Software Development Standards for CSEM and CRASH

Gabor Toth

April 24, 2024

Contents

1 Introduction

The Center for Space Environment Modeling (CSEM) and the Center for Radiation Shock Hydrodynamics (CRASH) have been and will be developing complex scientific software. The purpose of this document is to make this process as efficient, successful and painless as possible. We build on the experience of commercial software development practices, in particular the approaches referred to as **agile or extreme programming** and **object oriented programming**. We do not fully adopt these strategies since they would not fit the scientific environment, but we try to use as much as possible and reasonable.

The purpose of software development standards is to produce high quality software that is

- verified to be correct
- robust and portable
- efficient
- easy to understand
- easy to modify
- easy to use

During the development of the Space Weather Modeling Framework (SWMF) we have already adopted a number of software practices that proved to be useful in achieving the above goals:

- use of version control
- use of object oriented ideas as long as they do not significantly compromise the efficiency
- use of consistent data naming

This document aims at introducing further steps to improve the quality of the software developed at CSEM and CRASH. These are

- general guidelines for software development
- unit and functionality tests
- guidelines for use of object oriented features
- guidelines for formatting source code
- guidelines for documenting source code

The rest of this document will discuss these items in some detail.

Most of the guidelines are very general and apply to all kinds of software and programming language. Some of the standards are specific to the projects carried out at CSEM and CRASH and/or the Fortran 90 programming language. Many of the language specific recommendations can be easily applied to other programming languages. Fortran 90 is stressed because most of the development is done in this language at CSEM and CRASH.

Many of the guidelines may seem self-evident to many developers. That's good news. On the other hand developers have radically different approaches to software development. To be on the safe side a lot of basic ideas are explicitly discussed. The document may also be used by graduate students who just start working with scientific software. Some of the standards may seem arbitrary, although we try to explain the reason behind the standards. Note, however, that any standard is better than no standard at all.

Finally, we often use the word **should** in this document, because it is a customary euphemism in English and a text with a lot of musts and forbiddens is not easy to read. In this document, however, the word **should** actually means **must**. Once the standards are discussed and accepted by the CSEM and/or CRASH groups, they become standards that should be (i.e. must be) followed. This applies to all newly written or rewritten software. If a new software does not comform with the standards, the developer will be asked to make modifications so it does. In the worst case the software will be rewritten by other developers. It will be clear from the next section that developers do not own any part of the software, it can be modified by any other developer if and as necessary or useful.

2 General guidelines for software development

Our software development approach follows several ideas of 'extreme programming', in particular:

- Bottom-up development
- No software written for 'future'
- Simplicity and code reuse
- Continuous and comprehensive testing
- Shared ownership of software
- Pair-programming

While we have an overall vision for our project, the software implementation is done in a gradual manner. The development plan is broken down to small testable steps. Each step is implemented and tested, and kept tested. The main steps for the CRASH development are outlined in the proposal. These can be and should be further broken up into smaller steps. Based on the experience

with the intermediate stages we have the option of modifying the overall plan. This is a crucial advantage over the top-down design.

The following rules should be observed:

- Software should be written in a **modular way** using F90 modules (see section 5)
- Software should be as **simple** as possible. Software should be **reused** and repetitions (of expressions, code segments, procedures) should be avoided. An insignificant gain in efficiency does not justify complicated and/or repetitive code.
- Each module should contain a **unit test** that can be executed with a small driver program. The driver program should also be written. The test should run in a few seconds on a single processor and result in an unambiguous pass or fail message.
- Each new feature of the code has to be carefully **verified**. The developer is responsible verifying that the new feature works correctly.
- The new feature must be testable with a **functionality test**. The functionality test must be able to run on a single processor in less than 5 minutes. The results of the functionality test should be compared with a reference solution. Round-off errors should be ignored, but significant errors not.
- As soon as the new feature is considered functional and ready to be used, the functionality test must be included in the **nightly tests**. This can be done by either adding a new test, or by modifying an existing test to cover the new feature.

The software written is shared by the developers. This means that anyone can modify any piece of software. While this may be a somewhat disturbing notion, the other possibility is that developers write code for themselves, which results in code that only one person can understand, modify or debug. If that individual leaves the project (is on vacation) the whole project is in danger.

A particularly efficient way of ensuring that the developers do not get overly attached to a piece of software and they write code that others can understand, is **pair-programming**. Pair programming simply means that two developers work together at a single work station. They may take turns in typing. It has been found by studies of commercial code development that the efficiency of pair-programming can exceed the efficiency of the two programmers working alone. Most notably pair-programming tends to result in fewer errors in the algorithm, fewer bugs, better designed and more readable software. Although pair-programming is not enforced at CSEM and CRASH, it is often done and it is strongly encouraged.

To allow for multiple developers modifying the same source code the developers should follow strict guidelines to avoid frustration and inefficient development. These include

- communication between developers
- use of version control
- testing of changes
- documentation of source code
- consistent and intuitive data naming
- uniform formatting of source code

Most of these items are already part of the software development standards discussed in this document with the exception of the first one.

Developers should communicate before changes are made if possible. This means that the overall plan should be outlined at software meetings or in emails sent to other developers before any code is implemented. This allows others to make comments, suggest modifications or alternative solutions, point out unwanted consequencies etc. If a non-trivial change has to be done fast (for example fixing a bug), the developer should tell others what the change is about in an email sent at the same time the change is done. This saves work (other developers may be debugging the same problem), and allows others to make use of the fix rather than discovering the bug multiple times.

All software implementation should follow the development plan, and it should be implemented in the sequence determined by the development plan. Software that is written for the future is usually never finished and never gets used. Such software is a burden without any benefit.

Note that the software design should still take into account future needs, but the implementation should be motivated by current needs.

3 Version control

We use version control software, originally the Concurrent Version control System (CVS), currently Git, to

- store previous versions of the source code
- document code changes
- to merge parallel changes in a gradual fashion
- to allow automatic testing of the latest (=HEAD) version

We maintain **a single branch** of our software. Multiple branches result in bugs that are fixed in one version, but not the other, features that exist in one version, but not the others, and mergers that can be extremely time consuming and frustrating.

3.1 Committing changes into Git

Having a single version means that the developers have responsibility to keep this version usable for other developers and users of the code. This means that changes that may affect other people have to be tested before committed to the Git repository. New features that are not yet used by others may be committed in untested/non-functional state as long as they do not interfere with other code.

If there is a substantial change in the source code, the code must be tagged before the changes are made. For example

```
git tag BEFORE_CHANGE_26Sept2008
git commit ...
git tag AFTER_CHANGE_26Sept2008
```

This allows users to use the previous version in case the new version has problems, it also allows reproducing earlier results if the new version is not backwards compatible, and it makes debugging easier.

