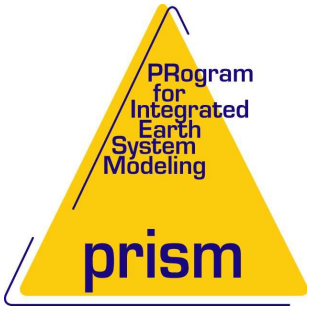
PRISM

Program for Integrated Earth System Modelling



PRISM Support Initiative

PRISM Coding Rules

Angelo Mangili, Mauro Ballabio, Djordje Maric, CSCS
Luis Kornblueh, MPIfM
Reiner Vogelsang, SGI
Philippe Bourcier, IPSL

PRISM Report Series # 23

Last Edited: 31st May 2002

Contents

1 Introduction **2**
1.1 Target Architecture for the Development of the PRISM System . . . 2

2 General Software Development Guidelines **3**
2.1 The Staged Delivery Model . . . 3

3 Coding Conventions for the Development of PRISM Components **4**
3.1 Fortran Coding Standard . . . 4
3.1.1 Restriction to the Language . . . 4
3.1.2 Style Rules . . . 5
3.1.3 Content Rules . . . 9
3.1.4 Package Coding Rules . . . 12
3.1.5 Interface to Other Languages . . . 16
3.1.6 Code Parallelization Issues . . . 16
3.2 C Coding Standard . . . 17
3.2.1 Restriction to the Language . . . 17
3.2.2 Content Rules . . . 18
3.2.3 Interface to Other Languages . . . 18
3.2.4 Code Parallelization Issues . . . 18

4 Component and Unit Testing **19**
4.1 Designing Good Component Tests . . . 19
4.1.1 Unit-tests . . . 19
4.1.2 Functional-tests . . . 20
4.1.3 Component-tests . . . 20
4.1.4 PRISM testing requirements and implementation details . . . 20

5 System Testing and Validation **21**
5.1 Model Testing Procedures for the PRISM system . . . 21
5.1.1 Development Testing Steps . . . 21
5.2 Model Validation Procedures for the PRISM System . . . 22
5.3 Port Validation of the PRISM System . . . 22

6 Code Maintenance Issues **23**
6.1 Software Revision Control . . . 23
6.1.1 What is CVS? . . . 23
6.1.2 What is CVS not? . . . 23
6.1.3 CVS Features . . . 24
6.1.4 CVS Repository Access . . . 25
6.1.5 Recommendations for Code Developers . . . 25
6.2 Code Maintenance Process . . . 25

7 Code Quality Assurance **27**
7.1 Code-Reviews . . . 27
7.1.1 Possible Implementation of Code-Reviews . . . 27
7.1.2 Recommendation . . . 28

Appendix A: Testing Terminology **29**

Bibliography **30**

1 Introduction

This PRISM Software Developer's Guide is the reference handbook that describes the development practices, standards, and conventions recommended for the development of PRISM base software and components.

The scope of this document includes any issue related to process, conventions, and standards in the design, implementation, and documentation of any software that will be developed in the frame of the PRISM project under the consideration of portability, sustained performance and ease of use.

This is an evolving document, part of this document is based on the work done by the *European Co-operation for Space Standardization* [4] [3] extensively covering the whole process of the software development and software life cycle, and on the *"Community Climate System Model, Software Developer's Guide"* (http://www.ccsm.ucar.edu/csm/working_groups/Software/dev_guide/dev_guide/). Suggestions on how to improve this Guide should be sent to Mangili@cscs.ch, Kornblueh@dkrz.de and Reiner@sgi.com.

1.1 Target Architecture for the Development of the PRISM System

The goal of this section is to attempt to propose the most likely target architecture on which the various PRISM components can be integrated into a PRISM system in a first instance. This recommendation is a summary of the conclusions of the document "PRISM REDOC III.2: Review of Current and Future HPC Architectures" [6].

The main HPC facilities available now and most probably to come in the next years are based on clusters of SMP (most of them microprocessor-based, shared-memory inside nodes, distributed-memory between nodes, complemented with fast node interconnects) built either out of RISC or vector CMOS CPUs. Such platforms might have complex memory hierarchies, and often memory bandwidth and latency issues dominate performance. For the PRISM system the first priority must be placed on making the PRISM components run efficiently on these platforms.

Targeting portability on several HPC facilities the PRISM system is likely to be run on both vector and RISC-based CPUs. Ideally, the code would have the flexibility to run efficiently on both. In practice, this can be difficult to achieve, since the choice of optimal data structures, loop ordering, and other significant design decisions may differ depending on whether code is intended for vector or RISC CPUs. This is however extremely important seen that the highest sustained performance for earth science applications will be most probably reached by combining both the powerful CMOS CPUs and the high level application parallelization, as it has been recently successfully demonstrated be the Japanese Earth Simulator. Ideally the PRISM software should run both on the Earth Simulator and ASCI roadmap similar machines.

It is a priority to write flexible code and, whenever possible, to use data structures that are likely to achieve acceptable performance on either architecture. For excellent performance on a given architecture, this can add considerable complexity, the most practical solution for some portions of the model may be the development of two code versions, one scalar and one vector (even if this is discouraged by obvious maintenance problems). In this frame a validation test suite, running regularly on ideally all of the target platforms, testing the different PRISM components and the whole PRISM system is essential.

2 General Software Development Guidelines

In this section we would like to give a general overview of the different steps involved in the software development process. The software life cycle is the sequence in which a project specifies, prototypes, designs, implements, tests, and maintains a piece of software. Explicit recognition of a life cycle encourages development teams to address development issues at the appropriate time; for example, to establish basic software requirements before design or coding begins. We recommend that developers roughly follow the staged delivery model (below) when designing significantly new versions of the full PRISM system and when developing large PRISM components and libraries.

2.1 The Staged Delivery Model

The staged delivery model involves the following steps [8] :

1. **Software Concept:** Collect and itemize the high-level requirements of the system and identify the basic functions that the system must perform.
2. **Software Requirements Analysis:** Write and review a requirements document, a detailed statement of the scientific and computational requirements for the software. Both scientists and the code development team should review and approve the requirements document .
3. **Software Architectural Design:** Define a high-level software architecture that outlines the functions, relationships, and interfaces for major components. Write and review an architecture document.
4. **Stage 1, 2, ..., n:** Repeat the following steps creating a potentially releasable product at the end of each stage. Each stage produces a more robust, complete version of the software.
   (a) **Detailed Design** Create a detailed design document and API specification. This can be done by writing code headers instrumented for ProTeX (as described in 3.1.3). Incorporate the interface specification into a detailed design document and review the design.
   (b) **Code Construction and Unit Testing** Implement the interface, debug and unit test.
   (c) **System Testing** Assemble the complete system, verify that the code satisfies all requirements.
   (d) **Release** Create a potentially releasable product, including User's Guide and User's Reference Manual. Frequently code produced at intermediate stages software will be used internally.
5. **Code Distribution and Maintenance** Official public release of the software, beginning of maintenance phase.

Small, simple pieces of software may not require reviews and separate documents at each stage, but it is still a good idea to prepare at least a design document and review it before implementation.

3 Coding Conventions for the Development of PRISM Components

It is recommended that all new PRISM components follow a coding convention. The goal is to create code with a consistent look and feel so that it is easier to read, understand, port and maintain.

