Space Weather Modeling Framework Maintenance Manual

Gábor Tóth, Darren De Zeeuw, Ovsei Volberg

Center for Space Environment Modeling
The University of Michigan

January 31, 2026

Contents

1 Git

Physics module and framework code is all maintained and developed through Git repositories at http://github.com/SWMFsoftw
A developer can check out either the whole framework with all physics modules, or any individual physics module and necessary common support code. All code changes must be tested (see TESTING.pdf) before committing back into the Git repository. A complete description of changes must be entered into the Git logs when committing any change.

2 How to Add a New Component

2.1 Requirements to Physics Modules and Components

The SWMF compliance definitions, both for physics modules and components created from them, were formulated in the SWMF Interoperability Document. The minimal set of requirements regarding the source code of a physics module only, which is the part of the physics module compliance definition, is briefly repeated below:

- The parallelization mechanism must employ the MPI standard only.
- A module must have the structure permitting the following two modes of execution:
  - As a stand-alone executable;
  - As a library which could be linked to another executable.
- A module must successfully run the test suite provided by its developers at least on
  - the Compaq ES45 machine,
  - the SGI Altix with the ifort compiler, and
  - Linux Beowulf clusters with the NAG F95 compiler.
- The module must adhere to the following Input/Output requirements:
  - Read input data from files only;
  - Write output data to files only;
  - The path to these files should be possible to change.
- A module must be implemented in the following languages only:
  - Fortran 77,
  - Fortran 90.
- The following is not permitted in the source code of the module:
  - Lack of modularity and absence of point entry for other software tools;
  - Input-output that cannot be turned off or redirected;
  - Absence of error handling in the code.
- The source code of the module should be documented at least through appropriate comments in source files.

The physics module should be converted to a SWMF component by constructing standard interfaces from programming blocks provided by the SWMF. The first standard interface is a wrapper. In addition to that for each link with another component the coupling interface must be constructed. The standard interfaces must at least enable the following features of a component:

- To be registered by the Control Module;
- To be initialized in parallel configuration;
- To accept and check input parameters obtained from the Control Module;
- To initialize for session execution and provide grid description to Control Module;
- To execute a time step and return the simulation time, which cannot exceed a specified maximum simulation time;
- To receive and provide data to other components via the coupler in the Control Module;
- To write its state into a restart file when requested;
- To finalize at the end of the execution.

There are restrictions regarding I/O operations and error handling:

- Input data must be read from the subdirectory named by the component ID;
- Output data must be written into the subdirectory named by the component ID;
- Errors should `call CON_stop` with an appropriate error message. No `stop` statements are allowed;
- Standard output should be prefixed with a string identifying the component and/or redirected into a file defined by the Control Module.

The inclusion of a new physics code as an alternative version of an existing component is a relatively simple task. The inclusion of a new component for a new physics domain is slightly more demanding task.

2.2 New Physics Code as an Alternative Component Version

To add a new physics code as an alternative version (for example FLAMPA) of an existing component (for example SP), a user should perform the following steps:

- Add a subdirectory for this code in the directory of the appropriate physics model (e.g. `SP/FLAMPA`).
- Create a source subdirectory (e.g. `SP/FLAMPA/src`) and put the physics code into this directory.
- Write a top-level Makefile (e.g. `SP/FLAMPA/Makefile`) which should have the targets:
  - `install`: installs the component version, for instance, selects the compiler specific compilation rules for files which could not be compiled with the generic rules;
  - `LIB`: builds the component library;
  - `clean`: cleans the source subdirectories;
  - `distclean`: runs ./Config.pl -uninstall to discard all the files created since installation;
  - `allclean`: used by ./Config.pl -uninstall to remove files created after installation;
  - `rundir`: copies files and directories that are needed to run the component into the directory `run`.

  Any of the existing component version Makefile's can be used as an example.