The code versions of the repositories are identified with Git references which are compiled into the executables and shown at the beginning of the simulation run.

3.2 Documenting code changes in Git

Each and every commitment must be accompanied with a short but descriptive log message. Multiple files can be committed together only when the changes are similar in all the files.

3.3 Merging code changes with Git

When a developer attempts to commit a file that has been modified by others since the version the developer has started from, Git will not allow the file to be committed. It will report a conflict. In this case the developer should

- move (or copy) his/her version of the file
- update (or merge) the file from the HEAD version (git pull)
- check and reconcile the differences between his/her code and the updated (merged) version
- check that the merged file still compiles and works correctly
- commit the merged file

Note that Git does a reasonable job of merging file versions as long as there is no overlap in the modified lines. If there is overlap, the result will be rather messy, and it is better to do the merge by hand using tools like tkdiff or xdiff.

3.4 Testing the HEAD version of Git

The SWMF and its components are tested on multiple platforms every night. All developers who make changes to the code must check if the changes had any unwanted effect on the functionality of the SWMF. The results of the nightly tests can be checked on the web page

`http://herot.engin.umich.edu/~gtoth`

that is updated at 9 am Eastern Time every day, including weekends. The test page shows changes in test results, error reports and logs, and changes in the source code relative to the previous day. It also has links to the manuals, so one can check if the manuals were generated successfully.

If the tests fail, the developer is responsible for debugging the problem and fixing the code as soon as possible.

4 Testing

4.1 Checking the code with the compiler

We should take advantage of the compilers checking capabilities as much as possible. **The `implicit none` statement must be used in every single module or external procedure in the Fortran source code**, so that the compiler can find undeclared variables. The intent of all arguments of a procedure must be explicitly given with the `intent(in)`, `intent(out)` and `intent(inout)` attributes. Procedures should be contained in modules so that their use is explicit and their actual argument list can be checked agains the interface by the compiler. The code should execute correctly even if all real variables are initialized to not-a-number values (NaN) when there is no explicit initialization value in the declaration. One cannot rely on the assumption that variables are initialized with some default values. The NAG F95 compiler allows this check with the `-nan` flag. This can be easily activated by the Config.pl script used by the SWMF and all the models in it:

```
Config.pl -debug -O0
```

This will also check index out of range, floating point errors etc.

4.2 Unit tests, verification and functionality tests

Development of smaller program units must include unit tests that run in a few seconds and verify the correctness of the unit exhaustively. The proper functioning of the new features of the components or the coupled code will be established through carefully designed tests. The correctness of the test results have to be verified carefully by the developer. This involves comparison with analytic solutions, grid convergence studies (including order of accuracy) and fabricated solutions.

Once the correct implementation of a new feature is properly verified, an appropriate number of functionality tests are added to the test suite, and/or the existing tests will be modified to cover the new feature. These tests will either test the components separately, or test the coupled code, as appropriate for the new feature. The tests must run in a few minutes on one processor of a work station. The one processor requirement is important, because error messages are often suppressed when the code crashes under MPI. It also makes debugging much easier if the code is running serially. Of course the functionality tests should also work in parallel runs at least up to four processors (this is the maximum number of processors used in the nightly tests).

While the functionality tests are unlikely to produce physically meaningful solutions (due to the coarseness of grids and small number of time steps), checking the results against reference solutions can still guarantee the proper functioning of the code. We may use FCAT or similar software to ensure that the functionality tests properly exercise all parts of the newly implemented code. It is also important to test as many as possible combinations of various features while maintaining a managable number of tests. This requires a careful planning of the functionality test suite.

The **input files (if any) and the reference solution must be small (<1 Mbyte) ASCII files** so they are platform independent and can be easily stored in Git. The reference solution typically contains some average values for each state variable at each time step, so any change at any grid point at any time step creates a difference, but the reference solution can still be stored in a small ASCII file.

The extended/modified test suite runs nightly on several platforms and compilers. Each machine checks out the latest version of the code, installs, configures, compiles and runs the tests, and compares with the reference solutions (also saved into Git). This procedure guarantees portability of the code and platform independence of the solution. It also verifies that new development does not introduce unintended changes in the already working and tested parts of the code. The results of the system tests are reported in an easy to interpret table on the web at

`http://herot.engin.umich.edu/~gtoth`

The source code changes relative to the previous day are also available at this web page. Since the tests run fast and require minimal computational resources, the developers can easily track down problems and fix the errors. Checking this test page daily should become a habit for the developers.

A useful tool for checking the results of unit and functionality tests against a reference solution is the script

`share/Scripts/DiffNum.pl`

that compares numbers in two files and reports back differences above a given absolute or relative tolerance. Tests should be done with double precision accuracy, and the reference solution should be written out to sufficient number of digits to show changes that exceed round-off errors.

Examples of unit testing can be found in

```
share/Library/test
```

Examples for functionality tests can be found in the main SWMF directory and the main BATSRUS directory in the file

```
Makefile.test
```

In both cases typing

```
make test
```

executes all the tests, and

```
make test NP=4
```

executes the parallel tests on 4 processors. Typing

```
make help
```

or

```
make test_help
```

shows how one can do individual tests. These examples should be closely followed so that any developer can test any part of the code without reading a manual or source code.

5 Guidelines for the use of Object Oriented features

Object oriented (OO) programming has proven to be a useful and productive approach in the commercial software development environment. It allows a modular development of large and complex software. The objects represent data and associated methods. Objects may have multiple instances. Objects belong to classes that define the data contained in the objects and the class methods that can operate on the objects. Classes form a hierarchical structure, and the relationship between a class and a subclass (also called derived class) results in inheritence. In the classical example the class of squares can be a subclass of the class of rectangles which can be a subclass of the class of quadrangles etc. One can have a general method that can calculate the area of a quadrangle in the quadrangle class. This general method can be inherited by the subclasses, or they can have their own more specialized methods.

The Fortran 90 language supports object oriented programming to some extent. With a lot of effort one can mimic essentially all features of OO, but it is not really convenient. In addition, the scientific code development may not benefit as much from a full OO approach as commercial software development. Here we provide some guidelines on the use of some of the OO concepts in our software development effort.

5.1 Use of modules

Legacy Fortran codes consist of a large number of subroutines and functions that are called by each other in an essentially arbitrary manner. The interfaces (argument lists) of the subroutines are not checked, the hierarchy of the methods (if any) is not expressed in an explicit manner. Shared variables are stored in common blocks and they can be used by any of the subroutines. Unfortunately, the adjective *legacy* is often a euphemism for *badly written*.

