PU/NE-98-13

SOFTWARE DESIGN AND IMPLEMENTATION DOCUMENT FOR THE PARCS-SPECIFIC DATA MAP ROUTINE IN THE COUPLED RELAP5/PARCS CODE

Douglas A. Barber, Thomas J. Downar

School of Nuclear Engineering
Purdue University
W. Lafayette, IN 47907-1290

submitted to:

Division of Systems Technology
Office of Regulatory Research
Nuclear Regulatory Commission
Washington, D.C. 20555-001

Contract NRC-04-97-039, Task 2
Performed Under Subcontract to SCIENTECH, Inc.

May 1998

I. Introduction

This document provides a description of the software design necessary to meet the Software Requirements Specification for the PARCS-Specific Data Map Routine (PDMR)[1]. The PARCS/PDMR code, which utilizes the FORTRAN-90 (F90) standard in the PDMR, is designed as a self-contained process which communicates with the General Interface process using standard message-passing constructs. Specifically, the Parallel Virtual Machine (PVM) package is utilized to handle all communication operations.

The PDMR functions as a secondary interface between the General Interface and PARCS. The software design description of Section II, provides detail concerning the nature of this implementation within PARCS, and outlines the key components of the PDMR. Specifically, the variable, functional, and error checking requirements are discussed and the necessary coding is demonstrated. In addition, detail of the process control and I/O is presented.

II. Software Design

The PDMR is designed to maintain consistency with the requirements of both the General Interface[2] and the RELAP5-Specific Data Map Routine (RDMR)[3], and the data to be transferred is specific to the RELAP5/PARCS coupling. In this implementation, entry into the functional units of the PDMR is controlled by PARCS through subroutine calls. In addition, the transfer of data between PARCS and the PDMR is achieved through the use of the PARCS header files.

The PDMR contains three functional units, one error checking unit, and one separate input processing unit. The first functional unit initializes the PVM process and establishes the communication with the General Interface process. The initialization unit also manages the communication of initial control information with the General Interface process, controls the input processing for the PDMR, and communicates the permutation matrices with the General Interface process. The second and third functional units handle both the transfer of time-dependent control information and the mapping of property data between the General Interface and PARCS processes. Finally, in the error checking unit, tests are performed to ensure that the information sent both to and from the General Interface is accurate.

For organization of the subroutines, advantage is take of the F90 `MODULE` construct. As will be shown in the following subsections, modules are utilized to group subroutines similar in function. Specifically, the subroutines associated with the initialization are contained within module `PDMR_Init_Calc`, the subroutines associated with the time-dependent mappings are contained within module `PDMR_Time_Calc`, the subroutines associated with the communication of control buffers between processes are contained within module `PDMR_Comm_Buf`, and the subroutines associated with the error checking are contained within module `PDMR_Error_Check`. In addition, a separate module, `Map_Input`, contains the subroutines necessary to perform the

input processing and construct the permutation matrices. A brief description of the subroutines and modules, along with a calling tree, is provided in Appendix B.

II.A. Variable Definition

The scalar and array variables used by the PDMR are grouped into eight separate classes, and a complete description of these variables, which includes data type, dimension, and range of values, is provided in Appendix A. The first class, shown in Table 1, relates to the variables needed for process control with PVM. The second class involves the thermal-hydraulic vector and control buffer data received from the General Interface (i.e. sent from RELAP5), and is described in Table 2. The third class, described in Table 3, involves the neutronic vector and control buffer data sent to the General Interface (i.e. sent to RELAP5). The vectors in the second and third classes, which contain the space-dependent property data, will utilize SI units and have the structure shown in the SRS for the PDMR.

Table 4 describes the variables associated with the two permutation matrices, both of which are sent to the General Interface from the PDMR. In an effort to minimize the amount of storage required for the matrices used by the General Interface, the Coordinate Storage Format[4] is utilized. This format requires three arrays corresponding to the non-zero elements in the matrices (`matvalth`, `matvaln`), the row identifier for each element (`matrowth`, `matrown`), and the column identifier for each element (`matcolth`, `matcoln`).

The fifth class of variables, shown in Table 5, relate to the parameters and dimension variables used for the vectors of property data and the permutation matrices. Table 6 describes some miscellaneous calculational control variables. And finally, Table 7 describes the variables associated with the error checking.

The variables shown in Tables 1-7 are declared in module `PDMR_Var_Decl`, which is used by each subroutine in the PDMR. The final class of variables, shown in Table 8, relate to the input processing and are declared locally in a separate module, `Map_Input`. For the declaration of all the variable arrays described in Tables 1-8, advantage is taken of the dynamic memory allocation of F90, and thus static dimension parameters are not required.

Header files are needed in the PDMR routines for both PVM and PARCS. For the utilization of PVM constructs, the header file 'fpvm3.h' is included. For access to PARCS memory locations the header files shown below are included. A description of the PARCS variables in these header files which are used by the PDMR is provided in Table 9.

param.h:	needed for all PARCS header files; contains dimension parameters
thparam.h:	needed for all thermal-hydraulic-related header files; contain dimension parameters for thermal-hydraulic variables
cntl.h:	ppm, rstrt, calcterm, errpdmr, errparcs, errextth
trcntl.h:	time, delt, deltd, scram, rsted, majed

geom.h:	nz, nxy, volcore
fbvar.h:	tdopl, tcool, dcool
thgeom.h:	pfa, hac
thopdata.h:	powfa, fracdc, fracdf
power.h:	plevel, relp

II.B. PARCS Process Control

The process flow for PARCS when coupled to the General Interface, and the entry points into the PDMR from PARCS, is shown in Figure 1. As mentioned, the PDMR contains three functional units, corresponding to initialization, thermal-hydraulic/heat structure to neutronic mapping, and neutronic to thermal-hydraulic/heat structure mapping, which is denoted in the figure by `PDMR(1)`, `PDMR(2)`, and `PDMR(3)`, respectively. Entry into these units is made from both `parcs.f` and `transient.f`, and is controlled using an integer route identifier (`iroute`) which has values of 1, 2, or 3, and is passed as a subroutine argument to the PDMR. As indicated in Figure 1, `parcs.f` controls entry into each of the three units and `transient.f` controls entry into only the two mapping units. The following lines of code depict the routing logic used in subroutine `PDMR()`. The code section shown below also demonstrates part of the exit handling procedure which will be discussed in more detail in Section II.G.

```
Check_Route: SELECT CASE (iroute)
CASE(1)
  CALL PDMR_Init()
  IF (done) THEN
    CALL PDMR_Clean()
    CALL PDMR_Exit()
  ENDIF
CASE(2)
  CALL PDMR_th2n()
  IF (errgi .OR. errth) THEN
    CALL PDMR_Clean()
    CALL PDMR_Exit()
  ENDIF
CASE(3)
  CALL PDMR_n2th()
  IF (done) THEN
    CALL PDMR_Clean()
    CALL PDMR_Exit()
  ENDIF
END SELECT Check_Route
```

The process flow through the three functional units is depicted in Figure 2, and Sections II.C through II.E will discuss in more detail the calculational procedure for each. In addition, a brief description of the subroutines used by the PDMR, along with a calling tree, is provided in Appendix B.

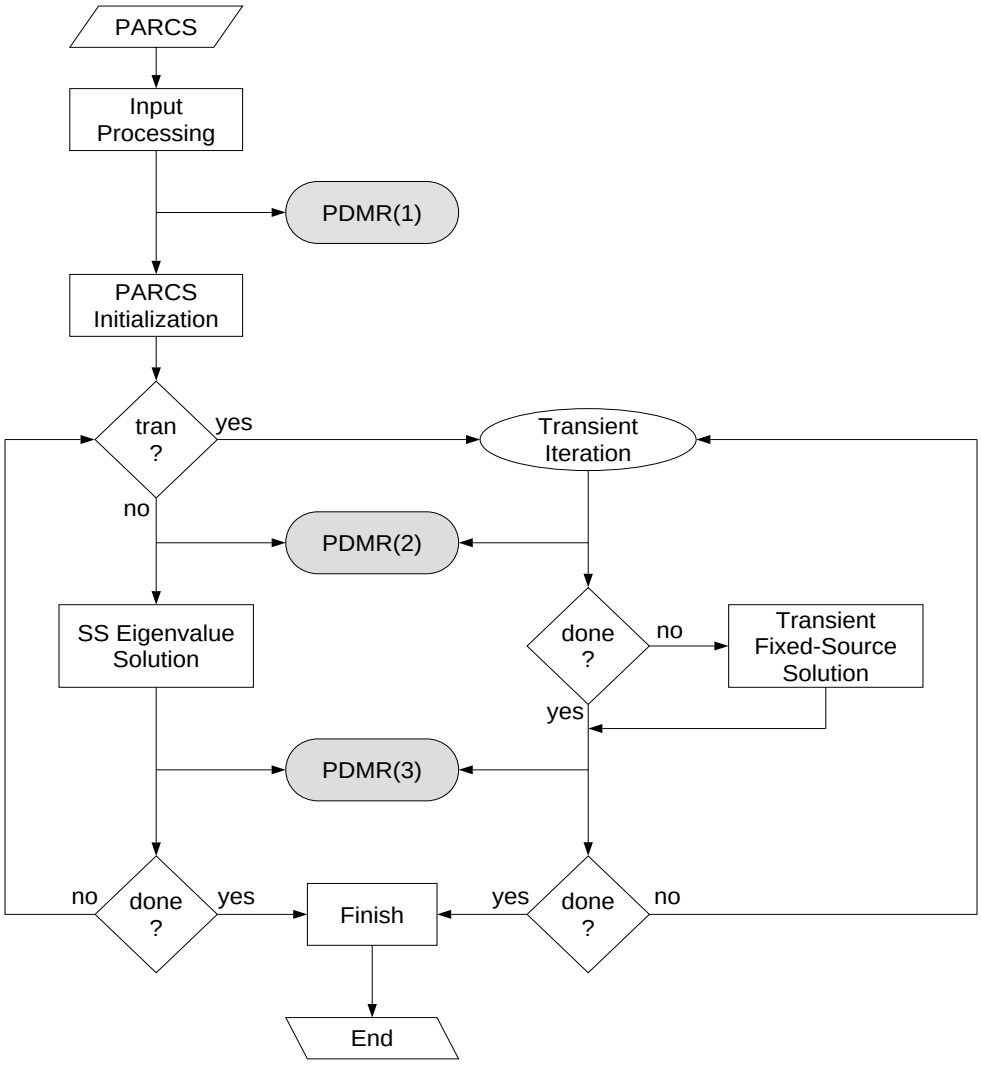


Figure 1: Flow Control Through PARCS/PDMR

The first step in the PARCS procedure is to process the user input. An additional input option which indicates whether PARCS should be run in "stand-alone" mode or with an external T/H code is included and processed in subroutine `inputd.f`. If the user turns the external T/H option on, which sets the PARCS logical `extth=.true.`, the next step following the input processing is to call the initialization unit of the PDMR. Upon return from the PDMR, PARCS will perform its initialization procedure. It should be noted that even though PARCS T/H will not be utilized, the T/H initialization procedure will still be performed. The feedback variables received from RELAP5 (temperatures, densities, etc..), will overwrite this initialized data.

