Chapter 7

Using DPEAC/CDPEAC in Programs

The most common use of DPEAC and/or CDPEAC in a CM-5 program is for writing highly efficient subroutines that are called from a program written in a high-level language. This chapter presents a simple example of just such a subroutine, shows how it can be written in either DPEAC or CDPEAC, and demonstrates how to call it from a CM Fortran program.

7.1 Example: An Arithmetic Subroutine

The subroutine described in this chapter calculates a specific arithmetic formula,

$$d = \frac{b^2 + c}{\sqrt{3.69a + 25.0b}}$$

elementwise across a set of four array arguments, **`a`**, **`b`**, **`c`**, and **`d`**. Each of these variables represents an element of a high-level array that is passed into the DPEAC or CDPEAC subroutine. The high-level program that calls this subroutine handles allocation of the arrays and subsequent processing of the results produced by the subroutine.

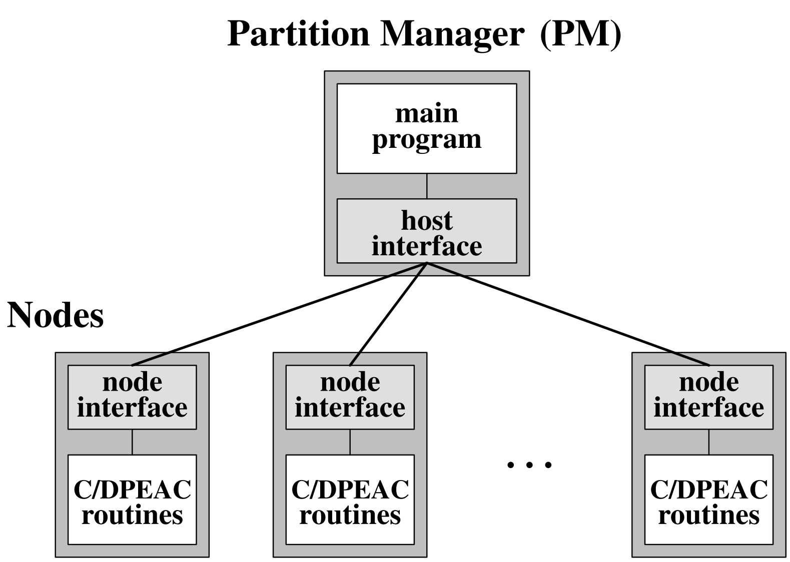
Note: You do *not* have to structure your programs as shown in this chapter to make use of the CM-5's vector units. The CM Fortran and C* compilers automatically define DPEAC routines in the process of compiling standard CM programs, and thus implicitly use the vector units whenever they are needed. The methods shown in this chapter allow you to duplicate the compiler's work for specific routines that you choose to write by hand.

7.2 Low-Level Program Structure

A CM-5 program that includes user-written DPEAC or CDPEAC routines has four main parts, each of which is contained in a separate source code file:

- The *DPEAC or CDPEAC subroutines*, which execute in identical copies on each of the processing nodes.
- The *node interface* functions, one for each DPEAC or CDPEAC routine, which define which node routines can be called from the PM.
- The *host interface* functions, on the PM, which broadcast a call to the node interface functions on all the nodes.
- The *main program*, written in a high-level language (such as CM Fortran), which calls the host interface functions to invoke the node subroutines.

The overall program structure is as shown below:



The host and node interface files describe the relationship between a specific set of function calls made on the PM, and a specific set of functions that are defined on the nodes. The interface files provide the “glue” that allows these function calls and definitions to compile and link correctly.

7.2.1 Program Files

Thus, a program with DPEAC or CDPEAC subroutines has four component source code files:

- A *main program* file, written in a high-level language.
- A *host interface* file, containing the definitions of all host interface functions called in the main program.
- A *node interface* file, containing the definitions for all node interface functions called in the host interface file.
- A *subroutine code* file, containing the definitions of all DPEAC/CDPEAC routines called from the node interface functions.

In addition, there is typically a *makefile* that is used to build the program via the UNIX **`make`** utility.