In our object oriented style Fortran 90 codes the data and the methods are stored in modules. Modules can have private and public data and methods, and these are explicitly declared at the beginning of the module. Argument lists are checked at compile time, since the module provides an explicit interface for them. One can also use named arguments, optional arguments and module procedures with multiple variants (similar to polymorphism in OO). This is not possible with external subroutines that do not have an explicit interface definition.

One module can use another one, which creates a hierarchy. Note that circular dependency is not allowed (more about this below). Well designed modules roughly correspond to classes. Objects correspond to complex data structures, usually derived types. The contained methods in the module correspond to the class methods. One module using another one corresponds to inheritence. Multiple instances of an object can be realized by pointer type data. Pointers can be allocated multiple times creating an arbitrary number of instances of an object.

In practice we use modules to collect cohesive parts of the program. For example the module ModLinearSolver may contain various methods to solve a linear system of equations. The module starts with a short description, the list of public data and methods:

```
module ModLinearSolver

  ! Contains various methods to solve linear system of equations.
  ! There are both serial and parallel solvers, and direct and
  ! iterative solvers.

  use ModMpi, ONLY: iComm
  use ModBlasLapack, ONLY: dgemm, dcopy

  implicit none
  save

  private ! except

  public:: linear_gmres    ! GMRES iterative solver
  public:: linear_bicgstab ! BiCGSTAB iterative solver
  public:: linear_block_lu ! LU preconditioner for up to hepta block-diagonal
  public:: linear_upper    ! multiply with upper block triangular matrix
```

```
public:: linear_lower    ! multiply with lower block triangular matrix
public:: test_linear_solver
```

This particular module does not have any public variables, but it could. Next comes the declaration of local (private) variables and then the contained subroutines and functions. There is no object associated with this module, it is simply a collection of methods.

Note that there is a method `test_linear_solver` that contains the unit test. It is also a public method so that the driver program can call it.

Also note that the variables (and methods) used from the other modules are explicitly listed after the `only` attribute. This is quite helpful for the developers, because it makes clear where the variable declarations can be found. This practice also helps avoiding (or discovering) data name conflicts.

5.2 Public data and public methods

In pure object oriented design objects can only be accessed through methods. While this approach has certain benefits (e.g. the objects can only be modified by the methods and not directly), it may not be a practical approach in scientific programming. Having a public data (say an array) allows other modules to do arbitrary operations on the data. There is no need to write an interface for all the possible operations. For example if there is a logical variable in a module, there is no need to write a subroutine to set it and a function to read its value.

The public interface of the module should be kept as simple as possible. Having dozens of public methods is not any better than having dozens of public variables. Subroutines or functions performing a single line are not useful. These should be avoided.

Public variables and methods should have a name that shows that they belong to the module. For example all of them can contain the string 'linear' if they belong to module ModLinear. Private variables and methods can have simpler names as they do not conflict with the name space of other modules. Although one can rename the variables in the use statement, e.g.

```
use ModLinear, ONLY: solve_linear => solve
```

we did not find this style useful for a number of reasons. One is that certain compilers get confused if two modules contain methods with identical names. Although this is a compiler bug, we have to work around it if we want to produce a portable code. Second, it is usually not a good idea to change the name of the same entity through the program. For example if the code contains a line

```
call solve_linear(x, y, b)
```

one tends to search for `subroutine solve_linear` in the source code.

A nice way to reduce the number of public methods is the use of module procedures. Module procedures allow the same method to be used with different argument lists. For example one needs to convert spherical coordinates to Cartesian coordinates. The coordinates can be given as 3 element arrays, or 3

scalars. Instead of defining 4 different public methods one can define a single generic method with 4 different argument lists:

```
public:: sph_to_xyz     ! convert spherical into Cartesian coordinates
interface sph_to_xyz
   module procedure sph_to_xyz11, sph_to_xyz13, sph_to_xyz31, sph_to_xyz33
end interface
```

where the four variants have the following interfaces

```
subroutine sph_to_xyz11(Sph_D, Xyz_D)
  real, intent(in) :: Sph_D(3)
  real, intent(out):: Xyz_D(3)

subroutine sph_to_xyz13(Sph_D, x, y, z)
  real, intent(in) :: Sph_D(3)
  real, intent(out):: x, y, z

subroutine sph_to_xyz33(r, Theta, Phi, Xyz_D)
  real, intent(in) :: r, Theta, Phi
  real, intent(out):: Xyz_D(3)

subroutine sph_to_xyz33(r, Theta, Phi, x, y, z)
  real, intent(in) :: r, Theta, Phi
  real, intent(out):: x, y, z
```

On the other hand using module procedures to define alternative argument lists for a method that is only used once in the whole code is pointless.

5.3 Circular dependencies

Most scientific codes are written as a collection of subroutines and functions. As we are gradually moving towards the object oriented style we are often confronted with a situation when `module A` uses something from `module B` and vice versa, `module B` uses something from `module A`. Often the use is indirect through other modules.

Although the fact that circular dependency is not allowed looks like an annoying restriction for the programmer used to traditional Fortran codes, it is actually a signal of the lack of hierarchy between modules. Having a hierarchy expresses the logical structure of the code.

Circular dependency can sometimes be resolved by moving some variables or methods between the modules. Sometimes one can pass a variable as an argument instead of using it from the module. But often it is necessary to change the hierarchical structure of the modules. Typically one can introduce module C that is split off from module A and contains the variables and/or methods needed by module B. Module C is therefore used by modules A and B, and module A can use whatever it needs from module B. This way the circular

dependency is resolved. It may also be possible to collect the shared variables and methods from both modules A and B into module C, and then modules A and B use module C, but not each other. The best solution depends on the particular situation. The new module structure should better express the logical structure of the code.

A bad solution is to add external subroutine(s) as an interface for the module(s) so that the external subroutine can be called by the other module, and then the external subroutine calls the internal method of the original module. This work-around beats the whole purpose of the OO approach. There is no argument list checking, since an external subroutine is used, there is a duplication of code, and the modules remain interdependent instead of forming a hiearchy.

5.4 Avoid fancy features

Fortran 90 comes with a number of features, such as operator overloading, functions of dynamic type, etc. that are certainly useful in certain type of programs, but they are not really needed for scientific software development. Using these features for their own sake creates a number of problems. We found again and again that the more advanced features of Fortran 90 cause problems for some of the compilers. It can also make the source code unnecessarily complicated.