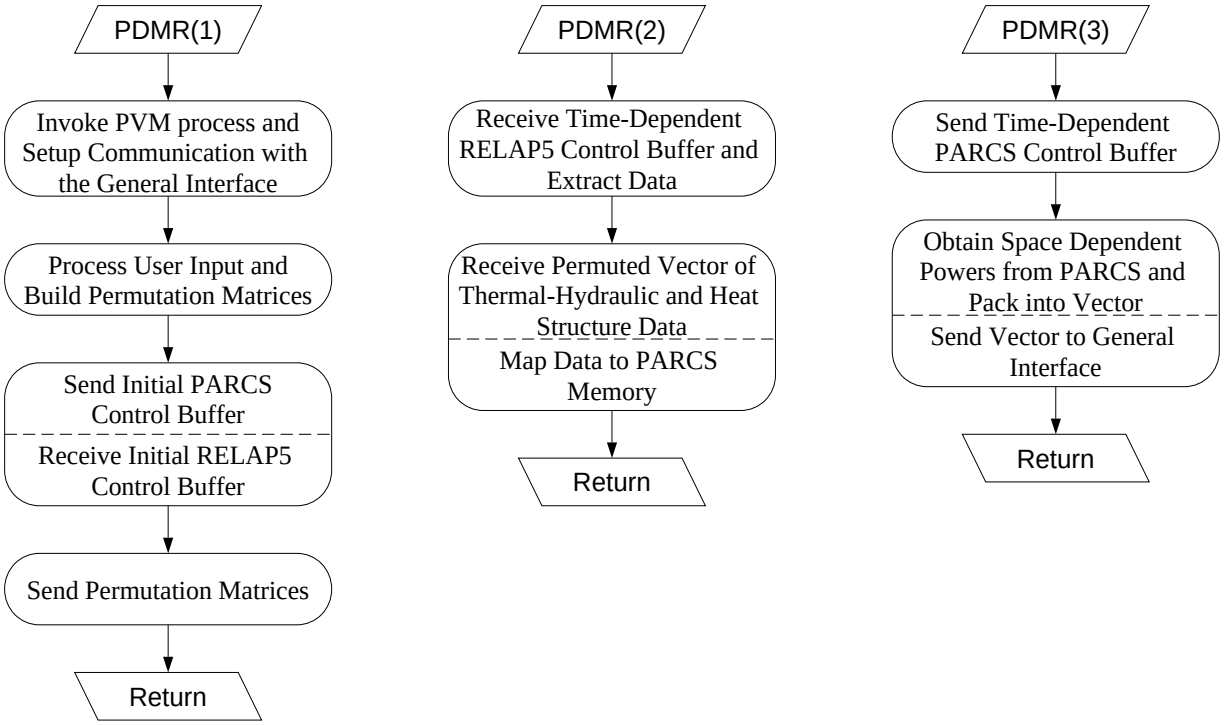


Figure 2: Flow Through the Three Functional Units of the PDMR

Once the PARCS/PDMR initialization process is complete, the second unit of the PDMR is called to obtain the initial condition data from RELAP5. Upon return to PARCS, the property data received from the General Interface is used to incorporate feedback into the cross sections. Once the updated cross sections have been obtained, the steady-state eigenvalue problem (EVP) is initialized and then solved (keeping total core power constant) to obtain the eigenvalue and the initial core power distribution. The third unit of the PDMR is then called to convert these space-dependent powers to units of watts and send them to the General Interface to be mapped to the thermal-hydraulic and heat structure problem domains.

With the completion of the steady-state EVP, the PARCS subroutine `transient.f` is called to drive the time-dependent calculation. Once the transient problem has been initialized, the PARCS T/H calculation is bypassed, and a call to the second unit of the PDMR is made to obtain the time-dependent thermal-hydraulic and heat structure data, along with other control information. Upon return from the PDMR, the cross sections are updated using the data sent from RELAP5, the transient fixed-source problem (TFSP) is initialized, and the time/space-dependent neutronic powers are obtained from solution of the TFSP. The third unit of the PDMR is then called to convert these powers to units of watts and send them to the General Interface. This procedure is then repeated until either RELAP5 indicates the calculation is complete or an error signal has been received. A more detailed description of the exit handling procedure is given in Section II.G.

It should be noted that for the current coupling of RELAP5/PARCS, only “new” problems can be performed. The capability to begin the calculation from a restart will require some modification to both RELAP5 and PARCS. Thus, the flow through PARCS will involve only one SS EVP to obtain the initial power distribution, followed by the time-dependent calculation. In addition, the steady-state initialization option, in which RELAP5 executes in steady-state mode and PARCS only performs and EVP at each “pseudo” time step, is not currently available.

II.C. Initialization

The initialization unit controls four basic tasks: (1) communicate process IDs with the General Interface, (2) construct the permutation matrices from user input, (3) communicate initial control buffers with the General Interface, and (4) send the permutation matrices to the General Interface. Tasks (1), (3), and (4) are performed by subroutines contained in the module `PDMR_Init_Calc`, and Task (2) is performed by subroutines contained in a separate module, `MAP_Input`.

Task (1) is performed by subroutine `PDMR_Obtain_IDs()`, and involves first enrolling the PARCS process in PVM by calling the function `pvmfmytid()` to obtain the PARCS/PVM process ID. The PARCS process will then join a dynamic process group consisting of the RELAP5 and General Interface processes, and wait on a barrier for these processes. Once all of the processes have arrived at the barrier, the PDMR will receive the process ID broadcasted from the General Interface, and send the PARCS process ID to the General Interface. This procedure provides communication coherency between the General Interface and PARCS processes, and is performed using pre-determined message tags (`mtypegi`, `mtypen`), which are consistent with those used by the General Interface[5]. The following lines of code demonstrate the necessary procedure:

```
c Establish communication with the General Interface process.
      ntasks = 3
      mtypegi = 1
      mtypen = 3
      group = 'procs'
c Enroll the PARCS process in PVM.
      CALL pvmfmytid( tidn)
c Join the dynamic process group and wait for the General Interface
c and thermal-hydraulic processes.
      CALL pvmfjoingroup( group, inum)
      CALL pvmfbarrier( group, ntasks, info)
c Receive ID broadcasted from the General Interface process.
      CALL pvmftrecv( -1, mtypegi, timeout, 0, info)
      CALL pvmfunpack( INTEGER4, tidgi, 1, 1, info)
c Send ID to the General Interface process.
      CALL pvmfinitsend( PVMDEFAULT, info)
      CALL pvmfpack( INTEGER4, tidn, 1, 1, info)
      CALL pvmfsend( tidgi, mtypen, info)
```

It should be noted that the function, `pvmftrecv`, which was shown above and is used throughout the code, is a non-blocking receive which terminates if no message has arrived after "`timeout`" seconds. `timeout`, which is described in Table 1 of Appendix A, is set to 600 seconds.

The next task in the initialization unit involves processing the permutation matrices to be sent to the General Interface. A description of the subroutines necessary to perform this task is provided separately in Section II.H.

Task (3) is controlled by subroutine `PDMR_Buf_Init()`. The first step of task (3) is to construct and send the initial PARCS control buffers to the General Interface process, and subsequently, the RELAP5 process. The data to be sent in the control buffers was described in the SRS for the PDMR, and the procedure required to construct the buffers is depicted in the following lines of code. The variables copied into the control buffers are described in the tables of Appendix A, and are consistent with the data being extracted by the RELAP5-Specific Data Map Routine.

```
c Allocate memory for the control buffers.
c  (memory deallocated in PDMR_Send_Bufn() )
      dimbuf = (/ 0, 7, 0, 5+nvol+nhs, 0, 0/)
      ALLOCATE( cbufn(dimbuf(1)))
      ALLOCATE( lbufn(dimbuf(2)))
      ALLOCATE( i2bufn(dimbuf(3)))
      ALLOCATE( i4bufn(dimbuf(4)))
      ALLOCATE( r4bufn(dimbuf(5)))
      ALLOCATE( r8bufn(dimbuf(6)))
c PARCS is sending the permutation matrices.
      recvn = .TRUE.
c Pack the PARCS control buffers.
      lbufn(1) = errcalcn
      lbufn(2) = errdatan
      lbufn(3) = errpvmn
      lbufn(4) = errdatagi
      lbufn(5) = errpvmgi
      lbufn(6) = done
      lbufn(7) = recvn
      i4bufn(1) = itrip
      i4bufn(2) = nvol
      i4bufn(3) = nhs
      i4bufn(4) = ncolth2n
      i4bufn(5) = nrown2th
      i4bufn(6:nvol+5) = ivol(:)
      i4bufn(6+nvol:nvol+nhs+5) = ihs(:)
```

Once the data has been copied, subroutine `PDMR_Send_Bufn()` is called to pack these control buffers, along with the dimension of each buffer, into a single PVM data structure and then send the structure to the General Interface process. The procedure is performed as follows:

```
      istride = 1
      CALL pvmfinitsend( PVMDEFAULT, info)
```

```
c Pack buffer dimensions.
      CALL pvmfpack (INTEGER4, dimbuf, 6, istride, info)
c Pack data type-dependent control buffers.
      CALL pvmfpack( STRING, cbufn, 6*dimbuf(1), istride, info)
      CALL pvmfpack( INTEGER4, lbufn, dimbuf(2), istride, info)
      CALL pvmfpack( INTEGER2, i2bufn, dimbuf(3), istride, info)
      CALL pvmfpack( INTEGER4, i4bufn, dimbuf(4), istride, info)
      CALL pvmfpack( REAL4, r4bufn, dimbuf(5), istride, info)
      CALL pvmfpack( REAL8, r8bufn, dimbuf(6), istride, info)
c Send data structure to General Interface process.
      CALL pvmfsend( tidgi, mtypen, info)
```

Once the structure has been sent, the memory for the buffers can be deallocated as shown below:

```
c Deallocate memory for the PARCS control buffers.
      DEALLOCATE( cbufn)
      DEALLOCATE( lbufn)
      DEALLOCATE( i2bufn)
      DEALLOCATE( i4bufn)
      DEALLOCATE( r4bufn)
      DEALLOCATE( r8bufn)
```