3.1 Fortran Coding Standard

This section defines a set of Fortran specifications, rules, and recommendations for the coding of PRISM components. The purpose is to provide a framework that enables users to easily understand or modify code, or to port it to new computational environments. In addition, it is hoped that adherence to these guidelines will facilitate the exchange and incorporation of new packages and parameterizations into PRISM components. Other works which influenced the development of this standard are "Community Climate System Model, Software Developer's Guide" (http://www.ccsm.ucar.edu/csm/working_groups/Software/dev_guide/dev_guide/), "Report on Column Physics Standards" (http://nsipp.gsfc.nasa.gov/infra/) and "European Standards For Writing and Documenting Exchangeable Fortran 90 Code" (http://nsipp.gsfc.nasa.gov/infra/eurorules.html).

3.1.1 Restriction to the Language

- **Fortran 95 Standard** The PRISM coupler will adhere to the Fortran 95 language standard and not rely on any specific language or vendor extension. The purpose is to enhance portability, and to allow use of the many desirable new features of the language. If a situation arises in which there is good reason to violate this rule and include Fortran code which is not compliant with the f95 standard, an alternate set of f95-compliant code must be provided (tested and validated...). This is normally done through use of a C-preprocessor #ifdef-#else-#endif construct.

- **English Language** Owing to the international user community, all naming of variables, modules, functions, and subroutines as well as all comments are to be written in English. In case you encounter a situation which you cannot solve in Fortran 95 (e.g. you need to include a library written in C), please make sure the non Fortran code only uses ANSI C with POSIX extensions as described in 3.2. This type of code should go into a support library or other specialized libraries.

- **Features to be Avoided**

  In terms of keeping code up to date and more easy to maintain the code should always follow the current standards of Fortran and ANSI C. We would like to restrict the languages use to the elements which are not obsolete and deleted — even if they are still available with almost all compilers. Examples for Fortran 95 are:

  - `COMMON` blocks - use the declaration part of `MODULE`s instead.
  - `EQUIVALENCE` - use `POINTER`s or derived data types instead to form data structures.
  - Assigned and computed `GOTO`s - use the `CASE` construct instead.
  - Arithmetic `IF` statements - use the block `IF`, `ELSE`, `ELSE IF`, `END IF` or `SELECT CASE` construct instead.
  - Labeled `DO` constructs - use unlabeled `END DO` instead. Due to the structure of the physics routines and the large amount of scientists working on them, we recommend for the ease of communication labeled do constructs of the `DO`- labeled `END DO` construct.
  - I/O routines `END` and `ERR` use `IOSTAT` instead (the use is somehow restricted due compiler implementation dependent error numbering).
  - `FORMAT` statements: use character parameters or explicit format specifiers inside the `READ` or `WRITE` statement instead.

- Avoid any unused statement like a labeled `CONTINUE` not being jumped to.
- The only sensible use of `GOTO` is to jump to the error handling section at the end of a routine on detection of an error. The target label should be a `CONTINUE` statement the label should be 999. However, it is recommended to avoid this practice and use `IF`, `CASE`, `DO WHILE`, `EXIT` or `CYCLE` statements or a contained `SUBROUTINE` instead. If you feel you cannot avoid a `GOTO`, then add a clear comment to explain what is going on and why you need to use `GOTO`. Also add a comment to the labeled `CONTINUE` statement.
- `PAUSE`
- `ENTRY` statements: a subprogram may only have one entry point.
- Fixed source form
- Avoid functions with side effects. Although this is common practice in C, there are good reasons to avoid this. First, the code is easier to understand, if you can rely on the rule that functions don't change their arguments, second, some compilers generate more efficient code for `PURE` (in Fortran 95 there are the attributes `PURE` and `ELEMENTAL`) functions, because they can store the arguments in different places. This is especially important on massive parallel and as well on vector machines.
- Try to avoid to use `DATA` and `BLOCK DATA`. This functionality is given by initializers in Fortran 95.

3.1.2 Style Rules

- **Preprocessor** Where the use of a language preprocessor is required, it will be the C preprocessor (cpp). cpp is available on any UNIX platform, and many Fortran compilers have the ability to run cpp automatically as part of the compilation process.
- **Free-Form Source** Free-form source will be used. The f95 standard allows up to 132 characters, but a self-imposed limit of 80 should enhance readability and make life easier for those with bad eyesight, who wish to make overheads of source code, or print source files with two columns per page. The world will not come to an end if someone extends a line of code to column 81, but multi-line comments that extend to column 100 for example would be unacceptable.
- **Fortran Keywords** Fortran keywords should be written in upper case, the remaining code in lower case.
- **Variables Names and Declaration**
  - Use meaningful English variable and constant names.
  - Usage of the `DIMENSION` statement or attribute is not necessary. Declare the shape and size of arrays inside parentheses after the variable name on the declaration statement.
  - The `::` notation is quite useful to show that this program unit declaration part is written in standard Fortran syntax, even if there are no attributes to clarify the declaration section.
  - Declare the length of a character variable using the `CHARACTER (len=xxx)` syntax — the len specifier is important because it is possible to have several kinds for characters (e.g. Unicode using two bytes per character, or there might be a different kind for Japanese eg. NEC).
  - Never use a Fortran keyword as a routine or variable name.
- **Parameter Declaration** Variables used as constants should be declared with attribute `PARAMETER` and used always without copying to local variables. This prevents from using different values for the same constant or changing them accidentally.

- **Program Units** Always name program units and always use the corresponding `END PROGRAM`; `END SUBROUTINE`; `END INTERFACE`; `END MODULE`; etc construct, again specifying the name of the program unit. This helps finding the end of the current program entity. `RETURN` is obsolete and so not necessary at the end of program units.
- **Loops** Loops should be structured with the `DO-END DO` construct as opposed to numbered loops.
- **Argument Comments** Input arguments and local variables will be declared 1 per line, with a comment field expressed with a "!" character followed by the comment text all on the same line as the declaration. Multiple comment lines describing a single variable are acceptable when necessary. Variables of a like function may be grouped together on a single line. For example:

```
INTEGER :: i,j,k ! Spatial indices
```

- **Continuation Lines** Continuation lines are acceptable on multi-dimensional array declarations which take up many columns. For example:

```
REAL(r8), DIMENSION(plond,plev), INTENT(in) :: &
array1, &! array1 is blah blah blah
array2   ! array2 is blah blah blah
```

  Note that the ISO/IEC 1539-1:1997 Fortran 95 standard defines a limit of 39 continuation lines.

  Code lines which are continuation lines of assignment statements must begin to the right of the column of the assignment operator. Similarly, continuation lines of subroutine calls and dummy argument lists of subroutine declarations must have the arguments aligned to the right of the "(" character. Examples of each of these constructs are:

```
a = b + c*d + ... + &
    h*g + e*f

CALL sub76 (x, y, z, w, a, &
            b, c, d, e)

SUBROUTINE sub76 (x, y, z, w, a, &
                  b, c, d, e)
```

- **Indentation** Code within loops and if-blocks will be indented 3 characters for readability.
- **Spacing Conventions**
  - Use blank space, in the horizontal and vertical, to improve readability. In particular try to align related code into columns. For example, instead of:

```
!  Initialize Variables
x=1
meaningfulname=3.0_wp
SillyName=2.0_wp
```

    write:

```
!  Initialize variables
x              = 1
meaningfulname = 3.0_wp
silly_name     = 2.0_wp
```

  - In general use a blank space after a comma (i.e. "a, b, c")
  - Do not use tab characters (an extension!) in your code: this will ensure that the code looks as intended when ported.