Source File Naming Conventions

The tools used to compile and link a program with DPEAC/CDPEAC routines impose the following restrictions on the program files:

- The host interface file must be written in C, and its name must end with the suffix "**`.c`**".
- The node interface file must also be written in C, but its name must end with the suffix "**`.pe`**". When the program is compiled, this file is run through a filter that produces a "**`.c`**" file for compilation.
- The subroutine code file must be written in DPEAC or CDPEAC, and must have the suffix "**`.pe.dp`**" (for a DPEAC routine file) or "**`.pe.cdp`**" (for a CDPEAC routine file).

It is a convention of the compilers and linkers used on the CM-5 that all object files containing code to be executed on the nodes must have the suffix "**`.pe.o`**". The suffix restrictions described above ensure that all object files produced in the compilation process will have the correct object file suffixes for the linker.

7.2.2 Host/Node Interface Naming Conventions

The tools used to compile and link a program with DPEAC/CDPEAC routines also impose the following restrictions on function names used in the program:

- The *host interface function* for each routine can have any legal name in the main program, but it must be defined in the host interface file with the same name, *in all lower case*, and with a trailing underscore "**`_`**" added.

  (For example, in the sample program below, the host interface function is called **`NODECALC`** in the CM Fortran source file, and **`nodecalc_`** in the host interface file.)

- The *node interface function* for each routine can have any legal function name in the host interface file, but its definition in the node interface file must have the same name with the prefix "**`CMPE_`**" attached.

  (In the sample program, the node interface function is called **`nodecalc`** in the host interface file, and **`CMPE_nodecalc`** in the node interface file.)

- The *DPEAC (or CDPEAC) subroutine* can have any legal function name in the node interface file, but its definition in the subroutine file must have the same name (and, in a DPEAC subroutine file, must have a leading underscore "**`_`**" character added).

  (In the sample program, the subroutine name used is **`nodecalc`** in the node interface file, **`nodecalc`** in the CDPEAC subroutine file, and **`_nodecalc`** in the DPEAC subroutine file.)

For the Curious: These special prefix and case requirements make it easy for the compiler and linker to determine which host and node interface functions correspond to which routines in the main program and the DPEAC code file.

The "**`CMPE_`**" prefix for the names of node functions callable from the host has the additional purpose of making it unlikely that a random function name used in the main program would happen to match a function name defined in the host/node interface.

7.3 Passing Arrays into DPEAC and CDPEAC Routines

The arguments of a DPEAC or CDPEAC subroutine depend on the manner in which the entire program is executing on the CM. For example, in programs that manipulate parallel arrays (such as the sample program in this chapter), the DPEAC or CDPEAC routine on each node handles the array subgrid that is stored in that node's memory.

In this case, the arguments to the DPEAC or CDPEAC subroutine are typically the memory addresses of the array subgrids located in the node's memory. The subroutines on each node handle the subgrids stored on that node, in such a way that every element of each array argument is handled by some node in the CM.

(There are other ways to pass data into DPEAC or CDPEAC routines. For example, you can use OS routines to allocate parallel memory yourself — Appendix H describes how to do this. You can then pass the addresses of these parallel memory regions into DPEAC or CDPEAC subroutines. However, this method of argument passing is not discussed further in this chapter.)

In CM Fortran, arrays are not referenced by the address of the array data itself, but instead by a pointer to a data structure known as an *array descriptor*. This descriptor contains, among other things, the address of the start of the array and the number of elements in the array.

Array descriptors are stored on the *partition manager*. The array location in the descriptor is a memory address in *node* memory. Thus, part of the job of the host interface function is to get the array location from the descriptors of any array arguments, and pass these memory addresses on to the node interface function.

The contents of an array descriptor can be extracted by calls to special accessor functions that are part of the CM Run-Time System (CMRTS). For example:

```
CMCOM_cm_address_t CMRT_desc_get_cm_location
(arr_desc);
int CMRT_desc_get_subgrid_size(arr_desc);
      CMRT_desc_t arr_desc;
```

CMRT_desc_get_cm_location returns the starting address (in node memory) of the array described by **arr_desc**.

CMRT_desc_get_subgrid_size returns the number of elements of the array that are stored in the memory of each VU (the "subgrid size" of the array). This value is required by the DPEAC routine, which must determine how many memory locations to operate on.

7.4 Sample Program Source Files

The sample program described below consists of five files:

- A main program written in CM Fortran: **`main.fcm`**
- A host interface file: **`host.c`**
- A node interface file: **`interface.pe`**
- A DPEAC subroutine file: **`dpeac_code.pe.dp`**
- A CDPEAC subroutine file: **`cdpeac_code.pe.cdp`**

The DPEAC and CDPEAC subroutine files contain the same routine, written appropriately for each of the two instruction sets.