The second step of Task (3) involves receiving the RELAP5 initial control buffers, and extracting the necessary information. The buffers, along with the buffer dimensions, are sent from the General Interface as one structure. This data structure is received and unpacked in subroutine `PDMR_Recv_Bufth()`, and the necessary coding is shown below:

```
c Receive data structure from the General Interface process.
      CALL pvmftrecv( tidgi, mtypegi, timeout, 0, info)
      istride = 1
c Extract buffer dimensions and allocate memory for the
c control buffers.
      CALL pvmfunpack( INTEGER4, dimbuf, 6, istride, info)
      ALLOCATE( cbufth(dimbuf(1)))
      ALLOCATE( lbufth(dimbuf(2)))
      ALLOCATE( i2bufth(dimbuf(3)))
      ALLOCATE( i4bufth(dimbuf(4)))
      ALLOCATE( r4bufth(dimbuf(5)))
      ALLOCATE( r8bufth(dimbuf(6)))
c Extract data type-dependent control buffers.
      CALL pvmfunpack( STRING, cbufth, 6*dimbuf(1), istride, info)
      CALL pvmfunpack( INTEGER4, lbufth, dimbuf(2), istride, info)
      CALL pvmfunpack( INTEGER2, i2bufth, dimbuf(3), istride, info)
      CALL pvmfunpack( INTEGER4, i4bufth, dimbuf(4), istride, info)
      CALL pvmfunpack( REAL4, r4bufth, dimbuf(5), istride, info)
      CALL pvmfunpack( REAL8, r8bufth, dimbuf(6), istride, info)
```

Upon return from subroutine `PDMR_Recv_Bufth()`, the necessary data is extracted from the buffers, and the memory for the buffers is deallocated, as shown below. This data was described in the SRS for the PDMR, and the variables used by the PDMR to store the data are

described in the tables of Appendix A. In addition, the following PARCS header files are required to be included for variables `rstrt`, `calcterm`, and `time` (a description of these variables was given in Table 9 of Appendix A): `param.h`, `cntl.h`, and `trcntl.h`.

```
c Extract data from RELAP5 control buffers.
      allocate(hsdscr(dimbuf(1)))
      hsdscr(1:dimbuf(1)) = cbufth(1:dimbuf(1))
      errcalcth = lbufth(1)
      errdatath = lbufth(2)
      errpvmth = lbufth(3)
      errdatagi = lbufth(4)
      errpvmgi = lbufth(5)
      done = lbufth(6)
      rstrt = lbufth(8)
      time = r8bufth(1)
      calcterm = done
c Deallocate memory for the control buffers.
c  (memory allocated in PDMR_Recv_Bufth() )
      DEALLOCATE( cbufth)
      DEALLOCATE( lbufth)
      DEALLOCATE( i2bufth)
      DEALLOCATE( i4bufth)
      DEALLOCATE( r4bufth)
      DEALLOCATE( r8bufth)
```

It should be noted, that in the current implementation `rstrt` will always be false, and `time` will always be zero. These two variables will be utilized in future implementations to treat transient calculations beginning from a restart. In such an implementation, RELAP5 would indicate whether a restart calculation was being performed (`rstrt=.true.`), and at what time point the calculation was beginning (`time`).

The final step of Task (3) determines the method to be used for calculating Doppler temperature. Initially, the user provides input indicating that the Doppler temperature should be calculated based on either the average fuel temperature (`idopl=0`) or a linear combination of surface and centerline fuel temperatures (`idopl=1`), as described in Section II.H. However, if the user chooses to utilize the linear combination approach, it is necessary to verify that the core heat structures used in RELAP5 correspond to a cylindrical geometry. Thus, RELAP5 passes heat structure geometry descriptors, which correspond to either “PIN” or “SLAB”, in the character control buffer, `cbufth`, to allow the PDMR to perform this verification. Specifically, if `idopl=1`, but any one of the core heat structures does not correspond to “PIN”, then the PDMR will default to calculating the Doppler temperature based on the average fuel temperature and display a message to the user. The logic used to perform this test is shown below:

```
c Test on geometry of HS;  if any descriptors indicate geometry
c other than 'PIN', then use average fuel temperature.
      IF (idopl .EQ. 1) then
        DO 10 i=1,size(hsdscr)
          IF (hsdscr(i) .NE. 'PIN') idopl=0
  10    CONTINUE
        IF (idopl .EQ. 0) THEN
```

```
          WRITE(ioutp,*)
          WRITE(ioutp,*)"*** Using Average Fuel Temperature for "
     &,      "Doppler Temperature ***"
          WRITE(ioutp,*)
        ENDIF
      ENDIF
```

The final task of the initialization unit involves sending the permutation matrices to the General Interface, which is accomplished with a call to subroutine `PDMR_Send_Mat()`. This subroutine packs the two permutation matrices, along with the total dimensions of the matrices, into a single structure. This structure is then sent to the General Interface, and the memory for the permutation matrices is deallocated. The following lines of code depict this procedure:

```
      istride = 1
      CALL pvmfinitsend( PVMDEFAULT, info)
c Pack buffer dimensions for the permutation matrices.
      CALL pvmfpack( INTEGER4, nmatth, 1, istride, info)
      CALL pvmfpack( INTEGER4, nmatn, 1, istride, info)
c Pack thermal-hydraulic/heat structure to neutronic matrix.
      CALL pvmfpack( REAL8, matvalth, nmatth, istride, info)
      CALL pvmfpack( INTEGER4, matrowth, nmatth, istride, info)
      CALL pvmfpack( INTEGER4, matcolth, nmatth, istride, info)
c Pack neutronic to thermal-hydraulic/heat structure matrix.
      CALL pvmfpack( REAL8, matvaln, nmatn, istride, info)
      CALL pvmfpack( INTEGER4, matrown, nmatn, istride, info)
      CALL pvmfpack( INTEGER4, matcoln, nmatn, istride, info)
c Send the structure to the General Interface process.
      CALL pvmfsend( tidgi, mtypen, info)
c Deallocate memory for the permutation matrices.
c  (memory allocated in Build_Mat() )
      DEALLOCATE( matvalth)
      DEALLOCATE( matrowth)
      DEALLOCATE( matcolth)
      DEALLOCATE( matvaln)
      DEALLOCATE( matrown)
      DEALLOCATE( matcoln)
```

II.D. General Interface to PARCS Mapping

This functional unit performs two basic tasks: (1) receive RELAP5 time-dependent control buffers, and (2) receive and store permuted vector of thermal-hydraulic and heat structure data. The two subroutines utilized for these tasks, `PDMR_Buf_th2n()` and `PDMR_Map_th2n()` respectively, are contained in module `PDMR_Time_Calc`. The procedure of `PDMR_Buf_th2n()` is similar to that used in `PDMR_Buf_Init()` for the receipt of the RELAP5 initial control buffers. The only difference here is seen in the data extracted from the buffers, as shown below. This data is described in the SRS for the PDMR, and a description of the variables is given in the tables of Appendix A. In addition, the following PARCS header files are required to be included for variables `rsted`, `majed`, `scram`, `deltd`, `delt`, and `calcterm`: `param.h`, `cntl.h` and `trcntl.h`.

```
c Extract data from RELAP5 control buffers.
      errcalcth = lbufth(1)
      errdatath = lbufth(2)
      errpvmth = lbufth(3)
      errdatagi = lbufth(4)
      errpvmgi = lbufth(5)
      done = lbufth(6)
      rsted = lbufth(7)
      majed = lbufth(8)
      scram = lbufth(9)
      deltd = delt
      delt = r8bufth(1)
      calcterm = done
```

In the current implementation of RELAP5/PARCS, `rsted` and `majed` will both be false. These flags are available for future implementations in which RELAP5 will dictate to PARCS when restart and major edits should be performed. The scram variable is available to allow RELAP5 to communicate to PARCS if the control rods should be scrammed. A value of true results in PARCS starting the scram procedure at the current time step based on PARCS user input for items such as insertion speed and stuck rod location(s). All other "planned" rod movement is controlled by PARCS user input.

The first step of Task (2) is to receive the data structure sent from the General Interface, which is composed of the permuted vector of thermal-hydraulic and heat structure data and the vector dimension. Once received, the vector dimension is unpacked first and used to allocate space for the permuted vector, which is only allocated once at the beginning of the time-dependent calculation. The following lines of code demonstrate this procedure:

```
      CALL pvmftrecv( tidgi, mtypegi, timeout, 0, info)
      istride = 1
c Unpack vector buffer dimension and allocate memory for the vector.
      CALL pvmfunpack( INTEGER4, nvecthp, 1, istride, info)
      IF (.NOT.allocated(vecthp)) ALLOCATE( vecthp(nvecthp))
c Unpack permuted vector of thermal-hydraulic/heat structure data.
      CALL pvmfunpack( REAL8, vecthp, nvecthp, istride, info)
```

The next step is to map this data to the appropriate memory locations used by PARCS. The variables used by PARCS are declared in header files, and the following are required to be included in subroutine `PDMR_Map_th2n()`: `param.h`, `thparam.h`, `cntl.h`, `geom.h`, and `fbvar.h`. As described in the SRS for the PDMR, the data stored in the vector is (in order): moderator temperature (K), moderator liquid density (kg/m^3), moderator vapor density (kg/m^3), moderator vapor void fraction, Boron concentration (ppm), average fuel temperature (K), centerline fuel temperature (K), and surface fuel temperature (K). The first data to be extracted is the moderator temperature, which is stored in PARCS variable `tcool` with units of Celsius, as shown below.

```
c Moderator Temperature:
      m=0
      DO 20 k=1,nz
        DO 10 l=1,nxy
          m=m+1
          tcool(k,l)=vecthp(m)-273.15
  10    CONTINUE
  20  CONTINUE
```

`nz` is the number of axial planes and `nxy` is the number of radial nodes. The next data to be extracted relates to the moderator density, which is stored in PARCS variable `dcool` with units of kg/m^3:

```
c Moderator Liquid Density:
      m=nvolnode
      DO 40 k=1,nz
        DO 30 l=1,nxy
          m=m+1
          dcool(k,l)=vecthp(m)
  30    CONTINUE
  40  CONTINUE
```

It should be noted that in the current version of PARCS (Ver. 2.1), only the moderator liquid density is used. Thus, the moderator vapor density and void fraction is not extracted and stored in PARCS at this time. However, for future versions of PARCS which may require this data, the following lines of code are included (and commented out) in the source.

```
c      Moderator Vapor Density:
c           m=2*nvolnode
c           DO 60 k=1,nz
c             DO 50 l=1,nxy
c               m=m+1
c               dcoolv(k,l)=vecthp(m)
c       50    CONTINUE
c       60  CONTINUE
c
c      Moderator Void Fraction:
c           m=3*nvolnode
c           DO 80 k=1,nz
c             DO 70 l=1,nxy
c               m=m+1
c               voidf(k,l)=vecthp(m)
c       70    CONTINUE
c       80  CONTINUE
```

The Boron concentration is extracted next, and for Ver. 2.1 of PARCS, the Boron is stored as a single value for the entire core (`ppm`). Thus, the following lines of code depict the procedure necessary to obtain the core-averaged Boron concentration. For future versions of PARCS which may use a space-dependent Boron concentration, this procedure will need to be modified.

```
c Boron Concentration:
      m=4*nvolnode
      ppm = 0.0
      vcore = 0.0
      DO 100 k=1,nz
        DO 90 l=1,nxy
          m=m+1
          ppm = ppm + vecthp(m)*volnode(l,k)
          vcore = vcore + volnode(l,k)
  90    CONTINUE
 100  CONTINUE
      ppm = ppm/vcore
```

Finally, the square-root of Doppler temperature is calculated and stored in PARCS variable `tdopl` with units of $\sqrt{K}$. The method for calculating Doppler temperature is determined based on user input and data from RELAP5 during the initialization, as described in Section II.C. The following demonstrates the logic necessary:

```
c Doppler Temperature:
c
c Doppler based on average fuel temperature.
      IF (idopl .EQ. 0) THEN
        m=5*nvolnode
        DO 120 k=1,nz
          DO 110 l=1,nxy
            m=m+1
            tdopl(k,l)=sqrt(vecthp(m))
 110      CONTINUE
 120    CONTINUE
c Doppler based on linear combination of centerline and surface
c fuel temperature.
      ELSEIF (idopl .EQ. 1) THEN
        m=5*nvolnode + nhsnode
        n=5*nvolnode + 2*nhsnode
        DO 140 k=1,nz
          DO 130 l=1,nxy
            m=m+1
            n=n+1
            tdopl(k,l)=sqrt((1-wdopl)*vecthp(m)+wdopl*vecthp(n))
 130      CONTINUE
 140    CONTINUE
      ENDIF
```

where `wdopl` is obtained from user input in subroutine `Read_Map()`.