- **Formatted I/O** Avoid separating the information to be output from the formatting information on how to output it on I/O statements.

- **Argument List Format** Routines with large argument lists will contain 5 variables per line. This applies both to the calling routine and the dummy argument list in the routine being called. The purpose is to simplify matching up the arguments between caller and callee. In rare instances in which 5 variables will not fit on a single line, a number smaller than 5 may be used. But the per-line number must remain consistent between caller and callee. An example is:

```
CALL linemsbc (u3(i1,1,1,j,n3m1), v3(i1,1,1,j,n3m1), &
               t3(i1,1,1,j,n3m1), q3(i1,1,1,j,n3m1), &
               qfcst(i1,1,m,j), xxx)

SUBROUTINE linemsbc (u,     v, &
                     t,     q, &
                     qfcst, xxx)
```

- **Array Arguments** Do not implicitly change the shape of an array when passing it into a subroutine. Although actually forbidden in the FORTRAN77 standard it was very common practice to pass n dimensional arrays into a subroutine where they would, say, be treated as a one dimensional array. Though officially banned in Fortran 95, this practice is still possible with external routines for which no interface block is supplied. The danger of this method is that it makes certain assumptions about how the data is stored.

- **Commenting style** Short comments may be included on the same line as executable code using the "!" character followed by the description. More in-depth comments should be written in the form:

```
!
! Describe what is going on
!
```

  Key features of this style are 1) it starts with a "!" in column 1; 2) The text starts in column 3; and 3) the text is offset above and below by a blank comment line. The blank comments could just as well be completely blank lines (i.e. no "!") if the developer prefers.

- **Operators** Use of the operators `<, >, <=, >=, ==, /=` is recommended instead of their deprecated counterparts `.lt., .gt., .le., .ge., .eq.`, and `.ne.` The motivation is readability. In general use the notation: `<Blank><Operator><Blank>`

- **File Format** Embedding multiple routines within a single file and/or module is allowed, encouraged in fact, if any of three conditions hold. First, if routine B is called by routine A and only by routine A, then the two routines may be included in the same file. This construct has the advantage that inlining B into A is often much easier for compilers if both A and B are in the same file. Practical experience with many compilers has shown that inlining when A and B are in different files often is too complicated for most people to consider worthwhile investigating.

  The second condition in which it is desirable to put multiple routines in a single file is when they are `"CONTAIN"`ed in a module for the purpose of providing an implicit interface block. This type of construct is strongly encouraged, as it allows the compiler to perform argument consistency checking across routine boundaries. An example is:

```
file 1:
      SUBROUTINE driver
      USE mod1
      REAL :: X, Y
      ...
      CALL sub1(X,Y)
      CALL sub2(Y)
      ...
      END SUBROUTINE

file 2:
      MODULE mod1

      PRIVATE
      REAL :: , var2
      PUBLIC sub1, sub2

      CONTAINS

      SUBROUTINE sub1(A,B)
      ...
      END SUBROUTINE

      SUBROUTINE sub2(A)
      ...
      END SUBROUTINE

      END MODULE
```

The number, type, and dimensionality of the arguments passed to sub1 and sub2 are automatically checked by the compiler.

The final reason to store multiple routines and their data in a single module is that the scope of the data defined in the module can be limited to only the routines which are also in the module. This is accomplished with the `"private"` clause.

If none of the above conditions hold, it is not acceptable to simply glue together a bunch of functions or subroutines in a single file.

- **Module Names** Modules MUST be named the same as the file in which they reside. The reason to enforce this as a hard rule has to do with the fact that dependency rules used by "make" programs are based on file names. For example, if routine A `"USE"`s module B, then "make" must be told of the dependency relation which requires B to be compiled before A. If one can assume that module B resides in file `B.o`, building a tool to generate this dependency rule (e.g. `A.o: B.o`) is quite simple. Put another way, it is difficult (to say nothing of CPU-intensive) to search an entire source tree to find the file in which module B resides for each routine or module which `"USE"`s B.

  Note that by implication multiple modules are not allowed in a single file.

  The use of common blocks is deprecated in Fortran 95 and their continued use in the PRISM component is strongly discouraged. Modules are a better way to declare static data. Among the advantages of modules is the ability to freely mix data of various types, and to limit access to contained variables through use of the `ONLY` and `PRIVATE` clauses.

- **Array Syntax** Array notation should be used whenever possible. This should help optimization regardless what machine architecture is used (at least in theory) and will reduce the number of lines of code required. To improve readability the array shape should be shown in brackets, e.g.:

```
onedarraya(:) = onedarrayb(:) + onedarrayc(:)

twodarray(:, :) = scalar * anothertwodarray(:, :)
```

When accessing sections of arrays, for example in finite difference equations, do so by using the triplet notation on the full array, e.g.:

```
twodarray(:,2:len2) = scalar                          &
                    * (anothertwodarray(:,1:len2-1) &
                    - anothertwodarray(:,2:len2))
```

`Note:` In order to to improve readability of long, complicated loops, explicitly indexed loops should be preferred. In general when using this syntax, the order of the loops indices should reflect the following scheme: (best usage of data locality).

```
DO k = 1, nk
   DO j = 1, nj
      DO i = 1, ni
         f(i, j, k) = ...
      END DO
   END DO
END DO
```

3.1.3 Content Rules

- **Implicit None** All subroutines and functions will include an `"implicit none"` statement. Thus all variables must be explicitly typed. It also allows the compiler to detect typographical errors in variable names. For `MODULE`s, one `IMPLICIT NONE` statement in the modules definition section is sufficient.

- **Prologues** Each function, subroutine, or module will include a prologue instrumented for use with the ProTeX auto-documentation script (http://dao.gsfc.nasa.gov/software/protex). The purpose is to describe what the code does, possibly referring to external documentation. The prologue formats for functions and subroutines, modules, and header files are shown. In addition to the keywords in these templates, ProTeX also recognizes the following:

```
!BUGS:
!SEE ALSO:
!SYSTEM ROUTINES:
!FILES USED:
!REMARKS:
!TO DO:
!CALLING SEQUENCE:
!CALLED FROM:
!LOCAL VARIABLES:
```

These keywords may be used at the developer's discretion. We would like to recommend to use the REMARKS part to add an responsible author, who can be contacted in case of problems.