The program is designed so that it can be compiled with either the DPEAC or the CDPEAC subroutine file; the main program and host/node interface files are identical in both cases. A sample makefile is also provided; this makefile can be used to compile the program with either (or both) of the subroutine files.

7.5 The Main CM Fortran Program (main.fcm)

The CM Fortran program **`main.fcm`** does three things:

- It initializes its arrays by assigning

  ```
  a = 3.0
  b = random numbers between 0.0 and 1.0
  c = 19.0
  d = 0.0
  ```

- It evaluates the formula **`d = (b*b+c)/sqrt(3.69*a+25.0*b)`** twice: first, by a ordinary CM Fortran expression (which is internally compiled into DPEAC code by the CM Fortran compiler); second, by a call to the host interface function **`NODECALC`**.

- The program then prints out the argument arrays and the computed results, for each of the two methods, to demonstrate that the two methods do in fact return the same values.

The **main.fcm** program is as follows:

```
       program main
       parameter (length=32)
       real a(length), b(length), c(length)
       real dh(length), dn(length)
       a=3.0
       b=0.0
       call CMF_RANDOM(b)
       c=19.0
c Host calculation
       dh=0.0
       dh=(b*b+c)/sqrt(3.69*a + 25.0*b)
c Node calculation
       dn=0.0
       call NODECALC(a,b,c,dn)
c Display results for comparison
       print *,''
       print *,'Computing d=(b*b+c)/sqrt(3.69*a + 25.0*b):'
       print *,' Item ',' A=  ',' B=  ',' C=  ',' Host ','
Node '
       do 10 i=1,length
          print 900, i, a(i),b(i),c(i),dh(i),dn(i)
 10   continue
       print *,''
       stop
 900  format (i6,f6.2,f6.2,f6.2,f6.2,f6.2)
       end
```

7.6 The Host Interface File (host.c)

The host interface file contains a single function, **nodecalc_**, which does three things:

- It calls **CMRT_desc_get_cm_location** once for each of the array arguments to get the actual node memory locations of the arrays.
- Because all the array arguments must have the same size and shape, the host interface function calls **CMRT_desc_get_subgrid_size** just once to get the subgrid size of the array arguments.
- Finally, the host interface function makes a call to the corresponding **nodecalc** function to execute the DPEAC routine.

The **host.c** host interface file is as follows:

```
#include<cm/cmrt.h>
void nodecalc_ (a,b,c,d)
      CMRT_desc_t a,b,c,d;
{
   CMCOM_cm_address_t aloc,bloc,cloc,dloc;
   int size;

   /* get memory location for each array */
   aloc=CMRT_desc_get_cm_location(a);
   bloc=CMRT_desc_get_cm_location(b);
   cloc=CMRT_desc_get_cm_location(c);
   dloc=CMRT_desc_get_cm_location(d);

   /* subgrid size is same for all arrays */
   size=CMRT_desc_get_subgrid_size(a);

   /* call node interface function */
   nodecalc(aloc,bloc,cloc,dloc,size);
}
```

7.7 The Node Interface File (interface.pe)

The node interface file contains one node interface function, **CMPE_node**. **CMPE_node** takes the array addresses and subgrid size provided by the host function and passes them directly to the DPEAC (or CDPEAC) subroutine.

The **interface.pe** node interface file is as follows:

```
void CMPE_nodecalc(aloc,bloc,cloc,dloc,size)
      unsigned aloc,bloc,cloc,dloc,size;
{
   CMPE_nodecalc(aloc,bloc,cloc,dloc,size);
}
```

Note: This file is passed through a filter, **mkpestubs**, which converts it into an appropriate C code file. (This filtering step is handled internally by the makefile.)