II.E. PARCS to General Interface Mapping

This functional unit performs two basic tasks: (1) process the PARCS time-dependent control buffers, and (2) send the buffers and the unpermuted vector of neutronic data. The two subroutines utilized for these tasks, `PDMR_Buf_n2th()` and `PDMR_Map_n2th()` respectively, are contained in module `PDMR_Time_Calc`. The procedure of `PDMR_Buf_n2th()` is similar to that used in `PDMR_Buf_Init()` for the sending of the PARCS initial control buffers. The difference here is seen in the data placed in the buffers, as shown below. In addition, these buffers are not sent immediately, but rather, are sent prior to the sending of the vector of neutronic data in `PDMR_Map_n2th()`. The data in the buffers is described in the SRS for the PDMR, and a description of the variables is given in the tables of Appendix A

```
c Allocate memory for the control buffers.
c  (memory deallocated in PDMR_Send_Bufn() )
      dimbuf = (/ 0, 6, 0, 0, 0, 0/)
      ALLOCATE( cbufn(dimbuf(1)))
      ALLOCATE( lbufn(dimbuf(2)))
      ALLOCATE( i2bufn(dimbuf(3)))
      ALLOCATE( i4bufn(dimbuf(4)))
      ALLOCATE( r4bufn(dimbuf(5)))
      ALLOCATE( r8bufn(dimbuf(6)))
c Pack the PARCS control buffers.
      lbufn(1) = errcalcn
      lbufn(4) = errdatagi
      lbufn(5) = errpvmgi
      lbufn(6) = done
```

The second and third words of the logical control buffer will be assigned in `PDMR_Map_n2th()` following the data error checking and prior to the sending of the control buffers.

The first step of Task (2) is to allocate space for the vector of neutronic data to be sent to the General Interface, the coding for which is shown below:

```
c Copy data from PARCS memory to vector buffer.
      nvecn = nsubn2th*nvolnode + nsubn2hs*nhsnode
      IF (.NOT.allocated(vecn)) ALLOCATE( vecn(nvecn))
```

As described in the SRS for the PDMR, the neutronic data to be sent to RELAP5 involves total power deposited in the coolant, and total power deposited in the fuel. PARCS stores the time-dependent total core power as normalized (`plevel`). Thus, in order to obtain the total core power in watts (`corepow`), it is first necessary to obtain the nominal core power, and then multiply by the current power level. Since PARCS stores nominal core power in megawatts on a per-assembly basis (`powfa`), the total nominal core power can be obtained by multiplying `powfa` by the effective number of fuel assemblies and converting to watts by multiplying by 1.0e-6. It should be noted that the effective number of assemblies is needed for quarter- and half-core symmetric models, in which quarter- and half-assemblies may reside on the line of symmetry. This value can easily be obtained by dividing the total active core volume (`volcore` = sum of the volume of calculation nodes corresponding to the fuel) by the active volume of a fuel assembly.

Once the total core power has been obtained, the total power to the coolant and the fuel is determined by multiplying by the fractions obtained through PARCS user input (`fracdc` and `fracdf`). The following lines of code depict the necessary procedure:

```
c Calculate total power to coolant and fuel.
      corepow = plevel*powfa*volcore*1.0e-6/(hac*pfa*pfa)
      tpowc = fracdc*corepow
      tpowf = fracdf*corepow
```

The space-dependent powers stored in PARCS (`relp`) are relative to the total core power. Thus, the node-wise power (in watts) deposited in the coolant and fuel can easily be calculated by multiplying `relp` by `tpowc` and `tpowf`, respectively. The space-dependent coolant powers, which will be mapped to thermal-hydraulic volumes, are packed into the unpermuted vector first. These are then followed by the space-dependent fuel powers, which will be mapped to heat structures. The following lines of code demonstrate the packing procedure:

```
c Pack neutronic vector.
      m=0
      DO 20 k=1,nz
        DO 10 l=1,nxy
          m=m+1
          vecn(m)=tpowc*relp(k,l)
  10    CONTINUE
  20  CONTINUE
      DO 40 k=1,nz
        DO 30 l=1,nxy
          m=m+1
          vecn(m)=tpowf*relp(k,l)
  30    CONTINUE
  40  CONTINUE
```

The PARCS variables described above are declared in the following header files, which are required to be include in subroutine `PDMR_Map_n2th()`: `param.h`, `thparam.h`, `geom.h`, `thopdata.h`, and `power.h`.

At this point, all data error checking in this unit should be complete and the PDMR error logicals should be current. These logicals are packed into the second and third words of the logical control buffer and sent to the General Interface, as shown below:

```
      lbufn(2) = errdatan
      lbufn(3) = errpvmn
      CALL PDMR_Send_Bufn()
```

Once the control buffers have been sent and the vector of neutronic powers has been constructed, this vector is packed into a PVM data structure, along with the vector dimension, and sent to the General Interface, as shown below:

```
      istride = 1
      CALL pvmfinitsend( PVMDEFAULT, info)
```

```
c Pack vector buffer dimension.
      CALL pvmfpack( INTEGER4, nvecn, 1, istride, info)
c Pack unpermuted vector of neutronic data.
      CALL pvmfpack( REAL8, vecn, nvecn, istride, info)
      CALL pvmfsend( tidgi, mtypen, info)
```

II.F. Error Checking

The error checking module, `PDMR_Error_Check`, contains three subroutines, `PDMR_Data_Errchk()`, `PDMR_PVM_Errchk()`, and `PDMR_Proc_Errchk()`, which are called from each of the previously described functional units. The error checking operations performed by `PDMR_Data_Errchk()` are specific to each unit, and the call to this subroutine includes one argument, `errcode`, which is an integer relating to the specific error check to be performed (shown in parenthesis below). If an error is detected in this subroutine, the logical `errdatan` (see Table 7 of Appendix A) is set to .TRUE., and the appropriate error message is displayed. In addition, the PDMR then communicates a "data" error, which indicates that the calculation should be terminated, to the General Interface process, and consequently the RELAP5 process, using the space available in the neutronic logical buffer.

<u>Initialization</u>:

(1) **[fatal error]** *A neutronic node specified by the user is not available in PARCS:*
result from: `1` > `ivolnode(:)` > `nz*nxy`
(or) `1` > `ihsnode(:)` > `nz*nxy`
where `nz` = number of planes (including top and bottom reflector)
and `nxy` = number of radial nodes (including radial reflector)

(2) **[fatal error]** *A neutronic node exists in PARCS which was not assigned a volume or heat structure number in the mapping input:*
result from: any one of the numbers in the range `1` to `nz*nxy` does not exist in array `ivolnode(:)`
(or) any one of the numbers in the range `1` to `nz*nxy` does not exist in array `ihsnode(:)`

(3) **[fatal error]** *Matrix elements outside prescribed range*:
result from: elements of `matvalth` or `matvaln` are outside the range specified in Table 4 of Appendix A.

(4) **[fatal error]** *Weighting factors in permutation matrix are inaccurate*:
result from: sum of row elements in thermal-hydraulic/heat structure to neutronic matrix do not sum to 1.0
(or) sum of column elements in neutronic to thermal-hydraulic/heat structure matrix do not sum to 1.0

Thermal-Hydraulics to Neutronics Mapping:

(5) **[fatal error]** *Inconsistency between matrix and vector dimensions*:
result from: `nvecthp` ≠ `MAXVAL(matrowth)`
(or) `nvecthp` ≠ `SIZE(vecthp)` (This check is required because the memory for `vecthp` is only allocated once at the beginning of the calculation. Thus, if the dimension, `nvecthp`, is not consistent with the size of the previously allocated vector, the unpacking of the vector will be in error.)

(6) **[fatal error]** *Property data extracted from permuted vector is outside specified bounds*:
result from: `0.0 K > Mod. Temp. > 10000.0 K`
(or) `0.0 kg/m^3 > Mod. Liquid Density > 5000.0 kg/m^3`
(or) `0.0 kg/m^3 > Mod. Vapor Density > 5000.0 kg/m^3`
(or) `0.0 > Void Fraction > 1.0`
(or) `0.0 ppm > Boron Concentration > 1.0E+06 ppm`
(or) `0.0 K > Fuel Temp. > 10000.0 K`

Neutronics to Thermal-Hydraulics Mapping:

(7) **[fatal error]** *Inconsistency between matrix and vector dimensions*:
result from: `nvecn` ≠ `MAXVAL(matcoln)`

(8) **[fatal error]** *Neutronic powers in unpermuted vector are outside specified bounds*:
result from: `0.0 watts > Power > 1.0E+15 watts`

It should be noted that the error bounds showed in “data” errors 6 and 8 are not based on physical property constraints; it is expected that RELAP5 will fail on property failures. Rather, these bounds are simply utilized to trap errors which may have occurred either in the General Interface mapping procedure or during communication.

`PDMR_PVM_Errchk()` detects errors in the PVM calls, and includes three arguments, `errcode`, `iunit`, and `istat`. `errcode` refers to the PVM function which returned an error, `iunit` refers to where `PDMR_PVM_Errchk()` is being called from, and `istat` is the integer PVM status code which indicates an error if its value is less than zero. An association table is shown below for `errcode` and `iunit`. If an error is detected in this subroutine, the logical `errpvmn` (see Table 7 of Appendix A) is set to .TRUE., and the appropriate error message is displayed. The PDMR then attempts to communicate a “PVM” error to the General Interface process, and consequently the RELAP5 process, using the space available in the neutronic logical

buffer. It should be noted that an error resulting from a PVM call may not allow for safe termination of all processes.