- Include the top level Makefile.conf into the source Makefile(s) (e.g. `SP/FLAMPA/src/Makefile`) and use the general rules for compilation. Any of the source Makefile's can be used.
- Write a wrapper file for the new component. This can be placed into `src` or a separate `srcInterface` directory (e.g. `SP/FLAMPA/srcInterface/SP_wrapper.f90`). The wrapper is constructed as a set of external subroutines which we will call methods below. The method names must have the prefix of the physics domain with underscore (e.g. `SP_`). The following methods are required for a new version of the SP component, for example:
  - `SP_set_param`,
  - `SP_init_session`,
  - `SP_run`,
  - `SP_save_restart`,
  - `SP_finalize`.
- Write a set of `put` and `get` routines for each component, with which your component should be coupled.
- Extend the couplers in `CON/Interface/src` as necessary.
- Provide a `PARAM.XML` file in the component version directory (i.e. `SP/FLAMPA/PARAM.XML`) to describe all the input commands. List this file in `doc/Tex/Makefile` to be included in the manual.
- Provide a `Config.pl` script in the component version directory (i.e. `SP/FLAMPA/Config.pl`) to facilitate installation, configuration, and uninstallation. The script can use the share/Scripts/Config.pl for all the common functionalities.
- Add the new component version in the main `Makefile` and in `CON/Makefile.def`.

2.3 Adding a New Physics Component

To include a new physics component which covers a new physics domain (for example Radiation Belt), it is necessary to perform the following steps:

- Create the subdirectory for this physics domain (e.g. `RB/`)
- Create version subdirectories for an empty and at least one working version (e.g. `RB/Rice/` and `RB/Empty/`);
- Add entries for this physics domain to the `CON_wrapper.f90` and `CON_couple_all.f90` files in `CON/Interface/src`;
- Add the new component with all versions in the main `Makefile`, in `CON/Makefile.def` and link it as a library in `CON/Control/src/Makefile`;
- Create the new coupling interfaces in `CON/Interface/src`.
- Add the new component and the new couplings in all input command definitions which list the components or couplings explicitly in `PARAM.XML`;
- Add the new component into the $ValidComp variable in `Config.pl` and in `share/Scripts/CheckParam.pl`;
- List the new component and the new couplings in a few commands in `PARAM.XML`;
- Do all the steps described in the previous subsection.

The files `CON/Interface/src/CON_wrapper.f90` and `CON_couple_all.f90` play the role of switchboards for all wrappers and coupling interfaces, respectively. In other words these two files emulate dynamic dispatching or run-time polymorphism in Fortran 90 written code. They allow to use a single subroutine name for a component method and resolve at run-time which particular component method was called.

3 Examples for Component Wrapper and Coupler

3.1 Example Wrapper for Component PM

This subsection provides explicit interfaces and examples for all the methods (subroutines) that must be defined by a component wrapper. The documentation is produced from the

`doc/Tex/PM_wrapper.f90`

file which can be used as a template for writing a new wrapper.

In this example the abbreviated name of the component (the component ID) is `PM` (which stands for Physics Module). If the file is used as a template for an actual wrapper, then the file should be placed into the appropriate component version interface directory named analogous to

`PM/PmVersion/srcInterface/PM_wrapper.f90`

Further the `PM` string should be replaced with the two-character ID of the actual component and/or the module, variable and subroutine names that belong to the physics module (all start with `PM_` in this file) should be replaced with the appropriate module, variable and subroutine names of the new physics module. For example if the component ID is 'IE' then the subroutine `PM_init_session` should be renamed to `IE_init_session`, while the variable `PM_iProc` in module `PM_ModProc` may have a completely different name.

The PM component wrapper provides the following six subroutines

- PM_set_param
- PM_init_session
- PM_run
- PM_save_restart
- PM_finalize

which will be described in detail below. All these subroutines are called by the control module from `CON/Interface/src/CON_wrapper`.

3.1.1 PM_set_param - set parameters for the PM module

INTERFACE:

```
subroutine PM_set_param(CompInfo, TypeAction)
```

USES:

```
use CON_comp_info, ONLY: CompInfoType, get, put   ! component info, access
use CON_coupler,   ONLY: PM_, set_grid_descriptor ! component ID, descriptor
use ModConst,      ONLY: cPi                      ! Pi = 3.1415...
use ModIoUnit,     ONLY: STDOUT_                  ! unit for STDOUT

use PM_ModProc, ONLY: PM_iComm, PM_iProc, PM_nProc ! MPI parameters
use PM_ModMain, ONLY: PM_Dt, PM_TimeAccurate       ! Some variables to set
use PM_ModSize, ONLY: PM_n, PM_m                   ! Grid size
use PM_ModIo,   ONLY: PM_iUnit, PM_Prefix  ! I/O unit and prefix for output

implicit none
```

INPUT/OUTPUT ARGUMENTS:

```
type(CompInfoType), intent(inout) :: CompInfo   ! Information for this comp
```