7.8 The DPEAC Subroutine File (dpeac_code.dp)

This file contains the DPEAC version of the arithmetic subroutine:

```
#include <cmsys/dpeac.h>

      dpentry _CMPE_nodecalc,0,0 ! Entry point

! By convention, function args are in SPARC ”input”
! registers, %i0, %i1, etc.

! Symbolic names for registers:
! Note that ”%” prefix is used explicitly in code
! to make SPARC/VU register distinction clear.
# define A i0
# define B i1
# define C i2
# define D i3
# define Size i4

! By default, CM Fortran sets vector length to 8,
! and vector mask mode to ”always”. The following
! is insurance; when it is not needed, it is simply redundant

      set_vector_length_and_vmmode 8, always

#define VECTOR_LENGTH 8

! Formula being evaluated is:
! d=(b*b+c)/sqrt(3.69*a + 25.0*b)

Loop:
      floadv  [%B]:4, V2  ! (Short format, memory stride)
                  !  Load subgrid slice of B into V2,
                  !  striding by 4 bytes for each of
                  !  the 8 vector elements.
      add %B,(4*8),%B ! (SPARC instruction)
                  !  Bump array pointer B to next slice
                  !  in subgrid (4 bytes * 8 elements)
      floadv  [%C]:4, V3; \
      fmadav  V2,V2,V3! (Short format, chain-loading)
                  !  Load subgrid slice of C into V3,
                  !  striding by 4 for 8 elements,
                  !  and mult-add V3=(B*B)+C in same
                  !  operation.
      add %C,(4*8),%C ! (SPARC instruction)
                  !  Bump array pointer B to next slice
                  !  in subgrid (4 bytes * 8 elements)
```

```
floadv  [%A]:4, V4; \
fmulv   V4, 0r3.69, V5  \
             ! (Immediate format, chain-loading)
             !  Load subgrid slice of A into V4,
             !  striding by 4 for 8 elements,
             !  and multiply V5=(3.69*V4) in same
             !  operation.
add %A,(4*8),%A ! (SPARC instruction)
             !  Bump array pointer A to next slice
             !  in subgrid (4 bytes * 8 elements)
fmadav  V2, 0r25.0, V5  ! (Immediate format)
             !  Mult-add V5=(25.0*B)
fisqtv  V5, V5      ! (Short format)
             !  Calculate V5=1/SQRT(V5)
fmulv   V5, V3, V5  ! (Short format)
             !  Multiply V5=(V3*V5)
fstorev [%D]:4, V5  ! (Short format, memory stride)
             !  Store result in D subgrid slice
             !  striding 4 bytes for 8 elements.
addcc   %Size,-VECTOR_LENGTH,%Size \
             ! (SPARC instruction)
             !  Subtract vector length (8) from
             !  subgrid size argument to see if
             !  there are subgrid slices left
bne Loop     ! (SPARC instruction)
             !  If result is non-zero,
             !  go back and do next subgrid slice.
add %D,(4*8),%D ! (SPARC instruction, DELAY SLOT)
             !  Bump array pointer D to next slice
             !  in subgrid (4 bytes * 8 elements)

dpretn       ! (DPEAC Accessor Instruction)
             !  Return from DPEAC subroutine
```

A few notes on the structure of this program:

- Note that the **floadv** and **fstorev** instructions explicitly specify the memory stride as 4. An alternative to this would be to set the value of the **dp_stride_memory** register to 4.

- CM Fortran sets the following VU control registers to these defaults:

dp_vector_length	8
dp_stride_rs1	0
dp_stride_memory	0
dp_vector_mask_mode	always

dp_vector_mask_direction	right
dp_alu_mode_fast	fast, not IEEE

Nevertheless, it is always a good precaution to set these registers to the values you require within your DPEAC code routines, to avoid unnecessary surprises should these defaults change.

- Note that the array size is assumed to be a multiple of 8. Since the vector length is set to 8, there is no remainder, or “tail” of leftover elements. To handle a more general case, any “tail” of remaining values would need to be processed in a separate section of code, by resetting the vector length to the tail length, and repeating the calculation just once for the tail values.