`errcode`:	associated with:
1	pvmfinitsend
2	pvmfpack
3	pvmfsend
4	pvmftrecv
5	pvmfunpack

`iunit`:	associated with:
1	PVM operations required in `PDMR_Obtain_IDs()`
2	PVM operations required in `PDMR_Send_Bufn()`
3	PVM operations required in `PDMR_Recv_Bufth()`
4	PVM operations required in `PDMR_Send_Mat()`
5	PVM operations required in `PDMR_Map_th2n()` to receive permuted thermal-hydraulic vector
6	PVM operations required in `PDMR_Map_n2th()` to send unpermuted neutronic vector

Finally, `PDMR_Proc_Errchk()` performs checks on the value of the error logicals sent from the General Interface, which includes both RELAP5 error logicals: `errcalcth`, `errdatath`, `errpvmth`, and General Interface error logicals: `errdatagi`, `errpvmgi`. In addition, this subroutine will check the value of the PARCS calculation error logical, `errcalcn`. The detection of any error from either the General Interface or RELAP5 process results in an error message being displayed indicating the origin of the error. At this point, the PDMR can begin the exit process without sending or receiving any additional data. If an error originates during the PARCS calculation, the PDMR will display the appropriate message and then send indication of a neutronic calculation fault to the General Interface process, and consequently the RELAP5 process, using the neutronic logical buffer. Once the buffer has been sent, the PDMR can begin the exit process.

II.G. Exit Handling

Exit handling in the PDMR and PARCS is invoked upon receipt of either a calculation- or fault-based signal. A fault-based signal results from a fatal error detected in the error checking module, as discussed in the previous section, and requires that all of the processes exit prematurely. If the error occurred either in the PDMR or PARCS, then the PDMR communicates the error the General Interface process, and consequently the RELAP5 process, using the next available control buffer send. Once the error has been communicated, the PDMR initiates its exit procedure, which is described below.

A calculation-based signal (`done`) is controlled by the RELAP5 process and is transferred to the PDMR through the thermal-hydraulic logical control buffer. Since RELAP5 does not know that the current time step is the last until after the calculation (i.e. after `done` has already been sent to PARCS), RELAP5 and PARCS will perform one final "stutter" step to allow RELAP5 to communicate the indication of normal termination. Thus, if the second unit of the PDMR detects that done is true, then upon return to PARCS, all calculations are bypassed and PARCS calls the third unit of the PDMR to send this normal termination signal back to the General Interface. Once this signal has been communicated, the PDMR will initiate its exit procedure, and upon return, PARCS will terminate its execution. This procedure can be inferred from Figure 1 in Section II.B.

Once termination of the PARCS/PDMR process has been determined, the subroutines `PDMR_Clean()` and `PDMR_Exit()` are called. `PDMR_Clean()` frees up any memory not previously de-allocated, and `PDMR_Exit()` removes the PARCS process from PVM. The coding which illustrates this procedure for subroutine `PDMR()` was shown in Section II.B.

II.H. Input Processing for the PDMR

The subroutines which are used to process user input for the PDMR and build the permutation matrices are contained in a separate module, `Map_Input`. The two subroutines which are utilized for these tasks are `Read_Map()` and `Build_Mat()`, respectively. In addition, a separate error checking subroutine, `Input_Errchk()`, is contained within module `Map_Input`, which is utilized to verify user input.

The input file which is read by `Read_Map()` is named `MAPTAB` (mapping table), and four card options are allowed: "TABLE1", "TABLE2", "TRIP", and "DOPL". These input cards are preceded by "%" in the first column (e.g. %TABLE1), and comment lines are preceded by "*" in the first column. The lines following an input card contain free format data relating to that card, where the input processor assumes that if the first column does not contain either "%" or "*", then the line being read corresponds to actual data. Subsequent lines of data will continue to be processed for the previously read input card until a new card is input. For example, if card "TABLE1" is read, the input processor will assume that the lines which follow correspond to this card until another card is detected such as "TABLE2". It should be noted that it is not required that the user input only one TABLE1 card and only one TABLE2 card. If these cards are entered

multiple times, the input processor will simply append the data that follows to the data which has already been read. However, for the “TRIP” and “DOPL” cards, only one line of data needs to be read following these cards. Thus, if multiple “TRIP” or “DOPL” cards are entered, the input processor will simply overwrite the previously read data with the new data.

Description of Input Cards:

TABLE1: This card relates to the mapping between thermal-hydraulic volumes and neutronic nodes. Each line of data which follows is read in free format and contains three numbers (integer, integer, real) corresponding to:

RELAP5 Volume Number *PARCS Node Number* *Weighting Factor*

TABLE2: This card relates to the mapping between heat structures and neutronic nodes. Each line of data which follows is read in free format and contains three numbers (integer, integer, real) corresponding to:

RELAP5 Heat Structure Number *PARCS Node Number* *Weighting Factor*

Note: The node numbering procedure used in PARCS was described in the SRS for the PDMR.

TRIP: This card provides the trip unit number used in RELAP5 for detecting control rod scram. Only one line of data needs to be read following this card, and this line, which is read in free format, contains the single integer:

RELAP5 Trip Unit Number

Note: If multiple lines are input, the input processor will overwrite the previous trip number with the new value.

DOPL: This card is used to determine the method of calculating Doppler temperature in the PARCS-Specific Data Map Routine. Only one line of data needs to be read following this card, and there are only two options for calculating Doppler temperature:

LINC *weighting factor*

(or)

AVG

If “AVG” is read, then the average fuel temperature from RELAP5 will be used and no other data is required. If “LINC” is read, then a linear combination of the centerline

and surface fuel temperatures will be used, and an additional weighting factor of type real is required to be input. This weighting factor is applied in the following manner:

$$T_{dopl} = (1-\omega)T_f^0 + \omega T_f^s$$

An example of the input file is provided below:

```
*
* Method of Calculating Doppler Temperature
*   LINC: Linear Combination (requires weighting factor)
*    AVG: Average Temperature
*
%DOPL
* LINC  0.7
 AVG
*
* Trip Unit Number for RELAP5
*
%TRIP
 501
*
* Volume to Node Table
*
%TABLE1
 335010000         1     1.0000
 335010000         2     1.0000
 335010000         3     1.0000
    ...          ...       ...
 335060000      3082     1.0000
 335060000      3083     1.0000
 335060000      3084     1.0000
*
* Heat Structure to Node Table
*
%TABLE2
   3360001         1     1.0000
   3360001         2     1.0000
   3360001         3     1.0000
     ...          ...       ...
   3360006      3082     1.0000
   3360006      3083     1.0000
   3360006      3084     1.0000
```

Description of Subroutine `Read_Map()`:

Subroutine `Read_Map()` processes the input file, `MAPTAB`, twice. On the first sweep (`iread=1`), the number of rows of mapping information for cards %TABLE1 and %TABLE2 are processed to determine the amount of memory to be allocated for variables `vol1(:)`, `vol2(:)`,

`vol3(:)`, `hs1(:)`, `hs2(:)`, and `hs3(:)`, which are described in Table 8 of Appendix A. Once the appropriate amount of space has been allocated, `Read_Map()` sweeps through the input file a second time to read in the data for each card. The following lines of code demonstrate this procedure:

```
      DO 20 iread=1,2
        OPEN(20,file='MAPTAB',status='old')

        IF (iread .EQ. 2) THEN
          ALLOCATE( vol1(nvolr), vol2(nvolr), vol3(nvolr))
          ALLOCATE( hs1(nhsr), hs2(nhsr), hs3(nhsr))
        ENDIF

        nvolr=0
        nhsr=0
  10    READ(20,'(a80)',end=100) line
        IF (line(1:1) .EQ. '*') GOTO 10
        IF (line(1:1) .EQ. '%') THEN
          READ(line(2:7),'(a6)') table
          GOTO 10
        ENDIF

c Read volume to neutronic node mapping information.
        IF (table .EQ. 'TABLE1') THEN
          nvolr=nvolr+1
          IF (iread .EQ. 2)
     &        READ(line,*,err=900,end=900)
     &           vol1(nvolr), vol2(nvolr), vol3(nvolr)

c Read heat structure to neutronic node mapping information.
        ELSEIF (table .EQ. 'TABLE2') THEN
          nhsr=nhsr+1
          IF (iread .EQ. 2)
     &        READ(line,*,err=900,end=900)
     &           hs1(nhsr), hs2(nhsr), hs3(nhsr)

c Read control rod trip unit number for RELAP5.
        ELSEIF (table(1:4) .EQ. 'TRIP') THEN
          IF (iread .EQ. 2) READ(line,*,err=900,end=900) itrip

c Determine method for calculating Doppler temperature.
        ELSEIF (table(1:4) .EQ. 'DOPL') THEN
          IF (iread .EQ. 2) THEN
            READ(line,*,err=900,end=900) cdopl
            IF (cdopl.EQ.'LINC') THEN
              idopl = 1
              READ(line,*,err=900,end=900) cdopl,wdopl
            ELSEIF (cdopl.EQ.'AVG') THEN
              idopl = 0
            ENDIF
          ENDIF
        ENDIF
        GOTO 10
```

```
 100    CONTINUE
        CLOSE(20)
  20  CONTINUE
```

Description of Subroutine `Build_Mat()`:

Once the input processing is complete, subroutine `Build_Mat()` is called to construct the two permutation matrices based on the mapping information supplied by the user. Prior to constructing the permutation matrices, arrays of distinct RELAP5 volume and heat structure numbers are obtained from the mapping input (`vol1`, `hs1`) and are sorted from smallest to largest. These arrays will be sent to the RELAP5 process during the initialization stage using the neutronic integer*4 control buffer (`cbufn`). In addition, the PARCS node numbers corresponding to volumes (vol2) and the node numbers corresponding to heat structures (hs2) will be scanned to construct arrays of distinct PARCS node numbers. These arrays, which are sorted from smallest to largest, will be used by the PDMR during the error checking procedure to verify the input node numbers against those in the PARCS database.