INPUT ARGUMENTS:

```
character (len=*), intent(in)     :: TypeAction ! What to do
```

LOCAL VARIABLES:

```
character (len=*), parameter :: NameSub = 'PM_set_param'

logical :: DoTest, DoTestMe
```

DESCRIPTION:

This subroutine serves multiple purposes. It is called multiple times with various values for the TypeAction argument. The CompInfo argument is a derived type that contains basic information about this component. The data in this object should be accessed with the get and put methods of the CON_comp_info class. This subroutine is first called with TypeAction='VERSION' so that the version name is provided by PM to CON. Next it is called with TypeAction='MPI' and CON provides the MPI communication group parameters to PM. The next actions are 'STDOUT' and possibly 'FILEOUT', which set the prefix and the I/O unit for the verbose output of PM. The 'STDOUT' and 'FILEOUT' actions may be called multiple times during the run. In each session the physics module reads its input parameters (TypeAction='READ') and checks the parameters ('CHECK'). The action 'READ' may be called multiple times before the 'CHECK' action. After the module is initialized for the session (with the PM_init_session method), the physics module provides a description of its grid to CON (TypeAction='GRID').

CONTENTS:

```
! Set DoTest and DoTesMe logicals. Check for the name of this subroutine
! in the StringTest parameter of the #TEST command in PARAM.in.
call CON_set_do_test(NameSub, DoTest, DoTestMe)
```

```
if(DoTest)write(*,*)NameSub,' called with TypeAction, iProc=',&
     TypeAction, PM_iProc

select case(TypeAction)
case('VERSION')
   ! Provide module name and version number for CON
   call put(CompInfo,                          &
        Use        = .true.,                   &
        NameVersion= 'CODE NAME (developers)', &
        Version    = 1.4 )

case('MPI')
   ! Set the MPI communicator, the processor rank, and number of PE-s
   call get(CompInfo, iComm=PM_iComm, iProc=PM_iProc, PM_nProc=PM_nProc)

case('STDOUT')
   ! Set PM_iUnit to STDOUT and prefix to 'PMx:' where x is the PE rank
   PM_iUnit=STDOUT_
   write(PM_Prefix,'(a,i1,a)') 'PM', PM_iProc, ':'

case('FILEOUT')
   ! Set PM_iUnit to the value provided by CON. No need for prefix.
   call get(CompInfo, iUnitOut=PM_iUnit)
   PM_Prefix=''

case('READ')
   ! Read input parameters for this component
   call read_param

case('CHECK')
   ! Get some basic parameters from CON
   call get_time(DoTimeAccurateOut = PM_TimeAccurate)

   ! Check interdependent parameters here for consistency

case('GRID')
   ! Provide grid description for the coupling toolkit.
   ! The grid is a uniform spherical grid divided into two blocks
   ! along the Theta coordinate: northern and southern hemispheres.
   ! Both hemispheres have a PM_n by PM_m grid.
   ! The module can run on 1 or 2 PE-s,
   ! so the processor array is (/0,0/) or (/0,1/)

   call set_grid_descriptor(          &
        PM_,                          &! component index
        nDim        =2,               &! dimensionality of the grid
        nRootBlock_D=(/2,1/),         &! block array for the 2 hemispheres
        nCell_D     =(/PM_n, PM_m/),  &! grid size for each block
        XyzMin_D    =(/0., 0./),      &! min colatitude and longitude indexes
        XyzMax_D    =(/cPi, 2*cPi/),  &! max colatitude and longitude indexes
        TypeCoord   ='SMG',           &! solar magnetic coordinate system
        iProc_A     =(/0,PM_nProc-1/)) ! processor assigment for the blocks

case default
```

```
     call CON_stop(NameSub//': invalid TypeAction='//TypeAction)
  end select

contains
  !============================================================================
  subroutine read_param
    ! Read parameter via ModReadParam
    use ModReadParam, ONLY: &
         read_line, read_command, read_var, i_line_read, lStringLine

    ! String for the name of the command
    character (len=lStringLine) :: NameCommand
    !---------------------------------------------------------------------
    do
       if(.not.read_line() ) EXIT
       if(.not.read_command(NameCommand)) CYCLE

       select case(NameCommand)
       case("#TIMESTEP")
          call read_var('Dt',PM_Dt)
       ! Read other component parameters
       case default
          if(PM_iProc==0) then
             write(*,'(a,i4,a)')NameSub//': ERROR at line ',i_line_read(),&
                  ' invalid command '//trim(NameCommand)
             if(UseStrict)call CON_stop('Correct PARAM.in!')
          end if
       end select
    end do

  end subroutine read_param
```