In general we suggest to **use subroutines instead of functions** if the return value is not a simple scalar, and especially if the size of the return value array or string depends on the input arguments. For example

```
function add_vector(n, a, b)
  integer, intent(in):: n
  real,    intent(in):: a(n)
  real,    intent(in):: b(n)
  real, dimension(n) :: add_vector
```

can be replaced with

```
subroutine add_vector(n, a, b, c)
  integer, intent(in) :: n
  real,    intent(in) :: a(n)
  real,    intent(in) :: b(n)
  real,    intent(out):: c(n)
```

This example also shows a useful convention: input arguments come first, output arguments are at the end of the argument list. This convention should be followed.

Although the `function add_vector` can be used in a somewhat more flexible manner than the subroutine, it is likely to cause problems for certain compilers, and it may also result in a less optimal code. Another potential problem with functions is that if they occur in a write statement, and there is a write statement inside the function itself (e.g. for debugging) the code crashes with a runtime error, since Fortran does not allow nested write statements.

Another nice feature of Fortran 90 are **automatic arrays**. Unfortunately we found that some F90 compilers fail to deallocate the automatic array after exiting the subroutine or function, which results in a memory leak and eventually a run time crash. For this reason we have to use allocatable arrays instead of automatic arrays. In some respect this makes the dynamic nature of the array more explicit. On the other hand one needs to pay attention to deallocate the arrays for all possible returns from the subroutine or function. Note that one can also use allocatable arrays with a save attribute if that is useful.

5.5 Derived types, pointers and arrays

The most general data structure in Fortran 90 is the derived type. It can contain an arbitrary number of elements, each with different types and sizes, it can contain pointers, and it even allows building recursively defined types to form linked data structures.

While this flexibility is very appealing, it is often found that the performance of the code is substantially degraded if derived types are used in the most calculation intensive parts of the code. For example

```
type MatrixType
  integer:: nRow
  integer:: nColumn
  type(RowType), allocatable:: Row(:)
end type MatrixType
```

is a really elegant data structure, but performing operations on this matrix will be far from optimal. Fortran already has arrays defined, so dense matrices can be represented by a simple allocatable array::

```
  integer:: nRow, nColumn
  real, allocatable:: Matrix_II(:,:)
```

Operations on this matrix are likely to be much better optimized. An alternative is to use a pointer

```
  integer:: nRow, nColumn
  real, pointer:: Matrix_II(:,:)
```

Both allocatable arrays and pointers can be allocated and deallocated. Pointers can be allocated multiple times, which can be useful to create multiple instances. On the other hand, in most cases only a single instance is needed. In this case the allocatable array is preferred, because one can check if the array is allocated or not. A typical statement is

```
   if(.not. allocated(Matrix_II)) allocate(Matrix_II(nColumn,nRow))
```

The initial status of uninitialized pointers is undefined. Trying to use the `associated` function on an unitialized pointer results in a run time error. For

this reason the use of allocatable arrays is preferred if only a single array is needed.

If the size of the matrix is known in advance, a static declaration may be even better for performance:

```
integer, parameter:: nColumn=30, nRow=100
real              :: Matrix_II(nColumn,nRow)
```

On the other hand **large arrays should not be declared with a static allocation**. They use up memory even if that part of the code is not used in a particular run. Certain debugging features (in particular the -nan flag of the NAG compiler) cannot be used for large static arrays.

Typically arrays can be allocated the first time they are used (if it is known for sure which part of the code accesses the array first), or they can be allocated in an initialization subroutine of a module:

```
module ModAdvance

  use ModEquation, ONLY: nVar
  use BALT_lib, ONLY: MinI, MaxI, MinJ, MaxJ, MinK, MaxK, MaxBlock
  real, allocatable:: State_VGB(:,:,:,:,:)

contains
  !============================================================================
  subroutine init_mod_advance

     allocate(State_VGB(nVar,MinI:MaxI,MinJ:MaxJ,MinK:MaxK,MaxBlock))
```

Another appealing use of derived type is to collect information of similar type but different meaning into one variable, for example

```
  type StateType
    real:: Rho                  ! mass density
    real:: RhoUx, RhoUy, RhoUz  ! momentum density
    real:: e                    ! energy density
  end type StateType
  type(StateType):: State_GB(MinI:MaxI,MinJ:MaxJ,MinK:MaxK,MaxBlock)
```

An alternative way of representing the same data is with the use of named indexes (see section 8.7.6):

```
  integer, parameter:: Rho_=1, RhoUx_=2, RhoUy_=3, RhoUz_=4, e_=5
```

While the derived type allows combination of different types, the array allows the use of loops and array syntax. Both approaches result in readable code, e.g.

```
  State_GB(i,j,k,iBlock) % Rho = 0.0
  State_VGB(Rho_,i,j,k,iBlock) = 0.0
```

but the array with named indexes is often easier to use, and is likely to produce more efficient code (if efficiency matters for this data).

Note that using plain numbers to represent elements of different meaning should be avoided. This practice is widely used in old fashioned scientific codes (e.g. U(1) is density, U(2) is velocity, U(3) is pressure etc.) and it results in unreadable and unreliable software.

6 Documentation

6.1 User Manual

At the most basic level unit tests and functionality tests show examples of use. Since the tests are checked every night, the examples of use remain up-to-date and valid.

We have also developed an automated generation of input parameter descriptions. These form the bulk and most frequently changing part of the SWMF user manual. The input parameters are described formally with XML syntax (e.g. their type, range, list of possible values, default values, interdependencies) that also includes a human readable description of the purpose, meaning, and typical use. The XML file is used to check the validity of the input parameters, and it also serves as the basis for the user manual. We will also use the XML file to generate the source code that reads and checks the input parameters and to generate a GUI for editing the parameters (this is a project funded by JPL). Using a single file to describe, check, read, and manipulate the input parameters guarantees that the use of the code remains properly checked and documented.

The rest of the user manual describes the implemented algorithms, the testing procedures, and provide examples of usage for relevant applications. This part of the user manual should be revised periodically, and verified with users who are not part of the development team.

6.2 Documentation of the source code

Source code documentation is a crucial part of the development process. The comments in the source code allow developers to quickly understand the algorithm and implementation details and this can greatly accelerate debugging or modification of the code. Even the original author benefits from writing comments: it helps clarifying the algorithm and it can also be helpful when the code is revisited after a longer period of time.