The construction of these arrays is similar for each and is shown below for the RELAP5 volume numbers (`ivol`) and the PARCS node numbers corresponding to volumes (`ivolnode`). This procedure uses temporary arrays (`itmp1`, `itmp2`) for storing the distinct numbers. Following the array construction, the number of elements in each array is known (`nvol` and `nvolnode`, respectively) and these dimensions are used to allocate space for `ivol` and `ivolnode`, to which `itmp1` and `itmp2` will be copied.

```
c Allocate temporary arrays used to store the distinct
c volume numbers and the distinct node numbers corresponding
c to the volumes.
      ALLOCATE( itmp1(nvolr))
      ALLOCATE( itmp2(nvolr))

      itmp1(1)=vol1(1)
      itmp2(1)=vol2(1)
      nvol=1
      nvolnode=1
      DO 50 i=2,nvolr

c Construct array of distinct volume numbers and order from
c lowest to highest.
        volflag = .TRUE.
        j=1
        DO 10 WHILE (volflag .AND. j.LE.nvol)
          IF (vol1(i) .EQ. itmp1(j)) volflag = .FALSE.
          j=j+1
  10    CONTINUE
c sort array.
        IF (volflag) THEN
          nvol=nvol+1
          index=nvol
```

```
          j=nvol-1
          lflag=.TRUE.
          DO 20 WHILE (lflag .AND. j.GE.1)
            IF (vol1(i) .LT. itmp1(j)) THEN
              index=j
            ELSE
              lflag=.FALSE.
            ENDIF
            j=j-1
  20      CONTINUE
          itmp1(index+1:nvol)=itmp1(index:nvol-1)
          itmp1(index)=vol1(i)
        ENDIF

c Construct array of distinct node numbers corresponding to
c the volumes and order from lowest to highest.
        volflag = .TRUE.
        j=1
        DO 30 WHILE (volflag .AND. j.LE.nvolnode)
          IF (vol2(i) .EQ. itmp2(j)) volflag = .FALSE.
          j=j+1
  30    CONTINUE
c sort array.
        IF (volflag) THEN
          nvolnode=nvolnode+1
          index=nvolnode
          j=nvolnode-1
          lflag=.TRUE.
          DO 40 WHILE (lflag .AND. j.GE.1)
            IF (vol2(i) .LT. itmp2(j)) THEN
              index=j
            ELSE
              lflag=.FALSE.
            ENDIF
            j=j-1
  40      CONTINUE
          itmp2(index+1:nvolnode)=itmp2(index:nvolnode-1)
          itmp2(index)=vol2(i)
        ENDIF
  50  CONTINUE

c Copy temporary arrays to permanent arrays.
      ALLOCATE( ivol(nvol))
      ALLOCATE( ivolnode(nvolnode))
      ivol(1:nvol) = itmp1(1:nvol)
      ivolnode(1:nvolnode) = itmp2(1:nvolnode)
      DEALLOCATE( itmp1, itmp2)
```

Following the construction of arrays `ivol`, `ivolnode`, `ihs`, and `ihsnode`, the thermal-hydraulic / heat structure to neutronic and the neutronic to thermal-hydraulic / heat structure permutation matrices are constructed. The first step in this procedure is to allocate space for the matrices which are stored in Coordinate Storage Format, as described in the SRS for the PDMR.

The size of the matrices are determined based on both the number of variables to be mapped (`nsubth2n`, `nsubhs2n`, `nsubn2th`, and `nsubn2hs`) and the number mappings input by the user (`nvolr`, `nhsr`):

```
c Allocate memory for the permutation matrices.
      nmatth = nsubth2n*nvolr + nsubhs2n*nhsr
      nmatn = nsubn2th*nvolr + nsubn2hs*nhsr
      ALLOCATE( matvalth(nmatth), matrowth(nmatth), matcolth(nmatth))
      ALLOCATE( matvaln(nmatn), matrown(nmatn), matcoln(nmatn))
```

Once the space for the matrices has been allocated, the arrays of row indices, column indices, and matrix elements are formed. The procedure for forming the heat structure to neutronic and neutronic to heat structure portions of the permutation matrices is similar to that for forming the portions of the permutation matrices relating volumes and nodes. Thus, the coding shown below depicts the construction of the arrays relating to only the volume/node portion of the permutation matrices. This procedure begins by looping over all of the rows of mapping information obtained from user input:

```
c T/H volume -> Neut. node  &  Neut. node -> T/H volume portions
c of the permutation matrices.
      DO 190 i=1,nvolr
```

Due to the fact that the volume to node submatrix is the transpose of the node to volume submatrix, it is possible to form the array of row indices for the volume to node submatrix simultaneously with the array of column indices for the node to volume submatrix, as shown below:

```
c form array of indices (matrowth, matcoln).
        j=1
        lflag=.TRUE.
        DO 130 WHILE (lflag .AND. j.LE.nvolnode)
          IF (vol2(i) .EQ. ivolnode(j)) THEN
            DO 110 k=1,nsubth2n
              l = (k-1)*nvolr + i
              matrowth(l) = j + (k-1)*nvolnode
 110        CONTINUE
            DO 120 k=1,nsubn2th
              l = (k-1)*nvolr + i
              matcoln(l) = j + (k-1)*nvolnode
 120        CONTINUE
            lflag=.FALSE.
          ENDIF
          j=j+1
 130    CONTINUE
```

Conversely, the array of column indices for the volume to node submatrix can be formed simultaneously with the array of row indices for the node to volume submatrix, as demonstrated in the following lines of code:

```
c form array of indices (matcolth, matrown).
        j=1
        lflag=.TRUE.
        DO 160 WHILE (lflag .AND. j.LE.nvol)
          IF (vol1(i) .EQ. ivol(j)) THEN
            DO 140 k=1,nsubth2n
              l = (k-1)*nvolr + i
              matcolth(l) = j + (k-1)*nvol
 140        CONTINUE
            DO 150 k=1,nsubn2th
              l = (k-1)*nvolr + i
              matrown(l) = j + (k-1)*nvol
 150        CONTINUE
            lflag=.FALSE.
          ENDIF
          j=j+1
 160    CONTINUE
```

The construction of the array of matrix elements for both the volume to node and node to volume portions of the permutation matrices is then performed using the volume/node weighting factors stored in `vol3(:)`, as shown below:

```
c form array of matrix values (matvalth, matvaln).
        DO 170 k=1,nsubth2n
          l = (k-1)*nvolr + i
          matvalth(l)=vol3(i)
 170    CONTINUE
        DO 180 k=1,nsubn2th
          l = (k-1)*nvolr + i
          matvaln(l)=vol3(i)
 180    CONTINUE
 190  CONTINUE
```

This procedure is then repeated for the heat structure to neutronic and neutronic to heat structure portions of the permutation matrices. Following this, the arrays used to store the input mapping information are deallocated:

```
c Deallocate all temporary input arrays.
      DEALLOCATE( vol1, vol2, vol3)
      DEALLOCATE( hs1, hs2, hs3)
```

Description of Subroutine `Input_Errchk()`:

Subroutine `Input_Errchk()` is utilized to check for errors in the user input, and is called from several points in subroutine `Read_Map()`. One call argument, errcode, is passed to `Input_Errchk()`, and is used for determining which of the eight error checks listed below is being performed.

(1) **[fatal error]** *Input file MAPTAB does not exist*:
result from: file MAPTAB not residing in the directory where the PARCS process is started.

(2) **[fatal error]** *Data was entered, but no card was specified*:
result from: data was processed (i.e. a '%' or '*' was not in the first column) before a card was input.

(3) **[fatal error]** *An error occurred while attempting to process line input*:
result from: error transfer control from a READ statement.

(4) **[fatal error]** *Invalid card type was input*:
result from: A card other than %TABLE1, %TABLE2, %TRIP, or %DOPL was input.

(5) **[fatal error]** *Card %TABLE1 or Card %TABLE2 was not input*:
result from: Card %TABLE1 or Card %TABLE2 was not found in the input file.

(6) **[fatal error]** *No data was entered for Card %TABLE1 or Card %TABLE2*:
result from: the number of rows of mapping input is less than one.
(i.e. `nvolr .lt. 1 .or. nhsr .lt.1`)

(7) **[warning]** *Card %TRIP or Card %Dopl was not input*:
result from: Card %TRIP or Card %DOPL was not found in the input file.

(8) **[warning]** *Invalid option for calculating Doppler temperature was input*:
result from: An option other than "AVG" or "LINC" was input following Card %DOPL.

If any of the fatal errors (`errcode` equals 1 through 6) is detected, `errinp` is set to .TRUE. and an error message is displayed. If any of the warnings (`errcode` equals 7 or 8), a warning message is displayed but code execution is not terminated. Prior to returning from `Read_Map()`, `errinp` is copied to `errdatan`, and if the value of this logical is .TRUE., the PDMR will initiate the fault handling procedure described in Section II.G, which includes bypassing the call to subroutine `Build_Mat()`.

III. Summary

This document described the software design necessary to satisfy the requirement specifications for the PARCS-Specific Data Map Routine. The major components outlined here provide the basis for code development and relate to the variable requirements, the design of the functional units, and the process flow through the PARCS/PDMR code. In addition, the design of the input and error checking procedures were described and are consistent with the specified requirements.

The PARCS/PDMR code is designed as an independent process, and the incorporation within the framework of the coupled RELAP5/PARCS code requires the use of both the General Interface and a message-passing protocol. The latter utilizes the PVM package to control the communication between the separate processes. The internal flow control through the functional units of the PDMR is treated logically and entry into these units is controlled by PARCS, as described in the previous sections.

As mentioned, the design of the PDMR is specific to the RELAP5/PARCS coupling, and as such requires that the data transferred between RELAP5 and PARCS be well-defined. The Software Requirements Specification for both the RDMR and PDMR provided this basis, and this document has described the procedure necessary to achieve the required data coherency. Finally, the design and implementation described by this document is consistent with that for General Interface.

IV. References

1. D. Barber and T. Downar, “Software Requirements Specification for the PARCS-Specific Data Map Routine in the Coupled RELAP5/PARCS Code,” Technical Report, PU/NE-98-11, Purdue University, (1998).

2. D. Barber and T. Downar, “Software Requirements Specification for the General Interface in the Coupled Code,” Technical Report, PU/NE-98-8, Purdue University, (1998).

3. W. Wang, G. Mortensen, D. Barber, and T. Downar, “Software Requirements Specification for the RELAP5-Specific Data Map Routine in the Coupled RELAP5/PARCS Code,” Technical Report, SCIE-NRC-368-98, SCIENTECH, Inc., (1998).

4. Y. Saad, “Numerical Methods for Large Eigenvalue Problems,” Manchester University Press, Manchester, UK., pp. 40-41 (1992).

5. D. Barber and T. Downar, “Software Design and Implementation Document for the General Interface in the Coupled Code,” Technical Report, PU/NE-98-10, Purdue University, (1998).

Appendix A: Variable Description

The following tables provide a description of the variables used by the General Interface routines:

Table 1: PVM and Process Control Variables

Name	Type	Dimension	Description	Range
done	logical	1	logical flag: execution is stopped when done=.true.	.true. / .false.
ioutp	int*4	1	unit number for output	$0 < i < 100$
ntasks	int*4	1	number of processes	$i = 3$
group	char*6	1	6-character descriptor for the group of processes	“procs”
tidn	int*4	1	process ID for the PARCS module	$0 < i < 2^{32}$
tidgi	int*4	1	process ID for the General Interface module	$0 < i < 2^{32}$
mtypen	int*4	1	message tag associated with the PARCS module	$i = 3$
mtypegi	int*4	1	message tag associated with the General Interface module	$i = 1$
inum	int*4	1	instance number in group	$0 \le i \le 2$
istride	int*4	1	striding of data in the buffer	$0 < i < 2^{32}$
info[(a)]	int*4	1	integer error flag for PVM calls	$-(2^{32}) < i < 2^{32}$
timeout	real*4	1	int*4 parameter indicating the max # of seconds to wait on a receive	$i = 600$
dimbuf	int*4	6	dimension of each of the 6 data type-dependent control buffers, in order, as shown in Tables 2 and 3	$0 \le i < 2^{32}$

[(a)] Values less than zero indicate an error has occurred in a PVM call.