3.1.2 PM_init_session - initialize PM at the beginning of the session

INTERFACE:

```
subroutine PM_init_session(iSession, TimeSimulation)
```

USES:

```
use PM_ModIo, ONLY: PM_iUnit, PM_Prefix ! I/O unit and prefix for output

implicit none
```

INPUT ARGUMENTS:

```
integer,  intent(in) :: iSession       ! session number (starting from 1)
real,     intent(in) :: TimeSimulation ! seconds from start time
```

LOCAL VARIABLES:

```
character(len=*), parameter :: NameSub = 'PM_init_session'

logical :: DoInitialize = .true.
```

CONTENTS:

```
! Initialize module for the first time
if(DoInitialize)then
   call PM_setup
   DoInitialize = .false.
end if

! Initialization for all sessions comes here

! Here is an example of using PM_iUnit and PM_Prefix
write(PM_iUnit,*) PM_Prefix, NameSub, ' finished for session ', iSession
```

3.1.3 PM_run - do one time step with PM

INTERFACE:

```
subroutine PM_run(TimeSimulation, TimeSimulationLimit)
```

USES:

```
use PM_ModMain, ONLY: PM_Time ! Simulation time of PM

implicit none
```

INPUT/OUTPUT ARGUMENTS:

```
real, intent(inout) :: TimeSimulation   ! current time of component
```

INPUT ARGUMENTS:

```
real, intent(in) :: TimeSimulationLimit ! simulation time not to be exceeded
```

LOCAL VARIABLES:

```
character(len=*), parameter :: NameSub = 'PM_run'
real                        :: Dt                ! Time step
```

CONTENTS:

```
! Check if simulation times agree (e.g. for restart)
if(abs(PM_Time-TimeSimulation)>0.0001) then
   write(*,*)NameSub,' PM time=',PM_Time,' SWMF time=',TimeSimulation
   call CON_stop(NameSub//' ERROR: PM and SWMF simulation times differ')
end if

! Limit time step
Dt = min(PM_Dt, TimeSimulationLimit - TimeSimulation)

! Call the appropriate subroutine of the physics module to advance by Dt
call PM_advance(Dt)

! Return the new simulation time after the time step
TimeSimulation = PM_Time
```

3.1.4 PM_save_restart - save restart files for PM

INTERFACE:

```
subroutine PM_save_restart(TimeSimulation)
  implicit none
```

INPUT ARGUMENTS:

```
  real,     intent(in) :: TimeSimulation   ! seconds from start time
```

CONTENTS:

```
  ! Call the appropriate subroutine of the physics module
  call PM_save_restart_files(TimeSimulation)
```

3.1.5 PM_finalize - finalize PM at the end of the run

INTERFACE:

```
subroutine PM_finalize(TimeSimulation)

  implicit none
```

INPUT ARGUMENTS:

```
  real,     intent(in) :: TimeSimulation   ! seconds from start time
```

CONTENTS:

```
! Call the appropriate subroutines of the physics module
call PM_save_files_final(TimeSimulation)
call PM_error_report
```

3.2 Example Coupler between Components PM and XY

This subsection provides an example for the component coupling interfaces. The documentation is produced from the

`doc/Tex/PM_XY_coupler.f90`

file, which contains methods for coupling particular versions of the componets PM and XY. It can be used as a starting point for writing new couplers.

The coupling interfaces are not standardized, but they follow a typical pattern in the SWMF. The coupler between components PM and XY consists of three parts:

- The subroutines `PM_put_from_xy` and `PM_get_for_xy` in `PM/PmVersion/srcInterface`;
- The subroutines `XY_put_from_pm` and `XY_get_for_pm` in `XY/XyVersion/srcInterface`;
- The module `CON_couple_pm_xy` in `CON/Interface/src`.

If the coupling is one way only, for example if data is sent from the XY component to the PM component only, then the `PM_get_for_xy` and `XY_put_from_pm` subroutines do not need to be implemented.