Comments are supposed to be brief, specific, comprehensible and informative. The following example shows useless comments:

```
!------------- SUBROUTINE USER_SET_PARAMETERS --------------
subroutine user_set_parameters

   ! This is the user_set_parameters subroutine. BE VERY CAREFUL
```

Repeating information that is clear from the source code is useless. Here is an example of useful comments

```
subroutine user_set_parameters

   ! Read parameters for the Kelvin-Helmholtz instability
   implicit none

   real:: Velocity0      ! the shear velocity far from the shear layer
   real:: Width          ! the width of the layer in the Y direction
   real:: Perturbation   ! the perturbation in the Y component of velocity
```

Note that the comments convey information that is not obvious from the source code itself. It is especially important to comment complicated expressions, tricks, and counter-intuitive solutions.

7 Guidelines for formatting source code

Many of the suggestions in this document can be implemented by running the `share/Scripts/FormatFortran.pl` script for one or multiple F90 source files. In fact, the nightly tests perform this for several directories and even commit the modified files. The goal is to have uniformly formatted code in the SWMF and its models.

Formatting the source code has the following purposes:

- Help developers to easily find appropriate parts of the code
- Help developers in following the flow of the algorithm
- Help developers to easily read the code so they can concentrate on the problem at hand.
- Allow all developers modify the source code and still maintain a uniform look.

Although one can come up with many alternative and equally good rules, a team of developers should follow the same set of rules. Since most of the SWMF conforms to a particular set of rules, it makes sense to follow these in future development.

- Fortran 90 syntax and language elements are used everywhere instead of obsolete Fortran 77 syntax and language elements.
- Fortran key words are written in small case letters with the exception of statements that break the flow of the algorithm. These are CYCLE, EXIT and RETURN that should be in all capitals.

- Indentation follows the rules used by the EMACS editor. One can easily format whole files using the ESC Ctrl-Q command in the EMACS editor.
- Comments are written with normal English capitalization. All upper case comments should be avoided unless there is a really good reason.
- No line can exceed 80 character width.
- Subroutines and functions are separated from each other by a single line `!=========` that extends to 79 characters.
- The declaration and the executable part of subroutines and functions are separated by a single line `!------------` that extends to 79 characters
- Blanks are used to make the source code more readable. In particular commas, single or double colons, semi-colons should be followed by a single space. Exception: indexes of multi-dimensional arrays may be separated by commas without a blank so that the array element remains more compact, e.g. $a\_II(1,2) = 3.0$
- The = sign, the + and − and the logical operators ==, / =, >, <, >=, <= should be surrounded by spaces on both sides. Exceptions: the named arguments of a function or subroutine may be written without spaces around the = sign, e.g. `open(iUnit, file=NameFile, iostat=iError)`. For unary + or − (as in $a = -2.0$) use space before, but not after the sign.
- Blank lines are used to separate logically coherent parts of the source code.

Here are some examples that should help clarify these rules. Figure 1 shows some badly formatted source code that does not follow the rules above. Note the use of obsolete Fortran 77 language elements: declarations without `::`, `.gt.` instead of >, goto and continue statements instead of loops with `EXIT` and `CYCLE`, numeric labels, plain `end` statements, `return` statement at the end of a subroutine and function, `ERR=` or `END=` specifiers instead of `IOSTAT=`, etc. In addition there are many formatting problems: wrong capitalization, wrong indentation, missing spaces, missing required separator lines, arbitrary separator lines, lack of empty lines separating program parts, a line exceeding the 80 character width, etc.

The same source code with proper formatting is shown in Figure 2. Hopefully this example is convincing enough that formatting is important. Indentation is very helpful in visually showing the beginning and end of conditional statements and loops. Sometimes nested loops can result in a very deep indentation that is difficult to read, as shown at the top of Figure 3. A better formatting is shown below. Note that the beginning and the end of the 3 nested loops are written into single lines. The CYCLE statement is often easier to read than an if statement that extends over the whole loop.

```
Module ModUser
implicit none
real x,y,z
Contains
SUBROUTINE read_parameters
1 continue
! Here is a badly indented comment
 write(*,*)'Please provide the x, y and z parameters that define the size of the region:
read(*,*,ERR=1)x,y,z
IF(x+y.gt.z)goto 100
!-----------------------------------------------------------
!----- I feel like putting here a lot of separator lines ---
!-----------------------------------------------------------
write(*,*)'x+y should be larger than z. Try again'
goto 1
!\
! $$$$$$$%%%%%%%####### I am so creative #####@@@@@XXXX
!/
100 continue
return
end
logical function is_ok()
is_ok=.true.
return
end
end
```

Figure 1: Badly formatted source code

```
module ModUser

  implicit none

  real:: x, y, z

contains
  !================================================================================
  subroutine read_parameters

    integer:: iError
    !--------------------------------------------------------------------------
    do
       ! Shouldn't we limit reading input to processor zero?
       write(*,*) 'Please provide the x, y and z parameters ', &
            'that define the size of the region:'
       read(*,*,IOSTAT=iError) x, y, z
       if(iError /= 0) CYCLE
       if(x + y > z) EXIT
       write(*,*) 'x+y should be larger than z. Try again'
    end do

  end subroutine read_parameters
  !================================================================================
  logical function is_ok()

    is_ok = .true.

  end function is_ok
  !================================================================================
end module ModUser
```

Figure 2: Well formatted source code

```
select case(NamePlotVar)
case('rho')
   do k = 1, nK
      do j = 1, nJ
         do i = 1, nI
            if(Used_GB(i,j,k,iBlock))then
               Plot_VGB(iPlotVar,i,j,k,iBlock) &
                   = State_VGB(Rho_,i,j,k,iBlock)
            end if
         end do
      end do
   end do
```

An alternative way to write this code is

```
select case(NamePlotVar)
case('rho')
   do k = 1, nK; do j = 1, nJ; do i = 1, nI
      if(.not. Used_GB(i,j,k,iBlock)) CYCLE

      Plot_VGB(iPlotVar,i,j,k,iBlock) = State_VGB(Rho_,i,j,k,iBlock)

   end do; end do; end do
```

Figure 3: Nested loops and CYCLE statement

8 Data Naming

Both CSEM and CRASH are developing a software framework. The software framework consists of the core and the components. Each component corresponds to a particular physics domain (for example the *solar corona* or *radiative transfer*). A particular physics model is regarded as a *component version*. It is important to develop some data naming standards so that the independently developed science models and the framework can be compiled and used together. Also it is useful to have consistent naming when many developers work together on the same part of the software.

The naming standard refers to all variable, type, procedure, module and file names. Here *variable name* means any constant or variable, while *procedure name* means subroutines, functions and main program units.

The purposes of the naming standard are the following:

- Avoid name conflicts between various models of the framework.
- Characteristics of variables, procedures and files are explicitly understood from the name.
- Improved code readability, ie. less time spent on figuring out what the code means and more time is spent on development.