Prologue for Functions and Subroutines

If the function or subroutine is included in a module, the keyword `!IROUTINE` should be used instead of `!ROUTINE`.

```
!-------------------------------------------------------------------------
! BOP
!
! !ROUTINE:  <Function name>  (!IROUTINE if the function is in a module)
!
! !INTERFACE:
            function <name> (<arguments>)

! !USES:
            use <module>

! !RETURN VALUE:
            implicit none
            <type> :: <name>               ! <Return value description>

! !PARAMETERS:
            <type, intent> :: <parameter>  ! <Parameter description>

! !DESCRIPTION:
! <Describe the function of the routine and algorithm(s) used in
! the routine.  Include any applicable external references.>
!
! !REVISION HISTORY:
! YY.MM.DD  <Name> <Description of activity>
!
! EOP
!-------------------------------------------------------------------------
! $Id: SEG_Fortran_Conv.tex,v 1.1 2002/10/03 13:27:54 amangili Exp $
! $Author: amangili $
!-------------------------------------------------------------------------
```

Prologue for a Module

```
!-------------------------------------------------------------------------
! BOP
!
! !MODULE:  <Module name>
!
! !USES:
        use <module>
! !PUBLIC TYPES:
            implicit none
            [save]

            <type declaration>

! !PUBLIC MEMBER FUNCTIONS:
!           <function>                      ! Description
!
! !PUBLIC DATA MEMBERS:
            <type> :: <variable>            ! Variable description

! !DESCRIPTION:
! <Describe the function of the module.>
!
! !REVISION HISTORY:
! YY.MM.DD  <Name> <Description of activity>
!
! EOP
!-------------------------------------------------------------------------
! $Id: SEG_Fortran_Conv.tex,v 1.1 2002/10/03 13:27:54 amangili Exp $
! $Author: amangili $
!-------------------------------------------------------------------------
```

Prologue for a Header File

```
!-------------------------------------------------------------------------
! BOP
!
! !INCLUDE:  <Header file name>
!
! !DEFINED PARAMETERS:
            <type> :: <parameter>          ! Parameter description

! !DESCRIPTION:
! <Describe the contents of the header file.>
!
! !REVISION HISTORY:
! YY.MM.DD  <Name> <Description of activity>
!
! EOP
!-------------------------------------------------------------------------
! $Id: SEG_Fortran_Conv.tex,v 1.1 2002/10/03 13:27:54 amangili Exp $
! $Author: amangili $
!-------------------------------------------------------------------------
```

- **I/O Error Conditions** I/O statements which need to check an error condition will use the "`iostat=<integer variable>`" construct instead of the outmoded `end=` and `err=`. Note that a 0 value means success, a positive value means an error has occurred, and a negative value means the end of record or end of file was encountered.
- **Intent** All dummy arguments must include the `INTENT` clause in their declaration. This is extremely valuable to someone reading the code, and can be checked by compilers. An example is:

```
SUBROUTINE sub1 (x, y, z)
IMPLICIT NONE
REAL(r8), INTENT(in) :: x
REAL(r8), INTENT(out) :: y
REAL(r8), INTENT(inout) :: z

y = x
z = z + x

END SUBROUTINE
```

3.1.4 Package Coding Rules

The term "package" in the following rules refers to a routine or group of routines which takes a well-defined set of input and produces a well-defined set of output. A package can be large, such as a dynamics package, which computes large scale advection for a single timestep. It can also be relatively small, such as a parameterization to compute the effects of gravity wave drag.

- **Self-containment** A package should refer only to its own modules and subprograms and to those intrinsic functions included in the Fortran95 standard. This is crucial to attaining plug-compatibility. An exception to the rule might occur when a given computation needs to be done in a consistent

manner throughout the model. Thus for example a package which requires saturation vapor pressure would be allowed to call a generic routine used elsewhere in the main model code to compute this quantity.

When exceptions to the above rule apply, (i.e. routines are required by a package which are not f95 intrinsics or part of the package itself) the required routines which violate the rule must be specified within the package.

- **Single entry point** A package shall provide separate setup and running procedures, each with a single entry point. All initialization of time-invariant data must be done in the setup procedure and these data must not be altered by the running procedure. This distinction is important when the code is being run in a multitasked environment. For example, constructs of the following form will not work when they are multitasked:

```
SUBROUTINE sub
LOGICAL first/.true./
IF (first) THEN
   first = .false.
   <set time-invariant values>
END IF
```

- **Interface Blocks** Explicit interface blocks are required between routines if optional or keyword arguments are to be used. They also allow the compiler to check that the type, shape and number of arguments specified in the `CALL` are the same as those specified in the subprogram itself. Fortran 95 compilers can automatically provide explicit interface blocks for routines contained in a module.

- **Communication** All communication with the package will be through the argument list or namelist input. The point behind this rule is that packages should not have to know details of the surrounding model data structures, or the names of variables outside of the package. A notable exception to this rule is model resolution parameters. The reason for the exception is to allow compile-time array sizing inside the package. This is often important for efficiency.

- **Package Attributes** Modules variables and routines should be encapsulated by using the `PRIVATE` attribute. What shall be used outside the module can be declared `PUBLIC` instead. Use `USE` with the `ONLY` attribute to specify which of the variables, type definitions etc. defined in a module are to be made available to the using routine. Of course you don't need to add the `ONLY` attribute if you include the complete module or almost all of its public declarations.

- **Precision** Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size. The f95 `KIND` feature should be used instead.

  In order also improve portability between 32 and 64 bit platforms, it is necessary to make use of kinds by providing a specific module declaring the "kind definitions" to obtain the required numerical precision and range as well as size of `INTEGER`. It should be noted that constants need to have attached a `_kindvalue` to have the according size.

  Example, with dp as a real kind with 12 significant digits (usually double precision, 8 byte), and wp a working precision:

```
USE my_kind, ONLY: dp, wp
IMPLICIT NONE

! Declaration of a constant
REAL(dp), PARAMETER :: pi = 3.14159265358979323846_dp
```

```
! Declaration of a 3d field
REAL(wp), POINTER :: geopotential(:,:,:)

! Declaration of a local variable
REAL(wp) :: a
...

a = 4.0_wp
...
```

Possible kinds in a proposed `my_kind`:

kind variable	assumed number of bits
real variables	
sp	32
dp	64
integer variables	
i4	32
i8	64
integer to contain a C pointer	
cp	bit size of a C pointer

Note:
The bit sizes given are not mandatory. The kind value is selected regarding a given precision, which results on current systems in this bit sizes. This may change in future.

A portable way to build a module providing several kinds is given in the following example:

```
MODULE my_kind

     IMPLICIT NONE

          ! Number model from which the SELECTED_*_KIND are requested:
          !
          !                    4 byte REAL      8 byte REAL
          !          CRAY:        -            precision =   13
          !                                    exponent  = 2465
          !          IEEE:    precision =  6   precision =   15
          !                   exponent  = 37   exponent  =  307
          !
          ! Most likely this are the only possible models.

          ! Floating point section

          INTEGER, PARAMETER :: sp = SELECTED_REAL_KIND(6,37)
          INTEGER, PARAMETER :: dp = SELECTED_REAL_KIND(12,307)

          INTEGER, PARAMETER :: wp = dp   ! working precision
```

```
  ! Integer section

  INTEGER, PARAMETER :: i4 = SELECTED_INT_KIND(9)
  INTEGER, PARAMETER :: i8 = SELECTED_INT_KIND(14)

  ! Pointer section, the cp type should provide an integer
  ! with a size for transporting a C pointer through Fortran

#ifdef CP4
           INTEGER, PARAMETER :: cp = i4
#else
           INTEGER, PARAMETER :: cp = i8
#endif


END MODULE my_kind
```

Thus, any variable declared `real(dp)` will be of sufficient size to maintain 12 decimal digits in their mantissa. Likewise, integer variables declared `integer(i8)` will be able to represent an integer of at least 13 decimal digits.

- **Bounds checking** All PRISM software and PRISM components must be able to run when a compile-time and/or run-time array bounds checking option is enabled. Thus, constructs of the following form are disallowed:

```
REAL(r8) :: arr(1)
```

  where `"arr"` is an input argument into which the user wishes to index beyond 1. Use of the `(*)` construct in array dimensioning to circumvent this problem is forbidden because it effectively disables array bounds checking.