7.9 The CDPEAC Subroutine File (cdpeac_code.cdp)

This file contains the CDPEAC version of the arithmetic subroutine:

```
#include <cm/cdpeac.h>
/* CDPEAC sample program.
   Formula being evaluated is:
   d=(b*b+c)/sqrt(3.69*a + 25.0*b) */
CMPE_nodecalc(aloc,bloc,cloc,dloc,size)
      unsigned aloc,bloc,cloc,dloc,size;
{
   /* Initialize SPARC registers for CDPEAC */
   dpsetup();

   /* By default, CM Fortran sets vector length to 8,
      and vector mask mode to ”always”. The following
      is insurance; when it is not needed, it is simply
      redundant */
   set_vector_length_and_vmmode(8,ALWAYS);

   /* Loop over each 8-element subgrid slice */
   for ( ; size ; size -= 8 );
   {
      loadv_u(f,bloc,4,V2); /*  (Short format, memory stride)
                                 Load subgrid slice of B into V2,
                                  striding by 4 bytes for each of
                                  the 8 vector elements. */
      bloc += (4*8);         /* Bump array pointer of B to new slice
                               in subgrid (4 bytes * 8 elements) */
```

```
    join2(            /* (Join of memory and ALU operations) */
        loadv_u(f,cloc,4,V3), /* (Short format, chain-loading)
                                 Load subgrid slice of C into V3,
                                 striding by 4 bytes for each of
                                 the 8 vector elements. */
      madav(f,V2,V2,V3)    /*  Mult-add V3=(B*B)+C
                                in same operation. */
      );          /* (End of join2 macro) */

    cloc += (4*8);    /* Bump array pointer of C to next slice
                           in subgrid (4 bytes * 8 elements) */

    join2(            /* (Join of memory and ALU operations) */
        loadv_u(f,aloc,4,V4),
                           /* (Immediate format, chain-loading)
                               Load subgrid slice of A into V4,
                               striding by 4 for 8 elements. */
      mulvi(f,V4,3.69,V5)      /* multiply V5=(3.69*V4)
                                in same operation. */
      );          /* (End of join2 macro) */

     aloc += (4*8);       /* Bump array pointer of A to new slice
                               in subgrid (4 bytes * 8 elements)*/

    madavi(f,V2,25.0,V5); /* (Immediate format)
                               Mult-add V5=(25.0*B) */

     isqtv(f,V5,V5);      /* (Short format)
                               Calculate V5=1/SQRT(V5) */

     mulv(f,V5,V3,V5);        /* (Short format)
                               Multiply V5=(V3*V5) */

    storev_u(f,dloc,4,V5);    /* (Short format, memory stride)
                               Store result in D subgrid slice
                               striding 4 bytes for 8 elements. */

     dloc += (4*8);       /* Bump array pointer of D to new slice
                               in subgrid (4 bytes * 8 elements)*/
  }
  /* Clean up VU control registers -- NOTE: not always needed */
  dpcleanup()
}
```

7.10 Makefile for the Sample Program (Makefile)

Below is a sample **`Makefile`** that can be used with the UNIX utility program **`make`** to compile and link the sample program described above.

For those who have not used **`make`** before, all you have to do is place the five code files plus this **`Makefile`** into the same directory, set that directory as the current one (i.e., **`cd`** to it in UNIX), and then type **`make`** to build the program. (You will want to be logged on to a CM-5 partition manager when you do this, so that you will have access to the appropriate compilers and libraries.)

Note: When compiling this program with **`make`**, you can select either of the two subroutine code files by providing an appropriate argument. For example:

`make dpeac`	builds the DPEAC version of the program (**`run_dp`**).
`make cpdeac`	builds the CDPEAC version (**`run_cdp`**).

By default, this **`Makefile`** builds both executable versions of the program.

Once you have used **`make`** to build the executable files (**`run_dp`** and/or **`run_cdp`**), you can run the program by typing the appropriate executable file name. (Again, you'll want to be logged on to a CM-5 partition manager.)

The **`Makefile`** shown here performs a number of different operations to bring the pieces of the sample program together:

- The main CM Fortran program is compiled by **`cmf`** to produce two object files, one for the PM (**`main.o`**) and one for the nodes (**`main.pe.o`**).
- The host interface program is compiled by **`cc`**, producing an object code file (**`host.o`**) for the PM.
- The node interface program is compiled by **`dpcc`** and then passed through a stubs filter (**`mkpestubs`**) to produce a PM object file (**`pe-call.o`**).
- The appropriate subroutine file(s) are assembled by **`dpas`**, producing node object files (**`dpeac_code.o`** and/or **`cdpeac_code.o`**).
- Finally, the various object code files are linked together (again by **`cmf`**) to produce the executable file(s) (**`run_dp`** and/or **`run_cdp`**).

The **`Makefile`** also includes a number of "suffix rule" definitions, which describe how the various code source files are compiled.