The `get` and `put` subroutines are specific for the component versions. These subroutines provide access to the data in the physics modules. The `PM_get` and `PM_put` subroutines are executed on the processors that belong to the PM component, while the `XY_put` and `XY_get` subroutines are executed on the processors that belong to the XY component. In these subroutines only communication internal to the component can occur, if any. The `get` subroutines are responsible for converting from the internal units of the sending component to SI units, while the `put` subroutines convert from SI units to the internal units of the receiving component. If the coupler does not use the SWMF coupling toolkit, the interpolation from the sending grid to the requested positions occurs in the `get` or the `put` subroutines. If the SWMF coupling toolkit is used, the mapping and interpolation are done by the toolkit.

The `CON_couple_pm_xy` module should be specific to a pair of components, but it should be general with respect to the component versions. The methods in the `CON_couple_pm_xy` module implement both PM to XY and XY to PM coupling if the components are coupled both ways. These methods allocate and deallocate the data buffers and transfer the data between the PM and XY processors. If the coupler does not use the SWMF coupling toolkit, the data transfer is implemented with plain MPI calls. If the SWMF toolkit is used, the data transfer is done by the toolkit based on the grid descriptors and the mapping information.

The following example shows a one way coupling with plain MPI calls. For the use of the SWMF coupling toolkit, please see the appropriate part of the reference manual and the examples in the SWMF source code.

3.2.1 XY_get_for_pm - get XY data for sending it to PM

INTERFACE:

```
subroutine XY_get_for_pm(Buffer_II, iSize, jSize, NameVar)
```

USES:

```
use XY_ModField, ONLY: XY_calc_field, XY_Field_II, XY_FieldUnit

implicit none
```

INPUT ARGUMENTS:

```
integer, intent(in)           :: iSize,jSize            ! Size of buffer
character (len=*),intent(in)  :: NameVar                ! Name of data
```

OUTPUT ARGUMENTS:

```
real, intent(out)             :: Buffer_II(iSize,jSize) ! Data buffer
```

LOCAL VARIABLES:

```
character (len=*),parameter :: NameSub='XY_get_for_pm'
```

DESCRIPTION:

This subroutine shows an example for getting data for another component. This subroutine should be in the srcInterface directory of the specific XY component version. The subroutine is called by the couple_xy_pm method of the CON_couple_pm_xy module in CON/Interface/src.
Calculate field for the positions needed by the PM component. We assume here that the name of the variable, its dimension, and the grid descriptor of the PM component are sufficient for extracting the data.

CONTENTS:

```
! Allocate and set XY_Field_II for the positions needed by the PM component
call XY_calc_field(NameVar, iSize, jSize)

! Put appropriate part of the XY_Field_II array into the buffer
! Convert to SI units
select case(NameVar)
case('FieldNorth')
   Buffer_II = XY_FieldUnit * XY_Field_II(1:iSize,:)
case('FieldSouth')
   Buffer_II = XY_FieldUnit * XY_Field_II(iSize+1:2*iSize)
case default
   call CON_stop(NameSub//' invalid NameVar='//NameVar)
end select
```

3.2.2 PM_put_from_xy - put into PM the data received from XY

INTERFACE:

```
subroutine PM_put_from_xy(Buffer_II, iSize, jSize, NameVar)
```

USES:

```
use PM_ModProc, ONLY: PM_iProc, PM_nProc
use PM_ModMain, ONLY: PM_FieldNorth_II, PM_FieldSouth_II, PM_FieldUnit

implicit none
```

INPUT ARGUMENTS:

```
integer,          intent(in) :: iSize, jSize            ! Size of buffer
real,             intent(in) :: Buffer_II(iSize, jSize) ! Data buffer
character(len=*), intent(in) :: NameVar                 ! Name of data
```

LOCAL VARIABLES:

```
character (len=*), parameter :: NameSub = 'PM_put_from_xy'
```

DESCRIPTION:

This subroutine shows an example for receiving data from another component. This subroutine should be in the srcInterface directory of the specific PM component version. The subroutine is called by the couple_xy_pm method of the CON_couple_pm_xy module in CON/Interface/src.
The data is assumed to be on a spherical grid of size iSize by jSize. Based on the NameVar argument the data corresponds either to the northern or to the southern hemisphere. The northern hemisphere is always solved with processor 0, while the southern hemisphere with processor PM_nProc-1 where PM_nProc is the number of processors (1 or 2) used by component PM.