- **Error conditions** When an error condition occurs inside a package, a message describing what went wrong will be printed. The name of the routine in which the error occurred must be included. It is acceptable to terminate execution within a package, but the developer may instead wish to return an error flag through the argument list. If the user wishes to terminate execution within the package, a generic PRISM Coupler termination routine `"endrun"` should be called instead of issuing a Fortran `"stop"`. Otherwise a message-passing version of the model could hang. Note that this is an exception to the package coding rule that "A package should refer only to its own modules and subprograms and to those intrinsic functions included in the Fortran 95 standard".

- **Inter-procedural code analysis** Use of a tool to diagnose problems such as array size mismatches, type mismatches, variables which are defined but not used, etc. is strongly encouraged. Flint is one such tool which has proved valuable in this regard. It is not a strict rule that all PRISM codes and packages must be "flint-free", but the developer must be able to provide adequate explanation for why a given coding construct should be retained even though it elicits a complaint from flint. If too many complaints are issued, the diagnostic value of the tool diminishes toward zero.

- **Memory management** The use of dynamic memory allocation is not discouraged because we realize that there are many situations in which run-time array sizing is desirable. However, this type of memory allocation can cause performance problems on some machines, and some debuggers get confused when trying to diagnose the contents of such variables. Therefore, dynamic memory allocation is allowed

only "when necessary". The ability to run a code at a different spatial resolution without recompiling is not considered to be an adequate reason to use dynamically allocated arrays.

The preferable mechanism for dynamic memory allocation is automatic arrays, as opposed to `ALLOCATABLE` or `POINTER` arrays for which memory must be explicitly allocated and deallocated. An example of an automatic array is:

```
SUBROUTINE sub(n)
REAL :: a(n)
...
END SUBROUTINE
```

The same routine using an allocatable array would look like:

```
SUBROUTINE sub(n)
REAL, ALLOCATABLE :: a(:)

ALLOCATE(a(n))
...
DEALLOCATE(a)
...
END SUBROUTINE
```

- **Constants and Magic Numbers** Magic numbers should be avoided. Physical constants (e.g. pi, gas constants) must NEVER be hardwired into the executable portion of a code. Instead, a mnemonically named variable or parameter should be set to the appropriate value, probably in the setup routine for the package. We realize than many parameterizations rely on empirically derived constants or fudge factors, which are not easy to name. In these cases it is not forbidden to leave such factors coded as magic numbers buried in executable code, but comments should be included referring to the source of the empirical formula.

  Hard-coded numbers should never be passed through argument lists. One good reason for this rule is that a compiler flag, which defines a default precision for constants, cannot be guaranteed. Fortran95 allows specification of the precision of constants through the "_" compile-time operator (e.g. 3.14_dp or 365_i8). So if you insist on passing a constant through an argument list, you must also include a precision specification. If this is not done, a called routine which declares the resulting dummy argument as, say, `real(dp)` or 8 bytes, will produce erroneous results if the default floating point precision is 4 byte reals.

  Some physical variables/constants which are shared between several PRISM model components will be distributed by the PRISM software.

3.1.5 Interface to Other Languages

- **Fortran / C Interface** Interfaces between Fortran and C (C routines called by FORTRAN) should be based on the cfortran package (http://wwwinfo.cern.ch/asd/cernlib/cfortran.html).

3.1.6 Code Parallelization Issues

- **Parallelization Paradigms:** Parallelization should be done with well documented and commonly accepted paradigms like MPI, OpenMP and Pthreads.

- **Threadsafe:** The coupler and other modules callable by any component of the PRISM system should be threadsafe to allow for a flexible usage of hierarchical programming models.

- **Scalability:** Each module of the coupler and the components to be coupled should follow coding strategies for parallelization which allow for runs on high processor counts. Parallelized model components should be programmed flexible with regard to the number of processors which can be used for parallel runs. If necessary hierarchical programming techniques should be used to achieve that goal.

- **Data Locality:** Regarding to parallelization the coding techniques should be concerned about memory hierarchies of modern computer architectures. A key to parallel efficiency is the locality of the memory references. On a SMP-like architecture one should constrain memory references within a compute node as much as possible rather than excessively accessing remote memory location since networks between compute nodes of these SMP-like architectures are slower than intra-node memory paths in general.

  A similar point of view is valid for intra-node memory hierarchies like caches.

- **Data and process placement:** The environment of a parallel run (scripts, configuration files, etc) and the parallelized components of a PRISM coupled system should allow for a flexible placement or mapping of the various tasks of a coupled simulation onto the computer resources like compute nodes, file systems, etc.

3.2 C Coding Standard

This section defines a set of C specifications, rules, and recommendations for the coding of PRISM components. The purpose is to provide a framework that enables users to easily understand or modify the code, or to port it to new computational environments. In addition, it is hoped that adherence to these guidelines will facilitate the exchange and incorporation of new packages and parameterizations into the model. We would like to adhere to the "GNU Coding Standard" Chap. 5 "Making the Best Use of C", (http://www.gnu.org/prep/standards_toc.html).

Some of the obvious rules already given in the section before and an extension to the GNU Coding Standard will be repeated here.

3.2.1 Restriction to the Language

- **C ANSI Standard** The PRISM coupler will adhere to the ANSI C language standard with POSIX extensions (ISO/IEC 9899:1990 compliant) and not rely on any specific language or vendor extension. The purpose is to enhance portability across different platforms. If a situation arises in which there is good reason to violate this rule and include C code which is not compliant with the ANSI C standard, an alternate set of ANSI C compliant code must be provided (tested and validated...). This can be achieved by use of the C-preprocessor.

- **English Language** Owing to the international user community, all naming of variables, modules, functions, and subroutines as well as all comments are to be written in English.

3.2.2 Content Rules

C code should include comment blocks detailing code behaviour according to the rules of the open-source auto-documentation system doxygen (http://www.doxygen.org/) whose user manual is available online at (http://www.stack.nl/~dimitri/doxygen/manual.html).

Before each function or class declaration a comment block should be added exactly detailing the calling interface and the optional return values. The purpose is to exactly describe what the code does, possibly referring to external documentation.

Doxygen comment blocks for C source are in the following format:

```
/*! \fn test
   ... text ...
    \param c1 an integer
    \param f2 a float
    \return a character pointer
 */
char *test(int c1; float f2) {
   ...
}
```

and must precede the declaration, rather than the definition of the code unit to be documented.
Special commands are used to indicate the type of program unit that is being documented, they are:

- `\class` to document a C++ class
- `\struct` to document a C-struct
- `\union` to document a union
- `\enum` to document an enumeration type
- `\fn` to document a function
- `\var` to document a variable or typedef or enum value
- `\def` to document a #define
- `\file` to document a file
- `\namespace` to document a namespace

Many more special commands are used for text-formatting. See the doxygen manual for a description of those and many detailed examples.

3.2.3 Interface to Other Languages

- **Fortran / C Interface** Interfaces between Fortran and C (Fortran routines called by C) should be based on the cfortran package (http://wwwinfo.cern.ch/asd/cernlib/cfortran.html)

3.2.4 Code Parallelization Issues

Please refer to "Code Parallelization Issues" in section 3.1.6

4 Component and Unit Testing

Quite complex and interdependent software such as PRISM components requires extensive testing in order to prevent system defects and to provide stable, reliable, and solid software to work with.

Layered testing has shown to be the most effective in catching software defects. Layered testing refers to testing on different levels, both testing individual subroutines as well as more complex systems. There may be several layers of simple to more complex systems tested as well. Testing the individual component models stand-alone is an example of a system less complex than the entire PRISM system.