Table 2: RELAP5 Data and Control Buffers

Name	Type	Dimension	Description	Range
cbufth[(a)]	char*6	dimbuf(1)	control buffer of 6-character words	N/A
lbufth[(a)]	logical	dimbuf(2)	control buffer of logical words	.true. / .false.
i2bufth[(a)]	int*2	dimbuf(3)	control buffer of 16 bit integer words	$-(2^{16}) < i < 2^{16}$
i4bufth[(a)]	int*4	dimbuf(4)	control buffer of 32 bit integer words	$-(2^{32}) < i < 2^{32}$
r4bufth[(a)]	real*4	dimbuf(5)	control buffer of 32 bit floating point words	$-(10^{38}) < x < 10^{38}$
r8bufth[(a)]	real*8	dimbuf(6)	control buffer of 64 bit floating point words	$-(10^{308}) < x < 10^{308}$
nvecthp	int*4	1	dimension of vecthp received from General Interface process	$0 < i < 2^{32}$
vecthp	real*8	nvecthp	permuted vector of space-dependent thermal-hydraulic and heat structure data	$0 \le x < 10^{308}$ [(b)]

[(a)] Used for both initial and time-dependent control information
[(b)] Individual ranges are set for each component of the vector as described in Section II.F.

Table 3: PARCS Data and Control Buffers

Name	**Type**	**Dimension**	**Description**	**Range**
cbufn[(a)]	char*6	dimbuf(1)	control buffer of 6-character words	N/A
lbufn[(a)]	logical	dimbuf(2)	control buffer of logical words	.true. / .false.
i2bufn[(a)]	int*2	dimbuf(3)	control buffer of 16 bit integer words	$-(2^{16}) < i < 2^{16}$
i4bufn[(a)]	int*4	dimbuf(4)	control buffer of 32 bit integer words	$-(2^{32}) < i < 2^{32}$
r4bufn[(a)]	real*4	dimbuf(5)	control buffer of 32 bit floating point words	$-(10^{38}) < x < 10^{38}$
r8bufn[(a)]	real*8	dimbuf(6)	control buffer of 64 bit floating point words	$-(10^{308}) < x < 10^{308}$
recvn	logical	1	indication that both permutation matrices are sent from the PARCS process	.true.
nvecn	int*4	1	dimension of vecn sent to General Interface process	$0 < i < 2^{32}$
vecn	real*8	nvecn	vector of space-dependent neutronic data	$0 \le x < 10^{308}$ [(b)]

[(a)] Used for both initial and time-dependent control information
[(b)] Individual ranges are set for each component of the vector as described in Section II.F.

Table 4: Variables Associated with the Permutation Matrices

Name	Type	Dimension	Description	Range
nmatth	int*4	1	dimension of the arrays describing the permutation matrices: matvalth, matrowth, and matcolth.	$1 \le i \le n$; $n = nvecthp*nvecth$
matvalth	real*8	nmatth	non-zero elements in the permutation matrix used to map thermal-hydraulic and heat structure data to neutronic nodes.	$0 < x \le 1$
matrowth	int*4	nmatth	row number corresponding to each element in matvalth	$1 \le i \le nvecthp$
matcolth	int*4	nmatth	column number corresponding to each element in matvalth	$1 \le j \le nvecth$
nmatn	int*4	1	dimension of the arrays describing the permutation matrices: matvaln, matrown, matcoln.	$1 \le i \le n$; $n = nvecnp*nvecn$
matvaln	real*8	nmatn	non-zero elements in the permutation matrix used to map neutronic data to thermal-hydraulic zones and heat structure components.	$0 < x \le 1$
matrown	int*4	nmatn	row number corresponding to each element in matvaln	$1 \le i \le nvecnp$
matcoln	int*4	nmatn	column number corresponding to each element in matvaln	$1 \le j \le nvecn$

Table 5: Parameters and Dimension Variables

Name	Type	Dimension	Description	Range
nsubth2n	int*4	1	parameter identifying the number of thermal-hydraulic-to-neutronic variables	$i = 5$
nsubhs2n	int*4	1	parameter identifying the number of heat structure-to-neutronic variables	$i = 3$
nsubn2th	int*4	1	parameter identifying the number of neutronic-to-thermal-hydraulic variables	$i = 1$
nsubn2hs	int*4	1	parameter identifying the number of neutronic-to-heat structure variables	$i = 1$
nvol	int*4	1	number of thermal-hydraulic volumes	$0 < i < 2^{32}$
nhs	int*4	1	number of heat structures	$0 < i < 2^{32}$
nvolnode	int*4	1	number of neutronic nodes corresponding to thermal-hydraulic volumes	$0 < i < 2^{32}$
nhsnode	int*4	1	number of neutronic nodes corresponding to heat structures	$0 < i < 2^{32}$
nrowth2n	int*4	1	total number of rows in thermal-hydraulic/heat structure to neutronic matrix	$i = \max(\text{matrowth})$
ncolth2n	int*4	1	total number of columns in thermal-hydraulic/heat structure to neutronic matrix	$i = \max(\text{matcolth})$
nrown2th	int*4	1	total number of rows in neutronic to thermal-hydraulic/ heat structure matrix	$i = \max(\text{matrown})$
ncoln2th	int*4	1	total number of columns in neutronic to thermal-hydraulic/heat structure matrix	$i = \max(\text{matcoln})$

Table 5: Parameters and Dimension Variables

ivol	int*4	nvol	array of RELAP5 volume numbers	$0 < i < 2^{32}$
ihs	int*4	nhs	array of RELAP5 heat structure numbers	$0 < i < 2^{32}$
ivolnode	int*4	nvolnode	array of PARCS node numbers corresponding to volumes	$0 < i < 2^{32}$
ihsnode	int*4	nhsnode	array of PARCS node numbers corresponding to heat structures	$0 < i < 2^{32}$

Table 6: Miscellaneous Calculational Control Variables

Name	Type	Dimension	Description	Range
hsdscr	char*6	nhs	array of heat structure geometry descriptors sent from RELAP5	“PIN” or “SLAB”
itrip	int*4	1	RELAP5 control rod trip card number (user input)	$0 < i < 2^{32}$
idopl	int*4	1	method of performing doppler temperature calculation 0 - use average fuel temperature 1 - use linear combination of centerline and surface temps.	0 or 1
wdopl	real*4	1	weighting factor to be used if linear combination option is chosen (user input)	$0 \leq x \leq 1$

Table 7: Variables Associated with the Error Checking

Name	Type	Dimension	Description	Range
errcalcn	logical	1	indication of a calculation error in PARCS	.true. / .false.
errdatan	logical	1	indication of a data error in the PDMR	.true. / .false.
errpvmn	logical	1	indication of a PVM error in the PDMR	.true. / .false.
errinit	logical	1	indication of error detected (errcalcn .or. errdatan .or. errpvmn) in the Initialization functional unit of the PDMR	.true. / .false.
errth2n	logical	1	indication of error detected (errcalcn .or. errdatan .or. errpvmn) in the Thermal-Hydraulic to Neutronic unit of the PDMR	.true. / .false.

Table 7: Variables Associated with the Error Checking

errn2th	logical	1	indication of error detected (errcalcn .or. errdatan .or. errpvmn) in the Neutronic to Thermal-Hydraulic unit of the PDMR	.true. / .false.
errdatagi	logical	1	indication of a data error in the General Interface	.true. / .false.
errpvmgi	logical	1	indication of a PVM error in the General Interface	.true. / .false.
errcalcth	logical	1	indication of a calculation error in RELAP5	.true. / .false.
errdatath	logical	1	indication of a data error in the RELAP5-specific data map routine	.true. / .false.
errpvmth	logical	1	indication of a PVM error in the RELAP5-specific data map routine	.true. / .false.
errgi	logical	1	indication of error in General Interface (errdatagi .or. errpvmgi)	.true. / .false.
errth	logical	1	indication of error on RELAP5 side (errcalcth .or. errdatath .or. errpvmth)	.true. / .false.

Table 8: Variables Associated with the Mapping Input Module (`Map_Input`)

Name	**Type**	**Dimen-sion**	**Description**	**Range**
nvolr	int*4	1	number of rows read in for volume-to-node correspondence table	$0 < i < 2^{32}$
nhsr	int*4	1	number of rows read in for heat structure-to-node correspondence table	$0 < i < 2^{32}$
vol1	int*4	nvolr	column 1 of volume table (RELAP5 volume number)	$0 < i < 2^{32}$
vol2	int*4	nvolr	column 2 of volume table (corresponding PARCS node number)	$0 < i < 2^{32}$
vol3	real*4	nvolr	column 3 of volume table (weighting factor)	$0 < x \leq 1$
hs1	int*4	nhsr	column 1 of heat structure table (RELAP5 heat structure number)	$0 < i < 2^{32}$
hs2	int*4	nhsr	column 2 of heat structure table (corresponding PARCS node number)	$0 < i < 2^{32}$
hs3	real*4	nhsr	column3 of heat structure table (weighting factor)	$0 < x \leq 1$
errinp	logical	1	indication of error during input processing	.true. / .false.
Variables Used By: `Read_Map()` & `Input_Errchk()`				
cdopl	char*4	1	doppler calculation method read from user input	“LINC” or “AVG”
table	char*6	1	card identifying the table to be read in following this card	“TABLE1” or “TABLE2” or “TRIP” or “DOPL”
line	char*80	1	temporary variable used to read in the lines from the input file	N/A

Table 8: Variables Associated with the Mapping Input Module (`Map_Input`)

flagt1	logical	1	indicates whether Card `%TABLE1` was input	.true. / .false.
flagt2	logical	1	indicates whether Card `%TABLE2` was input	.true. / .false.
flagtr	logical	1	indicates whether Card `%TRIP` was input	.true. / .false.
flagdp	logical	1	indicates whether Card `%DOPL` was input	.true. / .false.
Local Variables: `Read_Map()`				
iread	int*4	1	index for input processing: 1 - process dimensions 2 - process data	1 or 2
Local Variables: `Build_Mat()`				
volflag	logical	1	flag used to identify unique volume (and corresponding node) numbers read from input	.true. / .false.
hsflag	logical	1	flag used to identify unique heat structure (and corresponding node) numbers read from input	.true. / .false.
lflag	logical	1	flag used to sort the volume (and corresponding node) numbers, and the heat structure (and corresponding node) numbers	.true. / .false.
itmp1	int*4	nvolr -or- nhsr	temporary variable used to store the sorted volume and heat structure numbers	$0 < i < 2^{32}$
itmp2	int*4	nvolr -or- nhsr	temporary variable used to store the sorted neutronic node numbers corresponding to volumes and heat structures	$0 < i < 2^{32}$
Local Variables: `Input_Errchk()`				
errcode	int*4	1	identifier of which error check is being performed	$1 \le i \le 8$
filexist	logical	1	indicates whether file `MAPTAB` exists	.true. / .false.