CONTENTS:

```
select case(NameVar)
case('FieldNorth')
   if (PM_iProc /= 0) RETURN
   PM_FieldNorth_II = Buffer_II / PM_FieldUnit
case('FieldSouth')
   if (PM_iProc /= PM_nProc-1) RETURN
   PM_FieldSouth_II = Buffer_II / PM_FieldUnit
case default
   call CON_stop(NameSub//' invalid NameVar='//NameVar)
end select
```

3.3 Module CON_couple_pm_xy - couple PM and XY components

This is an example for a module for the data exchange between two components. This module should be in the CON/Interface/src directory. The order of the component ID-s in the name of the module is alphabetical. This example provides methods for coupling the XY component to the PM component (one way only). A single field is passed on the spherical grid of the PM component. The communication assumes that the XY component collects all data on its root processor, while the grid of the PM component consists of two blocks (two hemispheres) and it runs on 1 or 2 processors.

INTERFACE:

```
module CON_couple_pm_xy
```

USES:

```
use CON_coupler ! provides all the CON methods needed for the coupling

implicit none

private ! except
```

PUBLIC MEMBER FUNCTIONS:

```
public :: couple_pm_xy_init   ! initialize coupling
public :: couple_xy_pm        ! couple XY to PM
```

LOCAL VARIABLES:

```
logical, save :: UseMe        ! logical for participating processors
integer, save :: iSize, jSize ! size of the 2D spherical structured PM grid
```

3.3.1 couple_pm_xy_init - initialize XY-PM couplings

INTERFACE:

```
subroutine couple_pm_xy_init
```

LOCAL VARIABLES:

```
  logical :: IsInitialized = .false.  ! logical to store initialization
  integer :: nCells_D(2)              ! temporary array for grid size
```

DESCRIPTION:

This subroutine initializes the data exchange between the XY and PM components. The subroutine must be called by the couple_all_init method of the CON_couple_all module. This particular initialization stores the PM grid size iSiza and jSize and sets the logical UseMe to .true. for participating PE-s.

CONTENTS:

```
if(IsInitialized) RETURN
IsInitialized = .true.

! Get the block size iSize and jSize from the PM grid descriptor
! using the ncell_id method of CON_coupler
nCells_D = ncell_id(PM_)
iSize = nCells_D(1); jSize = nCells_D(2)

! Set UseMe to .true. for the participating PE-s
! using the is_proc() function of CON_coupler
UseMe = is_proc(PM_) .or. is_proc(XY_)
```

3.3.2 couple_xy_pm - couple XY to PM

INTERFACE:

```
subroutine couple_xy_pm(tSimulation)
```

INPUT ARGUMENTS:

```
real, intent(in) :: tSimulation  ! simulation time
```

LOCAL VARIABLES:

```
character (len=*), parameter :: NameSub='couple_xy_pm'

character (len=*), parameter, dimension(2) :: &   ! names of variables
     NameVar_B = (/ 'FieldNorth', 'FieldSouth' /) ! for both blocks

real, dimension(:,:), allocatable :: Buffer_II    ! buffer for 2D field

integer :: nSize                      ! MPI message size
integer :: iStatus_I(MPI_STATUS_SIZE) ! MPI status variable
integer :: iError                     ! MPI error code

integer :: iBlock                     ! block index
integer :: iProcTo                    ! rank of the receiving processor
```

DESCRIPTION:

This subroutine is called by the couple_two_comp method of the CON_couple_all module. This particular coupler send data Field from XY to PM.

CONTENTS:

```
! Exclude PEs which are not involved
if(.not.UseMe) RETURN

! Do Northern and then Southern hemispheres
do iBlock = 1, 2
```

```
   ! Get the rank of the PM processor for this block
   ! using the pe_decomposition() method of CON_coupler
   iProcTo = pe_decomposition(PM_, iBlock)

   ! Allocate buffers for the variables both in XY and PM
   allocate(Buffer_II(iSize, jSize), stat=iError)
   call check_allocate(iError, NameSub//": "//NameVar_B(iBlock))

   ! Calculate Field on XY.
   ! The result will be on the root processor of XY.
   if( is_proc(XY_) ) &
        call XY_get_for_pm(Buffer_II, iSize, jSize, NameVar_B(iBlock))

   ! Transfer variables from the root processor of XY to PM.
   ! Identify the root processor of PM with the i_proc0() method
   ! and the receiving processor with the i_proc() method of CON_coupler.
   if( iProcTo /= i_proc0(XY_))then
      nSize = iSize*jSize
      if(is_proc0(XY_)) &
           call MPI_send(Buffer_II, nSize, MPI_REAL, iProcTo,&
           1, i_comm(), iError)
      if(i_proc() == iProcTo) &
           call MPI_recv(Buffer_II, nSize, MPI_REAL, i_proc0(XY_),&
           1,i_comm(), iStatus_I, iError)
   end if

   ! Put variables into PM
   if( i_proc() == iProcTo )&
        call PM_put_from_xy(Buffer_II, iSize, jSize, NameVar_B(iBlock))

   ! Deallocate buffer to save memory
   deallocate(Buffer_II)
end do
```