The data naming standard consists of **formal rules** and **guidelines** with respect to the choices made within the limits of the formal rules. Whether the data names obey the formal rules or not can be checked with the script

`share/Scripts/CheckDataName.pl`

Even if the data name conforms with the standars, it can still be a bad data name that does not conform with the guidelines discussed next.

8.1 Guidelines on choosing data names and type

In the good old days programmers worked alone, and they could come up with arbitrary names for their variables and procedures. A certain programmer for example often named his variables 'bubu' because it looked nice in binary format (true story). Others simply went down the alphabet, and used variable names like 'a', 'b', 'c', 'a1' etc. As the size of programs and the number of programmers working on them started to increase, this approach soon turned out to be untenable.

Data names should be

- descriptive
- comprehensible
- concise
- unique

- logical
- easy to remember
- easy to read and write

This is quite a number of requirements which means that the developers need to think before coming up with a new data name. This conscious effort is as much part of programming, as designing an algorithm. The data naming standard helps to form unique and easy to read names, but even if the data name satisfies the formal rules, it may not meet the above requirements at all.

Here are some examples for bad data names:

```
Beta                            ! not specific, beta can be a lot of things
mp_ff                           ! incomprehensible
Number_of_grid_points_in_block  ! too long
nGrid_Points                    ! difficult to remember, not unique
CentripetalForce                ! misleading if it is really centrifugal force
CristophelCoeff                 ! pretentious when it simply means face area/dx
UseCT                           ! incomprehensible for non-experts
```

Here are some good data names for the same data

```
BetaLimiter                     ! specific
message_pass_face_flux          ! descriptive and comprehensible
nCellPerBlock                   ! concise and descriptive
nGridPoint                      ! easy to remember and guess
CentrifugalForce                ! correct name
AreaPerDxyz                     ! descriptive
UseConstrainedTransport         ! comprehensible
```

A note on the use of singular versus plural forms: a lot of data names can be written in singular or plural form. Sometimes the English grammar definitely selects one of these, sometimes both forms are acceptable. For example

```
nBlock            -  number of block(s)
nBlocks           -  number of blocks
calc_face_flux    -  calculate face flux (in general)
calc_face_fluxes  -  calculate face fluxes (for a block)
```

We suggest to use the singular form all the time, because this choice removes the ambiguity. In some cases this violates the rules of English (e.g. nBlock), but the source code is not written English. Not having to remember if a particular data name is in singular or plural form is more important than to (sometimes) follow English grammar.

The choice of variable type is also important. As a basic rule: **numbers should be denoted by numbers, logical values by logical variables, and everything else with strings or derived types.** Sounds logical, still one can find a lot of examples to the contrary. Examples for bad choice for data types:

```
integer:: iMethods       ! 1 = Roe scheme, 2 = HLL scheme, 3 = Lax-Friedricsh
integer:: Done           ! 0 = no, 1 = yes
real   :: cTen           ! = 10.0
logical:: FirstOrder     ! .true. = first order, .false. = second order
character(len=4):: VersionNumber = "8.01"
```

The name of a method should not be encripted into numbers. One could potentially introduce named constants, like `Roe_=1`, `Hll_=2` etc, but it is still more complicated than using a character string.

Logical (boolean) values are best represented by logical type variables. It is much more natural to write `if(Done)then` than `if(Done==1)then`.

Using a real constant named `cTen` instead of using the actual number 10.0 will not make the code any more readable. It may also promote a belief that using `cTen` would be preferred to using a simple number. This may even lead people to write 20 as `cTen + cTen` which is not just unreadable, but also ridiculous.

Using a logical to select between two possibilities is a good idea only if there will never be more than two possibilities. It is quite possible that one day we introduce a third, fourth or fifth order method (and in fact, we did). At that point the logical `FirstOrder` variable won't be very practical. A better choice is to use an integer.

Using a character string to store the version number is a good idea, if the version number may contain letters. But if the version number is truly a number, it is much better to use a number, because numbers can be easily compared. For example if the version number is stored as a real number one can easily check if the version of the code used to create a restart file is less than say 2.01, and then fall back to an earlier way of reading the restart file.

Here is the above list with a good choice of data types:

```
character(len=20):: TypeMethod ! = "Roe", "HLL" or "LF"
logical          :: Done       ! true or false
integer          :: nOrder     ! 1 = first order, 2 = second order
real, parameter  :: VersionNumber = 8.01
real, parameter  :: cPi = 3.1415926535897932
```

An important note about precision of real numbers: we declare most real numbers as `real` and set the precision of the real numbers at compile time using the appropriate compiler flags. All compilers allow promoting single precision (4-byte) reals to double precision (8-byte) reals. The precision can be easily set with

```
Config.pl -single
Config.pl -double
```

however we tend to use double precision almost all the time, because there are several algorithms that do not work well with single precision real numbers. We can also tell if it is running with single or double precision default reals by using the `nByteReal` variable defined in

`share/library/src/ModKind.f90`

The value of `nByteReal` is either 4 or 8 depending on the number of bytes used by the default real variables.

If a number has to be single or double precision, we use the `selected_real_kind` function of Fortran 90 to define real kinds `Real4_` and `Real8_` in module ModKind. These are used like this:

```
use ModKind, ONLY: Real4_, Real8_
implicit none
real(Real8_):: DoublePrecisionVariable
real(Real4_):: SinglePrecisionVariable
```

We **do not use** `double precision::` **in declarations**, since on some machines that corresponds to 16 byte reals. Using the kind definitions we can be sure that we use 4 and 8 byte reals. Note that the actual value of Real8_ and Real4_ varies from platform to platform.

8.2 Guidelines on the use of abbreviations and acronyms

It is quite customary to use abbreviations and acronyms in data names (variable names, file names etc). This saves some typing, makes the source code lines shorter, may even make the code easier to read. The old Fortran 77 standard limited variable and procedure names to 8 characters. This made abbreviations unavoidable. The Fortran 90 standard allows 32 characters, which is much more generous, but one can easily create data names that exceed 32 characters if full words are used. An example

```
subroutine message_pass_corrected_face_flux
```

So abbreviations and acronyms are useful in data names. On the other hand abbreviations that are obvious to one developer may be incomprehensible to others. Often the same word can have different abbreviations. Acronyms are especially likely to be incomprehensible to others. For example one may decide that MP stands for message passing, and CFF for corrected face fluxes. The name of the subroutine now becomes

```
subroutine mp_cff
```

which is nice, short, and completely meaningless to anyone else. One can shorten the name by using abbreviations

```
subroutine msg_pass_corr_face_flux
```

This is fine as long as message is always replaced with 'msg' and 'corrected' is always replaced with 'corr'.