Unit testing is the first layer, meaning testing individual subroutines or modules. Unit testing by itself will not catch defects that are dependent on relationships between different modules, but testing the entire system sometimes will not catch errors within an individual module. That is why using both extremes is useful in catching model defects. Section 5 covers testing for the entire PRISM system, this section goes over testing of individual model components and unit testing of subroutines and modules within those components.

Since, the PRISM system and PRISM components take substantial computer resources to run, catching errors early can also cut computing costs significantly. In addition to that as pointed out by McConnell-1993[7] development time decreases dramatically when formal quality assurance methods including code-reviews are implemented as described in section 7.

4.1 Designing Good Component Tests

Each PRISM component needs to develop and maintain it's own suite of testing for that given component. Each component development team should provide a written analysis of the testing-procedure and testing-plan required by the components they are developing. An automated procedure has to be provided to run a suite of standard tests, the result of this procedure is either a **'PASSED'** or **'FAILED'** result, this will be useful to ensure the models work and continue to work as needed. This is especially useful for making sure PRISM components continue to work on multiple platforms.

In order to design a comprehensive testing plan we want to take advantage of the following types of tests:

- **Unit-tests:** Testing done on a single subroutine or module.
- **Functional-tests:** Testing for a given functional group of subroutines or modules, for example, testing model dynamics alone without the model physics.
- **Component-tests:** Testing done on the whole component.

4.1.1 Unit-tests

Unit tests are a good way to flush out certain types of defects. Since unit tests only run on one subroutine they are easier to use, faster to build and run, allow more comprehensive testing on a wider range of input data, help document how to use and check for valid answers, and allows faster testing of individual pieces. By building and maintaining unit tests the same tests can be run and used by other developers as part of a more comprehensive testing package. Without maintaining unit tests developers often do less testing than required (since component tests are so much harder to do) or they have to "hack" together their own unit tests for each change. By maintaining unit tests we allow others to leverage off previous work and provide a format to quickly do extensive checking.

Good unit tests will do the following:

1. Applicable requirements are checked.
2. Exercise every line of code.
3. Check that the full range of possible input data works. (i.e. if Temperature is input check that values near both the minimum and maximum possible values work)

4. Boundary analysis. Logical statements that refer to threshold states are checked to ensure they are correct.

5. Check for bad input data.

6. Test for scientific validity.

By analyzing the code to be tested different test cases can be designed to ensure that all logical statements are exercised in the unit-test. Similarly input can be designed to test logical threshold states (boundary analysis). Testing scientific validity is of course the most difficult. But, sometimes testing states where the answer is known analytically can be useful. And ensuring (or measuring) the degree to which energy, heat, or mass is conserved for conservative processes can also often be done. These types of tests may also be applied for more complex functional and component tests as well.

4.1.2 Functional-tests

Functional tests take a given sub-set of the component and test this set for a particular functionality. Scientific functional tests are common. Important functional tests should be maintained in the source code revision system as separate modules that include the directories maintained for the main component and be distributed with the PRISM component.

4.1.3 Component-tests

Component-tests need to ensure that the given PRISM component compiles, builds, and runs and that it passes important model requirements. For example, restart capabilities.

4.1.4 PRISM testing requirements and implementation details

Unit and Functional tests should meet the following minimum requirements:

1. Maintained in CVS either with the rest of the models source code or as a separate module that can be used. Module and/or directory name should be easily identifiable such as 'unit_test'.

2. Have documentation on how to use it.

3. Have a Makefile associated with it. It may be useful to leverage off the main Makefile so that the compiler options are the same and so that platform dependencies don't have to be maintained twice.

4. Check error conditions so that an error will print out problems.

5. Interactive test should be avoided, unless a completely automatic test can be set-up using the same mechanism (i.e. redirecting the input).

6. In general unit tests should be run with as many compiler debug options on as possible (bounds checking, signal trapping etc) as well as using optimized options, making sure the results are comparable.

Component tests should meet the following minimum requirements:

1. Ensure that the given model will compile, build and run on at least one production platform.

2. Ensure that the given model will work with the PRISM system on at least one production platform.

5 System Testing and Validation

Regular system testing and validation of the whole PRISM system is required to ensure that components quality and integrity is maintained throughout the development process. This section establishes the system testing standards and the procedures that will be used to verify the standards have been met. It is assumed that PRISM component development teams have 'unit-tested' their component prior to making it available for system testing. See section 4 for more information on testing of individual components and unit-testing of individual subroutine and modules within components.

There are two general categories of model evaluations: *frequent short* test runs and *less frequent long* validation integrations.

Model testing refers to short (few days to one month) model runs designed to verify that the underlying mechanics and performance of the coupled models continues to meet specifications. This includes verifying that the model actually starts up and runs, benchmarking model performance and relative speed/cost of each model component as well as checking features like restarting capabilities. These tests are done on each of the target platforms. Model testing does not address whether the model answer is correct, it merely verifies that it mechanically operates as specified.

Model validation involves longer (at least 1 year) integrations to ensure that the model results are in acceptable agreement with both previous model climate statistics and observed characteristics of the real climate system. Model validation occurs with each minor PRISM system version (i.e. PRISM_2.1), PRISM_2.2) or on request (PRISM scientists or working groups). Once requested, model validation is only carried out after PRISM scientists have been consulted and the 'Model testing' phase has been successfully completed. The model validation results are documented on a publicly accessible web page (http://www.prism.enes.org/coupler/PRISM_X.Ybeta/tests/index.html).

Port validation is defined as verification that the differences between two otherwise identical configuration simulations obtained on different machines or using different environments are caused by machine roundoff errors only.

5.1 Model Testing Procedures for the PRISM system

Formal testing of the PRISM system is required for each tagged version of the model. The PRISM quality assurance leader is responsible for ensuring that these tests are run, either by personally doing it or having them run by a qualified person. If a model component is identified as having a problem, the component development team is expected to make resolving that problem their highest priority. The results of the testing and benchmarking will be included in the tagged model to document the run characteristics of the model. The actual testing and analysis scripts will be part of the PRISM software code repository to encourage use by outside users.

5.1.1 Development Testing Steps

1. *Successful build:* PRISM shall compile on each of the target platforms with no changes to the scripts, codes or datasets.

2. *Successful startup:* PRISM will start from an initial state and run for 10 days.

3. *Performance benchmarking:* The total CPU time, memory usage, output volume, disk space use and wall clock time for the 10 day run will be recorded. The relative cost of each component will also be recorded.

4. *Test report:* The results of all steps above are to be documented in a test report with emphasis on results, comparisons to the previous test and recommendations for improvements. Any faults or defects observed shall be noted and must be brought to the attention of the responsible for that component and the software engineering manager.

5.2 Model Validation Procedures for the PRISM System

Model Validation occurs with each minor PRISM version (i.e. PRISM_2.1, PRISM_2.2) or at the request of the PRISM scientists and working groups. Before starting a validation run, the PRISM Quality Assurance Leader will consult with the PRISM scientists to design the validation experiment.

Pre-Validation Steps:

1. *Tests run successfully* The validation will successfully complete the testing steps outlined above.
2. *Scientist sign-on* The PRISM scientists must agree to make themselves available to informally analyze the results of the run during the run and formally review the results within one week of the completion of the run.