4 Documentation

New or modified input parameters should be described in the XML files. For the control module parameters this file is

```
PARAM.XML
```

For the component versions the XML file is located in the top level of the component version directory named like

```
PM/PmVersion/PARAM.XML
```

A brief description of the syntax of the PARAM.XML files can be obtained by running

```
Scripts/TestParam.pl -X
Scripts/TestParam.pl -H
```

The XML files are used for parameter checking with the

```
Scripts/TestParam.pl
```

script. They are also read with a text editor by the users of the SWMF. Finally the XML files are used for producing documentation with the

```
share/Scripts/XmlToTex
```

script. The PARAM.XML file should be listed in the appropriate part of the

```
doc/Tex/Makefile
```

5 Reporting Bugs

Bugzilla is the Center for Space Environment Modeling (CSEM) bug-tracking system and is used to submit and review defects that have been found in the Space Weather Modeling Framework (SWMF) and other CSEM projects. Bugzilla is not an avenue for technical assistance or support, but simply a bug tracking system. If you submit a defect, please provide detailed information in your submission after you have queried Bugzilla to ensure the defect has not been reported yet.

- Community members can use this system to file bugs and to perform searches on the bug database.
- Developers can use the system to obtain details of bugs assigned to them, and to prioritize bug-fixing activities.

The Bugzilla system is available here:

```
http://csem.engin.umich.edu/bugzilla/
```

6 Known Issues

```
Known issues with the release of the University of Michigan
Space Weather Modeling Framework (SWMF) and its constituent components.

SWMF core:
1. The SC component can now use the corotating HGR frame.
   The SC/IH coupling is tested, but the SC/SP and IH/SC
            couplings do not work for this case yet.

        NOTE: All couplings work if SC is in the HGI frame.

RB/RiceV5:
        1. The Radiation Belt (RB) module included with this release
           does not conform to several SWMF requirements for physics
           modules.  These include:
                A. The module does not write or read restart files.
                B. The time-step taken by the module cannot be
                   controlled (reduced) by the control module (CON).

        2. Plotting and analysis tools for examining RB output are not
           included with the distribution.

        3. The RB module is quite sensitive to complier options,
           especially optimization settings.

UA/GITM:
        1. Although the UA/GITM model can restart from a known
           steady-state, its initial output depends on its coupling
           with IE.  Therefore UA results, even on restart from a
           steady-state, tend to lack fidelity for the first several
           couplings with the IE module.  Results from UA should be
           ignored during this period.

NOTE: the UA/GITM2 model restarts correctly.

UA/GITM2:
1. The new version 2 of GITM has not been tested on all platforms.
   Succesful test runs were made with
            SGI Altix/Intel ifort8.070, Linux/NAG f95, and Linux/pgf90.

IM/RCM:
        1. Restarting the RCM (IM module) requires that if both restart
           files and plot files were written that both files must exist
           in order to restart.  This is due to the fact that when RCM
           writes plot and restart files, it appends them to files that
           are assumed to exist already via a record number.  This is
           not a bug, but the behavior does seem somewhat counter
           intuitive and may need modification.
NOTE: The plotting output is being completely redesigned
              to address this.

GM, IH, SC  / BATSRUS:
        1. Automatic Mesh Refinement (AMR) on restart does not hold the
```

```
           blocks touching the body to a constant refinement level
           when told to do so.  The feature works within a run
           correctly but fails to perform correctly after a restart.

FIXED: 09/24/2005 G. Toth

        2. The user_routine structure seems to require a reformulation
           in order to avoid incompatibilities of the different
           routines and flags.

SP/Kota:
        1. The SEP module included with this release does not have the
           ability to restart.

           NOTE: One can save the MHD data needed by this model into files
                 and the SP/Kota model can be (re)run continuously
                 by itself.

        2. The shock sensing algorithm used in the SEP module is not
           robust enough to find the correct shock structure in all
           situations.
```