So here are some guidelines on the use of abbreviations and acronyms

- Do not use acronyms that are not comprehensible to ALL developers

- Use abbreviations only if necessary
- Use abbreviations in a consistent manner

Note that the data naming standard itself contains a number of acronyms, but these are supposed to be understood by all developers at CSEM and CRASH.

8.3 Avoiding name conflicts

Since the science models are mostly developed by scientists and not by software engineers, it seems reasonable to minimize the constraints on the data naming. The minimum requirements are the following

- Procedure and module names of a model should start with a unique identifier of the component.
- Input and output file names used by a model should start with a directory named as the component identifier.

Our current practice is not to enforce the rule for procedure and module names unless a conflict exists, on the other hand we enforce the rules for file names. We have developed tools to rename procedures and modules:

```
share/Scripts/Methods.pl
share/Scripts/Rename.pl
```

The first script can find all modules and external subroutines and functions in a list of source files. The second script can rename these by adding the appropriate component ID to the names.

Table 1 shows the IDs of the control module and the current and planned components of the SWMF. The ID `MH` stands for the generic magnetohydrodynamic physics module, which can model GM, IH, SC, OH, and possibly EE.

Here are some examples for module, procedure and file names

```
module CON_session
module PW_ModMain
subroutine GM_set_parameters
function IE_is_spherical_grid
NameLogFile = 'GM/plots/log.dat'
```

Note that internal (contained) subroutines and functions cannot create a name conflict and their name should not start with a component identifier. Also note that a model can use arbitrary input/output file names when it is running in stand-alone mode.

Table 1: SWMF control module and component IDs

CON	Control module of the SWMF
CZ	Convection Zone
EE	Eruptive Events
GM	Global Magnetosphere
IE	Ionosphere Electrodynamics
IH	Inner Heliosphere
IM	Inner Magnetosphere
MH	Generic MHD component
OH	Outer Heliosphere
PC	Particle-in-Cell
PS	Plasmasphere
PT	Particle Tracker
PW	Polar Wind
RB	Radiation Belt
SC	Solar Corona
SP	Solar energetic Particles
UA	Upper atmosphere

8.4 Directory structure and directory names

The components of the SWMF reside in the directory named as the component ID. The science models are in separate subdirectories named as the model. Examples:

```
GM/BATSRUS/
PW/PWOM/
SP/Kota/
```

Within each model directory there are typically the following files and subdirectories

```
Config.pl           configuration script (using share/Scripts/Config.pl)
Makefile            with targets install, LIB, rundir, clean, distclean
doc/                documentation of the model
input/              input files for functionality tests
src/                source code
output/             reference solutions of the functionality tests
```

We adopted the following convention for naming directories: **standard names like src, bin, lib, doc, input, output are spelled in small case, while other directory names are capitalized**. Examples:

```
bin/                executable codes *.exe
lib/                libraries lib*.a
```

```
Scripts/            Perl scripts
Param/              Example and test input parameter files
srcTest/            source code of driver programs for unit tests
srcPostProc/        source code for post-processing executables
```

The source code of the models is in the component ID/model/src directories, which is 3 levels down from the main SWMF directories. To makes searches easier we adopted this rule: **all source code should be 3 level down from the main SWMF directory**. This allows one to search all Fortran files as

```
grep XYZ */*/src*/*.f*
```

8.5 File names

File names of source files should reflect the content. If a source file contains a single procedure, it should be named the same as the name of the procedure with an extension specific to the language. For example a Fortran 90 source code file containing `subroutine calc_flux` should be named `calc_flux.f90`. If a source file contains a set of procedures, it should be named by a group name that describes the set of procedures, or by the main procedure in the group (note: it could be an even better idea to put that group of procedures into a module and name the file accordingly). If a source file contains a Fortran 90 module, it should be named accordingly, e.g. the file containing the `PW_ModMain` module should be named `ModMain.f90` or `PW_ModMain.f90`.

Source files reside in separate directories for each science model, therefore it is not necessary (although allowed) to include the component identifier into the file name.

8.6 Subroutine and function names

The procedure names consist of *procedure name parts* separated by underscores. A procedure name part starts with a lower case letter, followed by an arbitrary number of lower case letters and numbers. The use of lower case letters and underscores between the procedure name parts helps to distinguish procedure names from variable names, which use capitalization (see later). Subroutine names should describe the action done by the subroutine, so it typically starts with a verb. Examples:

```
advance_implicit       ! advance in time with implicit scheme
calc_face_flux         ! calculate face fluxes
set_b0                 ! set the B0 magnetic field
read_restart_file      ! read restart files
```

Function names should describe the type and meaning of the returned value. The type is defined by the first name part. The exact rules for the first name part will be given in the section for variable names. Examples for function names:

```
n_read_line()          ! integer: number of lines read
is_first_session()     ! logical: true in first session
cross_product()        ! real: cross product of two vectors
```

8.7 Module, variable and type names

The variable name standard was developed by G. Tóth, D. De Zeeuw and D. Chesney. We tried to create logical, unique, distinct and easy to read and write variable names. This naming system has been used in the core of the SWMF and most of the recently written or rewritten parts of the BATSRUS code. While some of the rules may not make sense for other science models, most of them are rather general and applicable to any (simulation) software.

8.7.1 Name parts

Each variable name may consist of one or more *name parts*. All the parts must start with a capital letter and continue with lower case letters and numbers. There is only one exception to this rule: the first name part can also start with a lower case letter if it consists of a consists of a single letter (possibly followed by numbers).

In Fortran capitalization is ignored by the compiler, so mistakes in the capitalization have no effect on the correctness of the code. On the other hand consistent capitalization is essential to improve readability of the code. Examples of correct capitalizations:

```
b
B
b0
B0CrossU
iMax
rMin
R2Min
Radius2Min
VarMhd
TypeCoordIh
```

Note that even for the acronyms (MHD and IH) only the first letter of the name part is capitalized. This is necessary because the capital letters show the beginning of the name parts.

8.7.2 Module names

There are two acceptable ways to name Fortran 90 modules.