The **Makefile** is as follows:

```
# ---------------------------------------------
# Makefile to assemble C/DPEAC example programs
# By William R. Swanson, 5/5/93
# ---------------------------------------------

# === Setup Definitions ===
# Don't display commands while building program
.SILENT:

# Alias macros that are used to clarify Make syntax
SOURCE_FILE   = $<
OBJECT_FILE   = $@

# debugging: set to -g to compile for debuggers
DEBUG     =

# === Target File Names ===
# Names of final executable files
DPEAC_EXECUTABLE  = run_dp
CDPEAC_EXECUTABLE = run_cdp

# Names of source and object files
MAIN      = main
HOST_INTF = host
NODE_INTF = interface
DPEAC_CODE= dpeac_code
CDPEAC_CODE   = cdpeac_code

# Object file sets
HOST_OBJS   = $(MAIN).o $(HOST_INTF).o $(NODE_INTF).o
DPEAC_NODE_OBJS    = $(MAIN).pe.o $(DPEAC_CODE).pe.o
CDPEAC_NODE_OBJS = $(MAIN).pe.o $(CDPEAC_CODE).pe.o
DPEAC_OBJS = $(HOST_OBJS) $(DPEAC_NODE_OBJS)
CDPEAC_OBJS    = $(HOST_OBJS) $(CDPEAC_NODE_OBJS)

# === Top-level Rules ===

# By default, trigger build of both executables
default: dpeac cdpeac

# To rebuild, do a clean and then trigger both builds
scratch: clean default

# Trigger build of just dpeac executable
dpeac: $(DPEAC_EXECUTABLE)

# Trigger build of just cdpeac executable
cdpeac: $(CDPEAC_EXECUTABLE)
```

```
# === Cleanup Rules ===

cleanobj:
	echo "Removing old objects, stub files, etc."
	rm -f *.o *.s $(NODE_INTF).c
	rm -f *# *#[0-9]* *% *~

clean: cleanobj
	echo "Removing old executable files"
	rm -f $(DPEAC_EXECUTABLE) $(CDPEAC_EXECUTABLE)

# === Program-specific Build Rules ===

# CMF driver is used to handle linking:
LINKER    = $(CMF)
LINKFLAGS = $(CMFFLAGS)

# Main linking step that builds the executable program:
$(DPEAC_EXECUTABLE): $(DPEAC_OBJS)
	echo "Linking ($(LINKER)) $(DPEAC_OBJS)"
	echo "to make executable file \"$(DPEAC_EXECUTABLE)\""
	$(LINKER) $(LINKFLAGS) $(DPEAC_OBJS) -o $(DPEAC_EXECUTABLE)

# Main linking step that builds the executable program:
$(CDPEAC_EXECUTABLE): $(CDPEAC_OBJS)
	echo "Linking ($(LINKER)) $(CDPEAC_OBJS)"
	echo "to make executable file \"$(CDPEAC_EXECUTABLE)\""
	$(LINKER) $(LINKFLAGS) $(CDPEAC_OBJS) -o
$(CDPEAC_EXECUTABLE)

# Host stubs obj file is produced from node interface file
interface.o: interface.c

# All other compilation steps are handled by suffix rules

# === Suffix Rules ===
# Add CMF and DPEAC suffixes to SUFFIX variable:
SUFFIXES += .fcm .dp .cdp .pe

# Clear out default suffix-list and install new list:
.SUFFIXES:
.SUFFIXES: $(SUFFIXES)

# To compile a C file, run it through $(CC)
CC        = cc
CFLAGS    = -DCM5_DASH -O $(DEBUG)
.c.o:
	echo "Compiling ($(CC)) $(SOURCE_FILE) into $(OBJECT_FILE)"
	$(CC) $(CFLAGS) -c $(OBJECT_FILE) $(SOURCE_FILE)
```

```
# To compile a CMF file, run it through $(CMF)
# Note: This step produces _two_ object files: .o and .pe.o
CMF       = cmf2.0
CMFFLAGS  = -cm5 -vu -O -Zcmld -Bstatic $(DEBUG)
NOLINK    = -c
LINK      =
.fcm.o:
      echo "Compiling ($(CMF)) $(SOURCE_FILE) into
$(OBJECT_FILE) and $(OBJECT_FILE:.o=.pe.o)"
      $(CMF) $(CMFFLAGS) $(NOLINK) $(SOURCE_FILE)

# To assemble a DPEAC file, run it through $(DPAS)
DPAS      = /usr/bin/dpas
DPFLAGS       = -t
.dp.o:
      echo "Assembling ($(DPAS)) $(SOURCE_FILE) into
$(OBJECT_FILE)"
      $(DPAS) $(DPFLAGS) -o $(OBJECT_FILE) $(SOURCE_FILE)

# To assemble a CDPEAC file, run it through $(DPCC)
# This produces one object file: .o
DPCC      = /usr/bin/dpcc
DPCCFLAGS =
.cdp.o:
      echo "Assembling ($(DPCC)) $(SOURCE_FILE) into
$(OBJECT_FILE)"
      $(DPCC) $(DPCCFLAGS) -o $(OBJECT_FILE) $(SOURCE_FILE)

# To process a DPEAC node interface file, run it through
$(MKSTUB)
MKSTUB    = /usr/bin/mkpestubs
MKSTUBFLAGS   = -n
.pe.c:
      echo "Processing ($(MKSTUB)) $(SOURCE_FILE) into
$(OBJECT_FILE)"
      echo '#include <cm/cmcom_types.h>' > $(OBJECT_FILE)
      $(MKSTUB) $(SOURCE_FILE) $(MKSTUBFLAGS) >> $(OBJECT_FILE)
```

