` (terminal line hang up) and `SIGQUIT` (terminal quit signal) are ignored, so that a model run will survive disconnection from the parent shell. `SIGSEGV` (segmentation violation), `SIGILL` (illegal instruction), `SIGBUS` (bus error), and, optionally, `SIGFPE` (floating-point exception), are treated by a signal handler routine, `mysig`, that either exits cleanly or drops core if post-mortem debugging is intended, depending on the setting of the config file variable '`Write core`'. The treatment of `SIGFPE` is controlled by the variable '`Trap FPE`', which can be set in the configuration file.

3. `process_args()`: Process command line arguments. The command line must at least contain the name of the configuration file. There are some optional switches, described in appendix A.

4. `read_config_file()`: A temporary matrix is allocated to record all dependencies. The config file is processed line by line using `process_config_line()`. This is done twice:

   In pass 1, lines containing '=' are given to `process_assignment()`, which searches the internal list of variables (defined in `variables.h` for the left-hand name, and, if successful, calls `assign()` which assigns the right-hand value via `sscanf()` to the address of the variable. Other lines are ignored.

   In the second pass, lines containing '`:`' are processed by `process_colon_entry()`, which searches the internal list of modules (defined in `modules.h` for the left-hand module name. However, it has still to sort out whether the line is a dependency or an explicit time entry. If the right-hand token starts with a digit or with '`START`' or '`END`', control goes to `process_explicit_times()`, which appends the entry to the list of explicit entries (`explicit_table[]`). Otherwise, `process_dependency()` adds an entry to the dependency matrix.

   `process_explicit_times()` uses `parse_time_entry()` to get first and last calling times, and `parse_time_interval()` for the calling interval.

After the config file has been processed, the dependency matrix is compressed into a linear list (by `compress_matrix()`) and then de-allocated. (This was originally intended to reduce permanent memory usage, but batch jobs on super computers are limited by the maximum memory usage at any point during program execution. Therefore, dynamic memory handling is unnecessary in this case, and the program could be simplified.)

At last, the list of explicit time entries has to be sorted. Items in the list are not moved, a sorted index into the table is used instead (`create_sorted_index()`, `create_explicit_index()`).

5. `set_fp_exception_handling()`: This routine is used only on HP-UX to set the handling of floating-point exceptions (depending on the variable '`Trap FPE`').

6. `time_loop()`: The CPU time counters are cleared, initial time is set, and a loop is entered in which the current time is set, the calling sequence for the time point is created. `create_calling_sequence()` collects all modules due for this time point from the sorted explicit time list. The function `i5cmp()` is used to compare time points. A time point precedes another if the year is smaller, or if years are equal, if the month is smaller, and so on.

   Once a module is registered for the calling list (by `register_call()`, which works itself recursively through module dependencies), the time entry for the next call is updated. The function `i5add()` is used to add the interval (as given in the explicit time entry) to the current time. The result is normalized. 'Just before' entries are preserved.

   The entry in the sorted index is shifted down to a new position (sorted by time, and by order of appearance in the config file).

   Then the calling sequence is executed. After a check if the final time point has been reached, the temporal difference to the next time point is calculated by the function `i5diff()`. If larger than zero, the time integration routine `timint()`, a FORTRAN subroutine, is called to cover the time interval (in seconds). If the interval is larger than 2 billion seconds, `timint` is called repeatedly, never integrating more than this limit at once.

7. `write_timing_info()`: If timing mode has been enabled, summary information about CPU time usage and percentage are printed for each module.

D Troubleshooting

Most problems arise from the use of the configuration file or the creation of the calling sequence.

Other common mistakes (in the sense that we had had unhappy experiences with) include different sizes of `INTEGER` or `REAL` variables used by FORTRAN as opposed to C, so that the master module did not correctly initialize variables.

Several error messages can result from parsing the config file. These are intended to be self-explanatory. Some important ones are:

`Unknown module "..." in config file, line ...`

There is no entry for this module in `modules.h` (spelled in the same way as given in the configuration file). Either you have misspelled the name compared to the spelling of the string name in `modules.h`, or the entry is missing at all.

`Unknown variable "..." in config file, line ...`
: There is no entry for this variable in `variables.h` (in exactly the same spelling as given in the configuration file). Same as above.

`Assignment failed in config file, line ...`
: The right-hand side of the assignment didn't fit the data type specified in `variables.h`. This probably happens only if you try to assign a non-numerical value to an integer or real variable.

`There are missing parameters.`
: Not every variable that requires an initial value has been set. This error message is preceded by detailed warnings for each unset variable.

Finally, there are reports of memory allocation problems. Most of these can be fixed by increasing internal constants in `master.h`, since the master module is largely written without making use of dynamic memory allocation. Messages are:

`Couldn't allocate temporary dependency matrix!`
: If allocation fails, which should rarely happen, the only fix is to reduce the memory usage by other processes on the machine.

`Couldn't allocate temporary dependency counts!`
: See above.

`Explicit times list full (max ... entries)`
: The list of explicit time entries is currently not dynamically allocated, but an array with fixed size (100 entries). You have to increase the preprocessor constant `NUMBER_OF_EXPLICIT_ENTRIES` in `master.h` and recompile.

`Dependency list full (max ... entries)!`
: Increase `INDEX_BUFFER_SIZE` (currently 1000) in `master.h` and recompile.

`calling sequence list full (max ... entries)!`
: Too many modules have to be called at the same point of time. Increase `CALLING_SEQUENCE_SIZE` (currently 100) in `master.h` and recompile.

E More to come

For the next version, your feedback is needed. What feature would you like to see? For example, an idea for facilitating batch runs on compute servers would be to specify a maximum cpu time for the current run. The master module can monitor remaining cpu time, and since it knows all common blocks (and would additionally need to know all saved (static) variables), it could dump them and its internal tables to a file, and resume the run in a subsequent batch job. Comments are welcome.