1. The module names starts with the name part `Mod` and it consists of capitalized name parts with no underscores. This can optionally be preceded by the component ID followed by an underscore. For example

```
ModMain
ModRestartFile
EE_ModCommonVariables
```

2. Well written Fortran 90 codes consist exclusively of modules. Examples are the control module of the SWMF and the BATL library. In these codes the module name starts with an all-capital component/model/library identifier followed by all lower case name parts separated by underscores, similarly to procedure names. For example

```
CON_session
CON_buffer_grid
BATL_tree
BATL_high_order
```

8.7.3 Type names

Fortran 90 derived type names should end with the string `Type`. For example:

```
BlockType
TimeType
```

8.7.4 Indication of variable type

There are strict rules in this data naming standard to indicate the type (real, integer, logical, or character string) of a variable. These rules are made as easy to read and write as possible:

All integer variable names must start with one of the following name parts (we also indicate the usual context):

```
i               first index of
j               second index of
k               third index of
l               length of (e.g. character string)
m               fourth index of
n               number of
Di              difference of index i
Dj              difference of index j
Dk              difference of index k
Dl              difference of length
Dm              difference of index m
Dn              difference of number of
Ijk             i, j and k indexes
Min             minimum number of
Max             maximum number of
Int             generic integer
```

All character and character string type variable names must start with any of

```
Name            name of
Type            type of
Char            generic character
String          generic string
```

All logical variable names must start with one of

```
Do              followed by a verb
Done            followed by a noun
Is              followed by an adjective
Use             followed by a noun
Used            followed by a noun
Unused          followed by a noun
```

Finally all real type and derived type variable names must start with a name part which was not listed in any of the above lists. If the name part ends with some numbers, it does not modify the meaning. For example, "i1" is an integer, "Name12" is string, and "Done2" is a logical.

These rules should also help to make the order of the name parts less arbitrary. For example the minimum of the pressure should be named `pMin` and not `MinP`, because it is a real number which cannot start with the name part `Min` which is reserved for integers. Examples:

```
logical::             DoReadFile          ! read file if true
logical::             DoneComputation     ! done with the computation
logical::             IsNegative          ! is the value negative
logical::             UseConstrainedB     ! use constrained transport scheme
logical::             UsedBlock           ! used block
logical::             UnusedBlock         ! unused block
integer::             iBlock              ! index of block
integer::             nBlock              ! number of blocks
integer::             MaxBlock            ! maximum number of blocks
integer::             MinBoundary         ! minimum value of boundary index
real::                x, r, r2            ! local variables
real::                RhoSolarWind        ! Density of solar wind
real(Real8_)::        TimeCpu             ! CPU time
character(len=20)::   TypeFlux            ! type of the numerical flux
character(len=100)::  NameLogFile         ! name of the log file
character(len=100)::  StringLogVar        ! variables saved into the log file
```

8.7.5 Array variable names

Array variable names are distinguished from scalar variable names by the indication of indexes. The indexes are represented by an underscore followed by capital case letters at the end of the array variable name. Each capital letter represents one or more well defined indexes, and their order must be the same as the order of indexes in the declaration of the array variable. The currently used index abbreviations are shown in Table 2.

Table 2: Index abbreviations used in array names

name	typical range	meaning
A	`1:MaxBlock*nProc`	**A**ll blocks on all processors
B	`1:MaxBlock`	**B**locks on one processor
C	`1:nI,1:nJ,1:nK`	**C**ell centers without ghost cells
C	`1:MaxComp`	**C**omponents of the SWMF
D	`1:nDim` or 1:MaxDim	**D**imensions (of the grid)
E	`1:2*nDim`	**E**dges (of a block)
F	`1:nI+1,1:nJ+1,1:nK+1`	**F**aces (of cells)
G	`MinI:MaxI,MinJ:MaxJ,MinK:MaxK`	**G**host and physical cells
I	`?:?`	**I**ndex (none of the others)
N	`1:nI+1,1:nJ+1,1:nK+1`	**N**odes (of grid cells)
P	`1:nProc`	**P**rocessors
Q	`1:4`	**Q**uadrants for four finer neighbors
S	`1:2`	**S**ides, two ends
V	`1:nVar`	**V**ariables (state, plot...)
W	`1:nWave`	**W**aves
X	`1:nI+1,1:nJ,1:nK`	**X** faces (of grid cells)
Y	`1:nI,1:nJ+1,1:nK`	**Y** faces
Z	`1:nI,1:nJ,1:nK+1`	**Z** faces

Examples:

```
Dx_B(:)           - real array indexed by blocks
GradRho_C(:,:,:)  - real array indexed by cells
B0x_XB(:,:,:,:)   - real array indexed by the X faces and blocks
nRoot_D(:)        - integer array indexed by the 3 directions (x,y,z)
Buffer_VII(:,:,:) - real buffer with one variable and two other indexes
```

8.7.6 Named indexes

To make the integer indexes descriptive, *named indexes* are introduced. A named index is an integer constant (defined with the parameter statement in Fortran 90). The named index consists of the usual name parts followed by an underscore. The underscore is a reminder that named indexes have to do with arrays (array names also contain an underscore), and it also makes the syntax of named indexes different both from scalar and array variable names. Some examples for named indexes:

```
name      value   meaning
-----------------------------------------------
x_        1       X index
y_        2       Y index
z_        3       Z index
r_        1       radial index
Phi_      2       longitudinal index
Theta_    3       latitudinal index
Rho_      1       density index
RhoU_     1       momentum index
RhoUx_    2       X momentum index
RhoUy_    3       Y momentum index
RhoUz_    4       Z momentum index
Bx_       5       Bx index
By_       6       By index
Bz_       7       Bz index
p_        8       pressure index
```

Examples of use:

```
! extract state of a grid cell
State_V = State_VGB(:,i,j,k,iBlock)

! Assign density flux of the X face
Flux_XV(iFace,j,k,Rho_) = State_V(RhoUx_)
```

8.7.7 Real type constants

We use `c` as the first name part of real constants. For convenience, fractions can be written as `c3over5`, which is an exception as the name parts are numbers. Most of these constants are defined in the

```
share/Library/src/ModConst.f90
share/Library/src/ModNumConst.f90
```

modules. Some examples

```
real, parameter:: cPi              = 3.1415926535897932
real, parameter:: cTwoPi           = 2*cPi
real, parameter:: cRadToDeg        = 180/cPi
real, parameter:: cDegToRad        = cPi/180
real, parameter:: cElectronCharge  = 1.6022E-19
real, parameter:: cProtonMass      = 1.6726E-27
real, parameter:: c7over120        = 7.0/120.0
```

8.7.8 Pointer variable names

Pointers should be named according to the rules that apply to the variable it is pointing to. In addition the name should end with the name part `Ptr` to indicate that the variable is a pointer. Examples:

```
integer, pointer:: iGridPtr
logical, pointer:: UsedGridPtr_I(:)
real,    pointer:: SizeGridPtr_D(:)
```