Table 9: PARCS Variables Utilized in the PDMR

Name	Type	Dimension	Description
		`cntl.h`	
ppm	real	1	global boron concentration
rstrt	logical	1	new logical for restart calculation (currently not used)
calcterm	logical	1	new logical for communicating calculation termination from PDMR to PARCS
errpdmr	logical	1	new logical for communicating PDMR errors to PARCS
errparcs	logical	1	new logical for communicating PARCS errors to PDMR
errextth	logical	1	new logical for communicating an error outside the PDMR (i.e. in General Interface or RELAP5) to PARCS
		`trcntl.h`	
time	real	1	value of the current time point
delt	real	1	current time step size
deltd	real	1	previous time step size
scram	logical	1	indication of a control rod scram
rsted	logical	1	new logical indicating that a restart edit should be performed (currently not used)
majed	logical	1	new logical indicating that a major output edit should be performed (currently not used)
		`geom.h`	
nz	integer	1	number of axial neutronic planes
nxy	integer	1	number of radial neutronic nodes
volcore	real	1	volume of active core
		`fbvar.h`	
tdopl	real	(mxnzth, mxnchan) [(a)]	space-dependent Doppler temperatures in $\sqrt{K}$

Table 9: PARCS Variables Utilized in the PDMR

tcool	real	(mxnzth, mxnchan)	space-dependent moderator temperatures in Celsius
dcool	real	(mxnzth, mxnchan)	space-dependent moderator liquid density in kg/m^3
`thgeom.h`			
pfa	real	1	fuel assembly pitch
hac	real	1	active fuel assembly height
`thopdata.h`			
powfa	real	1	power per fuel assembly in megawatts
fracdc	real	1	fraction of power deposited in coolant
fracdf	real	1	fraction of power deposited in fuel
`power.h`			
plevel	real	1	fractional power level
relp	real	(mxnzth, mxnchan)	fractional relative power

[a] Parameters declared in thparam.h.

Appendix B: Subroutine Report

Subroutines Deleted:

none

Subroutines Added:

Subroutine PDMR(): Logic Controller for the PARCS-Specific Data Map Routine.
Uses Modules: PDMR_Var_Decl, PDMR_Error_Check

Contains Subroutines:

Subroutine PDMR_Init(): Controls the Initialization functional unit.
Uses Module: PDMR_Init_Calc, Map_Input

Subroutine PDMR_th2n(): Controls the Thermal-Hydraulic to Neutronic functional unit.
Uses Module: PDMR_Time_Calc

Subroutine PDMR_n2th(): Controls the Neutronic to Thermal-Hydraulic functional unit.
Uses Module: PDMR_Time_Calc

Subroutine PDMR_Clean(): Frees up memory for arrays not previously de-allocated.

Subroutine PDMR_Exit(): Removes PARCS process from PVM.

Module PDMR_Init_Calc: Contains the subroutines used during the initialization stage.
Uses Module: PDMR_Var_Decl, PDMR_Error_Check

Contains Subroutines:

Subroutine PDMR_Obtain_IDs(): Establishes communication with the General Interface process.

Subroutine PDMR_Buf_Init(): Communicates initial control buffers with the General Interface process.
Uses Module: PDMR_Comm_Buf

Subroutine PDMR_Send_Mat(): Sends the permutation matrices to the General Interface process.

Subroutines Added (cont.):

Module PDMR_Time_Calc: Contains the subroutines used for the time-dependent data mapping.
Uses Module: PDMR_Var_Decl, PDMR_Error_Check

Contains Subroutines:

Subroutine PDMR_Buf_th2n(): Receives thermal-hydraulic control buffer from the General Interface process.
Uses Module: PDMR_Comm_Buf

Subroutine PDMR_Buf_n2th(): Sends neutronic control buffer to the General Interface process.

Subroutine PDMR_Map_th2n(): Receives permuted vector of thermal-hydraulic data from the General Interface process and stores in appropriate PARCS memory locations.

Subroutine PDMR_Map_n2th(): Builds the vector of PARCS space-dependent powers and sends the vector to the General Interface process.
Uses Module: PDMR_Comm_Buf

Module PDMR_Comm_Buf: Contains the subroutines used to process and communicate control buffers with the General Interface process.
Uses Module: PDMR_Var_Decl, PDMR_Error_Check

Contains Subroutines:

Subroutine PDMR_Recv_Bufth(): Receive thermal-hydraulic control buffer from the General Interface process.

Subroutine PDMR_Send_Bufn(): Send neutronic control buffer to the General Interface process.

Module PDMR_Error_Check: Contains the subroutines used to perform the error checking for the PARCS-Specific Data Map Routine.
Uses Module: PDMR_Var_Decl

Contains Subroutines:

Subroutine PDMR_Data_Errchk(): Perform error checking on data used by PARCS-Specific Data Map Routine.

Subroutine PDMR_PVM_Errchk(): Perform error checking on PVM status variables.

Subroutine PDMR_Proc_Errchk(): Check the value of error logicals sent from the General Interface which relate to both thermal-hydraulic and General Interface errors.

Subroutines Added (cont.):

Module PDMR_Var_Decl: Declares the variables used by the PARCS-Specific Data Map Routine.

Subroutines (PARCS) Modified:

parcs.f - Added logic to route the calculation flow through the PDMR if PARCS is coupled to an external thermal-hydraulics code.

inputd.f - Added user input option for running PARCS in “stand alone” mode coupled to an external thermal-hydraulics code.

init.f - Added a call to xsecfb.f if PARCS is running coupled to an external thermal-hydraulics code. This is needed because init.f is called after the call to the thermal-hydraulic/heat structure to neutronic mapping unit of the PDMR and prior to the steady-state eigenvalue problem solution. Thus, the cross sections need to be updated based on feedback from the thermal-hydraulics code.

initxsec.f - The indices for the feedback variables were changed to correspond to neutronic nodes if using external thermal-hydraulics code. This was needed because if PARCS is running in “stand alone” mode, these variables correspond to the thermal-hydraulic nodes used in PARCS, where PARCS handles the mapping between the thermal-hydraulic nodes and the neutronic nodes.

initth.f - Minor change to turn off PARCS thermal-hydraulic input edits. Also, changed the indices on relp (relative power) to correspond to neutronic nodes if using external thermal-hydraulics code.

outerss.f - Modified outer iteration control for the steady-state eigenvalue problem if using external thermal-hydraulics code. This is needed because PARCS iterates between the thermal-hydraulic solution and the neutronic solution if running in “stand alone” mode.

xsecfb.f - See initxsec.f

norm.f - If PARCS is running in “stand alone” mode, no power normalization is performed if the user turns off thermal-hydraulic feedback. If PARCS is running coupled to an external thermal-hydraulics code, then the feedback is effectively always on, and the power is normalized accordingly.

relpow.f - changed indices on relp and absp (absolute power) to correspond to neutronic nodes. (See initth.f)

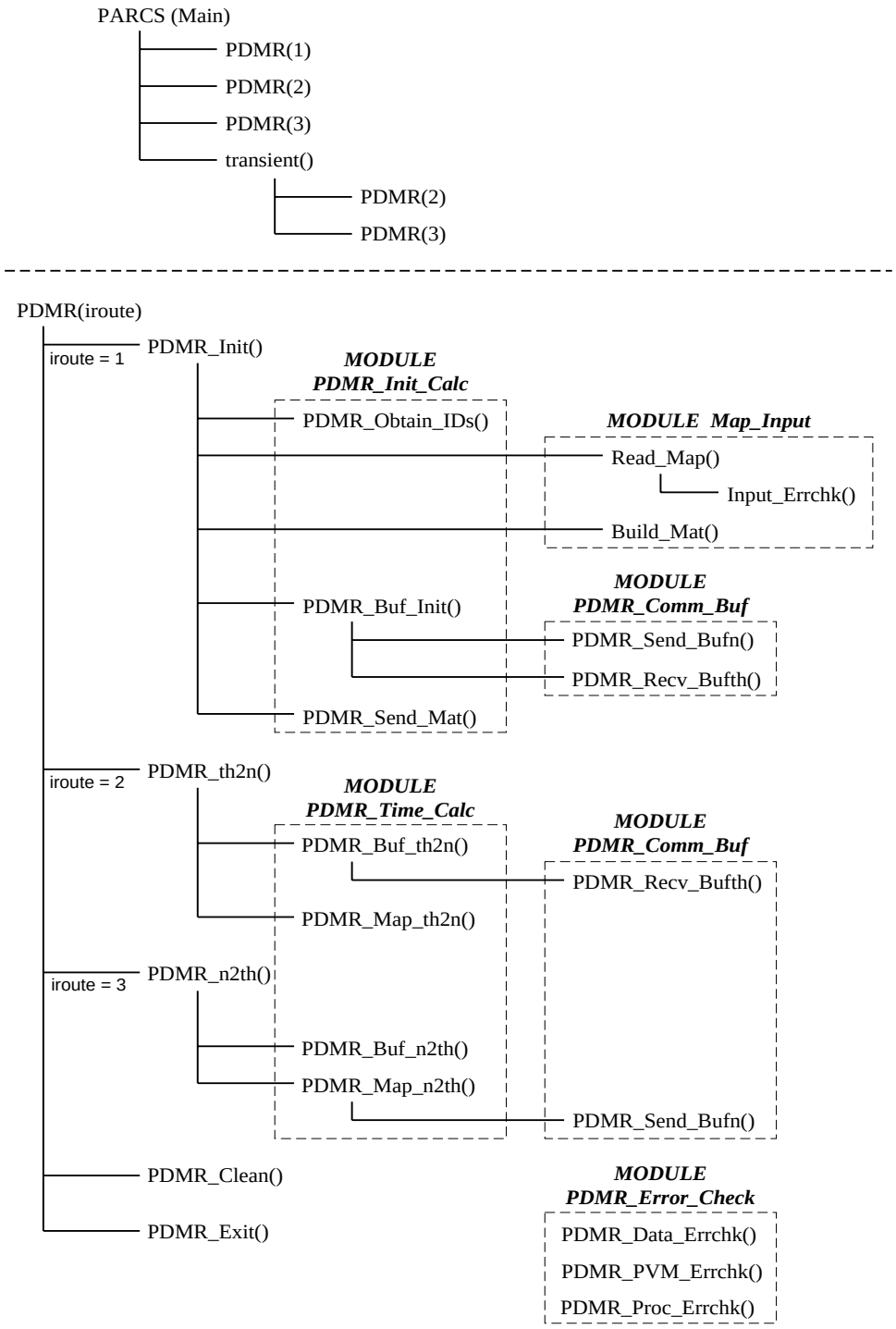
Subroutines (PARCS) Modified (cont.):

transient.f - Added logic to route the calculation flow through the PDMR and turned off the PARCS thermal-hydraulics calculation if PARCS is coupled to an external thermal-hydraulics code.

inittran.f - Control variables relating to the PARCS thermal-hydraulic update are turned off if PARCS is running coupled to an external thermal-hydraulics code.

outer.f - See outerss.f

Subroutine Calling Tree:



Note: The subroutines contained in module `PDMR_Error_Check` are called from most subroutines, and thus, the paths of entry into these error checking subroutines are not shown.