7.11 Sample Run of the Program

Here is a UNIX session in which the sample program is built and run. (The following assumes that you have logged on to the partition manager of a CM-5, and are currently in a directory containing the five code files and the makefile.)

```
%: make clean
Removing old objects, stub files, etc.
Removing old executable files

%: ls
Makefile              host.c              out
cdpeac_code.pe.cdp    interface.pe
dpeac_code.pe.dp      main.fcm

%: make
Compiling (cmf2.0) main.fcm into main.o and main.pe.o
cmf [CM5 VecUnit 2.0 Beta 2]
Compiling main.fcm
Compiling (cc) host.c into host.o
Processing (/usr/bin/mkpestubs) interface.pe into \
      interface.c
Compiling (cc) interface.c into interface.o
Assembling (/usr/bin/dpas) dpeac_code.pe.dp into \
      dpeac_code.pe.o
Linking (cmf2.0) main.o host.o interface.o \
      main.pe.o dpeac_code.pe.o \
      to make executable file ”run_dp”
cmf [CM5 VecUnit 2.0 Beta 2]
Linking...done.
Assembling (/usr/bin/dpcc) cdpeac_code.pe.cdp into \
      cdpeac_code.pe.o
Linking (cmf2.0) main.o host.o interface.o \
      main.pe.o cdpeac_code.pe.o \
      to make executable file ”run_cdp”
cmf [CM5 VecUnit 2.0 Beta 2]
Linking...done.
24.1u 16.5s 2:22 28% 0+676k 21+652io 177pf+0w
```

```
%: run_dp

Computing d=(b*b+c)/sqrt(3.69*a + 25.0*b):
 Item    A=     B=     C=    Host  Node
     1   3.00   0.77 19.00   3.56   3.56
     2   3.00   0.77 19.00   3.55   3.55
     3   3.00   0.67 19.00   3.68   3.68
     4   3.00   0.59 19.00   3.81   3.81
     5   3.00   0.19 19.00   4.78   4.78
     6   3.00   0.44 19.00   4.09   4.09
     7   3.00   0.20 19.00   4.73   4.73
       . . .  < other values omitted >
    30   3.00   0.88 19.00   3.44   3.44
    31   3.00   0.99 19.00   3.34   3.34
    32   3.00   0.39 19.00   4.21   4.21

 FORTRAN STOP

%: run_cdp
Computing d=(b*b+c)/sqrt(3.69*a + 25.0*b):
 Item    A=     B=     C=    Host  Node
     1   3.00   0.06 19.00   5.34   5.34
     2   3.00   0.88 19.00   3.43   3.43
     3   3.00   0.24 19.00   4.60   4.60
     4   3.00   0.25 19.00   4.60   4.60
     5   3.00   0.54 19.00   3.90   3.90
     6   3.00   0.04 19.00   5.48   5.48
     7   3.00   0.91 19.00   3.41   3.41
       . . .  < other values omitted >
    30   3.00   0.50 19.00   3.97   3.97
    31   3.00   0.41 19.00   4.16   4.16
    32   3.00   0.06 19.00   5.38   5.38

 FORTRAN STOP
```