Validation Steps:

1. *Comparison with previous model runs* Result agrees with previous model runs
2. *Comparison with observed climate* Result agrees with observed climate

5.3 Port Validation of the PRISM System

Port validation is defined as verification that the differences between two otherwise identical model simulations obtained on different machines or using different environments are caused by machine roundoff errors only. Roundoff errors can be caused by using two machines with different internal floating point representation, or by using a different number of processing elements on the same machine which may cause a known re-ordering of some calculations, or by using different compiler versions or options (on a single machine or different machines) which parse internal computations differently.

The following paper offers a primary reference for port validation:

Rosinski, J.M. and D.L. Williamson: The Accumulation of Rounding Errors and Port Validation for Global Atmospheric Models. SIAM Journal on Scientific Computation, Vol. 18, No. 2, March 1997, pp.552-564.

As established in this article, three conditions of model solution behavior must be fulfilled to successfully validate a port of atmospheric general circulation models:

1. during the first few timesteps, differences between the original and ported solutions should be within one to two orders of magnitude of machine rounding;
2. during the first few days, growth of the difference between the original and ported solutions should not exceed the growth of an initial perturbation introduced into the lowest-order bits of the original solution;
3. the statistics of a long simulation must be representative of the climate of the model as produced by the original code.

The extent to which these conditions apply to models other than an atmospheric model has not yet been established. Also, note that the third condition is not the focus of this section.

6 Code Maintenance Issues

This section summarize code maintenance issues and challenges related to the PRISM project and gives an brief overview on how this problem will be addressed. The software maintenance process is a tedious and often costly activity, it's a fundamental part of the software engineering process that can not be just avoided or simply ignored. The software engineering literature is plenty of examples of died projects simply because the maintenance costs just exploded. Still several actions and measures can be undertaken in order to minimize the effort of the maintenance process. In the PRISM project we tried to find out the right compromise of software engineering rules and conventions being conscious and considering that most of the PRISM components are software packages already developed with their own software life cycle and consequently their own software maintenance process already in place since years.

It is recommended that all PRISM components follow some general guidelines for basic code maintenance as a code revision control and a code configuration, building. The goal is to provide a single code repository with a consistent and unique revision control scheme, as well as to provide an easier port, installation, building and validation on different architectures.

6.1 Software Revision Control

It is recommended that all PRISM development teams develop code with the help of a robust software configuration management (SCM) tool. The goal is to establish, document, control, and track the evolution of source code and related documentation throughout the software life cycle.

As part of the PRISM system this software configuration management tool is CVS.

6.1.1 What is CVS?

CVS stands for Concurrent Versions System and is a version control system. Using it, one can record the history of his source files.

For example, bugs sometimes creep in when software is modified and one might not detect the bug until a long time after the modification was made. With CVS, one can easily retrieve old versions to see exactly which change caused the bug. This can sometimes be a big help.

CVS stores all the versions of a file in a single file in a clever way that only stores the differences between versions.

CVS also helps developers when they are part of a group of people working on the same project. It is all too easy to overwrite each others' changes unless one is extremely careful. Some editors, like GNU Emacs, try to make sure that the same file is never modified by two people at the same time. Unfortunately, if someone is using another editor, that safeguard will not work. CVS solves this problem by insulating the different developers from one another. Every developer works in his own directory and CVS merges the work when each developer is done.

One can get CVS in a variety of ways, including free download from the Internet. For more information on downloading CVS and other CVS topics, see:

http://www.cvshome.org/

http://www.loria.fr/~molli/cvs-index.html.

6.1.2 What is CVS not?

- **CVS is not a build system.**

  Though the structure of a repository and modules file interact with a build system (e.g. 'Makefile's), they are essentially independent.

  CVS does not dictate in any way how anything is built. It merely stores files for retrieval in a previously planned tree structure.

- **CVS is not a substitute for management.**

  For instance, the dates of schedules, release dates or branch points are up to to people working on a certain project. If certain milestones are not kept, CVS can't help.

- **CVS is not a substitute for communication.**

  A bug fix of a source module has to be reviewed before releasing it or checking it into the official source tree.

- **CVS does not have change control**

  Change control refers to a number of things. First of all it can mean bug-tracking, that is being able to keep a database of reported bugs and the status of each one (is it fixed? in what release? has the bug submitter agreed that it is fixed?). For interfacing CVS to an external bug-tracking system, see the 'rcsinfo' and 'verifymsg' files (see section C. Reference manual for Administrative files).

  Another aspect of change control is keeping track of the fact that changes to several files were in fact changed together as one logical change. If one has checked in several files in a single CVS commit operation, CVS then forgets that those files were checked in together, and the fact that they have the same log message is the only thing tying them together. Keeping a GNU style 'ChangeLog' can help somewhat.

  Yet another aspect of change control, in some systems, is the ability to keep track of the status of each change. Some changes have been written by a developer, others have been reviewed by a second developer, and so on. Generally, the way to do this with CVS is to generate a diff (using CVS diff or diff) and email it to someone who can then apply it using the patch utility. This is very flexible, but depends on mechanisms outside CVS to make sure nothing falls through the cracks.

- **CVS is not an automated testing program**

  It should be possible to enforce mandatory use of a test suite using the commit info file.

6.1.3 CVS Features

As mentioned CVS stores the changes of a source in a repository in a compact way . That repository can be located on a remote server. The access to the remote server is controlled by login id, password, file permissions and by the so-called CVS tags. For remote login it is even possible to define the remote shell. This can be even the secure shell which gives the administrator of the repository even more access control (Is the accessing host trusted? Did the user provide a public key? etc.).

CVS allows to define branches of a source tree and individual access rights for that branches. Complete individual branches can be accessed via tags which allow to check out sources which were logically built as one single unit, according to the changes which have been applied. Different branches can be merged.

CVS allows to check in and check out sources with automatic provision of a release number and history of changes applied.

One can compare releases of individual source files or complete branches.

CVS has a mechanism to ask the developer for comments concerning the changes applied before checking in. These comments are kept alongside with the changes.

CVS has a locking mechanism which prevents race conditions during a source check in into the same branch. A developer can query the status of file and get information about who else has checked out a file for editing.

Last but not least CVS provide keyword substitution, ie. the keyword '$Header$' is substituted by the full pathname of the RCS file, the revision number, the date (UTC), the author, the state, and the locker (if locked).

For more information we suggest to refer to the reference manual. Moreover, we would like to mention that a GUI clients exist:

Tk based: tkCVS (http://www.twobarleycorns.net/tkcvs.html)

Java based: jCVS (http://www.jcvs.org/)

6.1.4 CVS Repository Access

The PRISM CVS Repository will be installed and accessible under http://prism.enes.org/.

6.1.5 Recommendations for Code Developers

We strongly recommend the use of the keyword '$Header$'. CVS keywords are often only used within source code comments, we encourage developers to additionally define a static string variable storing the expanded keyword information. The advantage consists in the possibility of tracking module revisions within the binary, by using tools like 'whatis' or 'rcsinfo'. This greatly help bug analysis:

After checking out a code for editing the developer may therefore insert these lines in its program. In either C:

```
static char PRISM_CVSID[]="$Header: /home/amangili/Proj/PRISM/Work/CodingRules/ARCDI_II_5/RCS/SEG
```

or Fortran 95:

```
CHARACTER(len=80), PARAMETER:: PRISM_CVSID='$Header: /home/amangili/Proj/PRISM/Work/CodingRules/A
```

6.2 Code Maintenance Process

The maintenance process is related to activities and tasks of the software maintainer(s). This process is activated when the software product undergoes modifications to code and associated documentation due to a problem or the requirement for improvement or adaptation. The objective is to modify an existing software product while preserving its integrity.

This process briefly consists of the following activities (see also:[3])

- **Problem and Modification Reporting Procedure:** We shall establish procedures for receiving, recording and tracking problem reports and modification requests from the PRISM users. Whenever problems are encountered, they shall be recorded, documented and made 'public' in order to be aware of the 'potential' problem related to a specific PRISM version. For the tracking mechanism we propose the use of the free-software RT (Request Tracker). *"RT is an industrial-grade ticketing system. It lets a group of people intelligently and efficiently manage requests submitted by a community of users. RT is used by systems administrators, customer support staffs, NOCs, developers and even marketing departments at over a thousand sites around the world."*

  For more informations please see:
  (http://www.fsck.com/projects/rt/) or (http://freshmeat.net/projects/requesttracker/).

- **Problem and Modification Analysis:** The maintainer shall analyze the problem report or modification requests for its impact on the organization, the existing system, and the interfacing systems for the following:

  - **type** (e.g. corrective, improvement, preventive, or adaptive to new environment);
  - **scope** (e.g. size of modification, cost involved, time to modify);

– **criticality** (e.g. impact on performance; scientific results).

The maintainer shall be able to reproduce or verify the problem. Based upon the analysis, the maintainer shall develop options for implementing the modification. The maintainer shall document the problem/modification request, the analysis results and implementation options.

- **Implementation of the Modifications:** The maintainer shall conduct analysis and determine which documentation, software units, and versions there of shall be modified. The maintainer shall implement the modifications. The requirements of the development process shall be supplemented as follows:

  Test and evaluation criteria for testing and validating the modified as well as the unmodified parts (basic software units, components, ...) of the system shall be defined and documented.

  The complete and correct implementation of the new and modified requirements shall be ensured following the usual PRISM Components and System Validation Conventions (as described in section 4, and 5)

- **Maintenance Review/Acceptance:** The maintainer shall conduct a joint Peer Review(s) to determine the integrity of the modified component(s) and system. Upon successful completion of the reviews, a baseline for the change shall be established, defining the importance and impact of the new revision on the entire PRISM system (minor or major revision, main line or branches, etc...).

7 Code Quality Assurance

By adopting quality assurance techniques during the development process of PRISM components, the code generated will be of greater quality implying that development and integration times, as well as maintenance effort, can be substantially lowered. The later in the software life cycle an error is detected the higher is the cost of correcting it.

For the development of PRISM components we proposed several mechanisms targeting quality assurance, a summary is reported below here:

- **Coding Rules and Conventions** Detailed coding rules and coding conventions have been defined. This should in particular improve code portability among different platforms, code readability and understanding, especially when the code has not been written by the same person.
- **Automatic Code Documentation Extraction** The proposed standard, based on the ProTex tool for commenting and documenting header files, modules, routines and interfaces in general, allow automatic extraction of technical documentation directly from the source code. The main advantage of this mechanism is to ensure a consistent and up-to-date code documentation at any time.
- **Code Revision Control and Versioning** All PRISM components, libraries as well as validation test suites (test applications and test data) will be centralized and managed by a Code Revision Control mechanism in order to ensure a coherent and always up-to-date version of the PRISM system. This mechanism allows in particular to also easily access early versions of the software, making one able to back-trace problems on previous versions.
- **Components and System Validation** Test and validation suites will be part of the software development process and will ensure a rapid and early discovering of software problems in general; from bugs, performance and portability issues, up to scientific discrepancy on the expected results. We would like to push in the direction of an automatic test and validation suite that can be executed periodically or on demand on a predefined list of target platforms.

In addition another mechanism that we would like to introduce for improving code quality is code-review as described in the following section.

7.1 Code-Reviews

Formal reviews of the code where the code is gone through line-by-line in groups or in pairs has shown to be one of the most effective way to catch errors McConnell-1993 [7]. As such it is recommended that component model development teams create a strategy for regularly reviewing the code.

7.1.1 Possible Implementation of Code-Reviews

Code-reviews can be implemented in many different fashions, but in general they involve having at least one person besides the author go through the written code and examine the implementation both for design and errors. Jones-1986[5] states that "the average defect-detection rate is only 25 percent for unit testing, 35 percent for function testing, and 45 percent for integration testing. In contrast, the average effectiveness of design and code inspections are 55 percent and 60 percent. McConnel-1993[7] also notes that as well as being more effective in catching errors, code-reviews also catch different types of errors than testing does. In addition when developers realize their code will be reviewed they tend to be more careful themselves when programming.

Code reviews can be implemented in different ways:

- **Code validator** Before code is checked into CVS it goes through a "validator" who not only is responsible for testing, and validation of the changes, but also reviews it for design and conformance to code standards.

- **Peer reviews** Before code is checked into CVS a peer developer reviews the changes.
- **Pair programming** All code is developed with two people looking at the same screen (one of the practices of Extreme Programming [2]).
- **Formal group walk-through** Code is presented and gone through by an entire group.
- **Formal individual walk-through** Different individuals are assigned and take responsibility to review different subroutines.

7.1.2 Recommendation

It is recommended that development teams provide both a mechanism to review incremental changes, and also have formal walk-through of important pieces of code in group. This serves two purposes: the design is communicated to a larger group, and the design and implementation is also reviewed by the entire group. The frequency of the review has to be defined in advance and should in particular consider the criticality of the code in respect to the entire PRISM system.

Appendix A: Testing Terminology

Industry-accepted definitions exist for software errors and defects (faults), found in the ANSI/IEEE, Glossary of Software Engineering Terminology, are listed in Table A1.

Table A1 – IEEE Software Engineering Terminology [1]

Category	**Definition**
Error	The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.
Fault	An incorrect step, process, or data definition in a computer program.
Debug	To detect, locate, and correct faults in a computer program.
Failure	The inability of a system or component to perform its required functions within specified performance requirements. It is manifested as a fault.
Testing	The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.
Static analysis	The process of evaluating a system or component based on its form, structure, content, or documentation.
Dynamic analysis	The process of evaluating a system or component based on its behavior during execution.
Correctness	- The degree to which a system or component is free from faults in its specification, design, and implementation. - The degree to which software, documentation, or other items meet specified requirements. - The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.
Verification	- The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. - Formal proof of program correctness.
Validation	The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

References

[1] IEEE Standard Glossary of Software Engineering Terminology. Technical Report IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, December 1990.

[2] Beck, K. *Extreme Programming Explained; Embrace Change*. Boston, Mass.: Addison-Wesley, 2000.

[3] European Cooperation for Space Standardization. Space Product Assurance: Software Product Assurance. Technical Report ECCS-Q-80A, ECSS, Requirements and Standard Division, April 1996.

[4] European Cooperation for Space Standardization. Space Engineering: Software. Technical Report ECCS-Q-40A, ECSS, Requirements and Standard Division, April 1999.

[5] Jones, C. *Programming Productivity*. New York, New York: McGraw-Hill, 1986.

[6] Mangili, A., et al. PRISM REDOC III.2: Review of Current and Future HPC Architectures. Technical Report , PRISM, June 2002.

[7] McConnell, S. *Code Complete*. Redmond, Wash.: Microsoft Press, 1993.

[8] McConnell, S. *Rapid Development*. Redmond, Wash.: Microsoft Press, 1996.