Quick Reference for shmdefine

CONCURRENT COMPUTER CORPORATION™

0898010-020
July 2004

Printed in U. S. A.

Preface

Scope of Manual

This guide is designed to assist you in getting started with use of the **shmdefine** utility.

Structure of Manual

This manual consists of one chapter.

- Chapter 1 introduces you to the **shmdefine** utility, describes the syntax of the command as well as its options, and guides the user through an example demonstrating the steps necessary for programs to share data when targeting a PowerMAX OS system and a second example which targets a Linux system.

Syntax Notation

The following notation is used throughout this manual:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*s.
list bold	User input appears in **list bold** type and must be entered exactly as shown. Names of directories, files, commands, options and system manual page references also appear in **list bold** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appear in `list` type.
`[ ]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such options or arguments

Referenced Publications

The following publications are referenced in this document:

0890240	*hf77 Fortran Reference Manual*
0890497	*Concurrent C/C++ Reference Manual*
0890516	*MAXAda Reference Manual*

Contents

Chapter 1 Using shmdefine

1 Using shmdefine

The **shmdefine** utility is designed to facilitate the use of shared memory by a set of cooperating programs. Although you may have a number of programs that will cooperate in using one or more shared memory segments, it is necessary to invoke the utility only once. Because **shmdefine** produces object files that must be linked to the source object file, you must invoke it prior to linking.

shmdefine currently operates with the Concurrent C/C++, Fortran, and Ada compilers (**ec**, **ec++**, **xf77**, and MAXAda™) for programs that will execute on PowerMAX OS™ target systems. For programs that will execute on RedHawk™ Linux® target systems, **shmdefine** works with MAXAda and Concurrent Fortran as well as the GNU C, C++, and Fortran compilers.

The initialization file generated by **shmdefine** contains an executable function to access the shared memory services at program start-up time. On PowerMAX OS target systems, the function is automatically called before the main program starts. However, on target systems running Linux, the function must be explicitly called before any shared memory regions are referenced.

The linker map file (PowerMAX OS) or linker command script (Linux) generated by **shmdefine** describes the shared memory regions to the linker.

Command Syntax

The format for executing the **shmdefine** utility is as follows:

shmdefine [**-CGHNZ**] [**-b** *base_name*] [**-t** *target*] [*files*]

NOTE

Options beginning with '**--**' are valid only on Linux hosts.

The options are as follows:

-C
--case-sensitive

Pay attention to case when interpreting symbols in the input. **shmdefine** interprets symbols in a case-insensitive manner by default.

-G
--gnu-f77

On Linux, use GNU Fortran naming conventions for common blocks. By default, **shmdefine** uses Concurrent Fortran naming conventions. If you are using GNU Fortran to compile common block definitions, you should use this option to ensure that your common block data is shared properly.

NOTE

This option is only valid on RedHawk Linux target systems since GNU Fortran is not supported on PowerMAX OS. When generating files for use on a PowerMAX OS target, the **-G**/**--gnu-f77** option is ignored.

-H
--help

Display the help screen and stop.

-N
--no-copy

On PowerMAX OS, do not copy initial data into the shared memory region. It is the user's responsibility to ensure the proper initialization of a shared memory region.

On Linux and PowerMAX OS, suppress calls to data initialization subprograms specified in the `INIT USING` clause in any **shmdefine** input file (see "Initialization" on page 1-6).

-Z
--allow-nubbins

Allow regions with a size <= 0.

-b *base_name*
--base *base_name*

Use *base_name* as the prefix of the generated output files. **shmdefine** will append **.sm.c** and **.sm.ld** to generate the output files. The default prefix is **shm_init**.

-t *os*
--target *os*

Generate output files suitable for the target operating system *os*. Valid values for *os* include **pmax** and **linux**.

If no *files* are specified, **shmdefine** will read standard input. Otherwise, each file is parsed, in order, until all have been read. When all input has been consumed, **shmdefine** will generate a C source file and a matching linker map file or linker command script describing the defined shared memory region(s).

The C source file generated for use on a PowerMAX OS target contains an executable function that accesses shared memory services at program start-up time. The C source file also takes care of data initialization. If an `INIT USING` clause in the **`shmdefine`** input file specifies a callback subprogram, the C source file also makes the call to the specified subprogram. Data initialization of variables that are associated with a shared memory segment via **`shmdefine`** and calls to subprograms specified in `INIT USING` clauses occur when a newly executed program attaches to the corresponding shared memory segment - if and only if it is the only program currently attached to that segment.

The C source file generated for a Linux target contains an executable function that accesses shared memory services at the time the user calls it. If an `INIT USING` clause in the **`shmdefine`** input file specifies a callback subprogram, the C source file also makes the call to the specified subprogram. Calls to subprograms specified in `INIT USING` clauses occur when a newly executed program attaches to the corresponding shared memory segment - if and only if it is the only program currently attached to that segment. Linux programs which desire access to shared memory regions defined using **`shmdefine`** must call `shm_init()` to initialize the shared memory before using it, and must call `shm_rm()` when finished to release and/or destroy the shared memory.

WARNING

Linux targets do not provide automatic data initialization of the shared memory segment even if the original source code specified initial values for the variables associated with the shared memory segment. See "Initialization" on page 1-6 for information on initializing shared regions.

The default name for the generated C source file is **`shm_init.sm.c`**.

The following table describes how to call the `shm_init()` and `shm_rm()` subprograms from each supported language.

Language	**Initialize / Destroy**

C	`extern void shm_init(void);`
	`extern void shm_rm (void);`
	
	`shm_init();`
	`shm_rm();`

Fortran	`call shm_init` `call shm_rm`

Ada	`procedure shm_init;`
	`procedure shm_rm;`
	
	`pragma import (C, shm_init);`
	`pragma import (C, shm_rm);`
	
	`shm_init();`
	`shm_rm();`

The linker map file (PowerMAX OS) or linker command script (Linux) describes the shared memory segments to the linker. The default name for this file is **shm_init.sm.ld**.

Shared Regions

Input to the **shmdefine** utility defines the shared memory segment or segments that are to be used by cooperating programs. You may define the segments using standard input, or you may specify one or more files that contain the definitions. Although input in either case may be free-form, the general format for defining a shared memory segment is as follows:

```
[TARGET os]
[FORTRAN COMPILER compiler]
SHARED REGION region_name
    [INIT USING subprogram_name]
    [attribute1, attribute2, ...]
    variable_clause1, variable_clause2, ...
END SHARED REGION
```

Note that blanks, tabs, and newlines are recognized only as separators. The hash character (#) can be used to indicate that the rest of the line is a comment.

Target

The optional `TARGET` clause specified in the shared memory definition (see "Shared Regions" on page 1-5) takes `LINUX`, `POWERMAX_OS`, or `PMAX` as a parameter.

If a `TARGET` clause is specified, it is checked for consistency with any command line '**-t** *os*' target option. If the specified targets do not match, the command line option will override the target specified in the `TARGET` clause and a warning will be issued.

If the `TARGET` clause is not specified, the native operating system is assumed.

Fortran Compiler

The optional `FORTRAN COMPILER` clause specified in the shared memory definition (see "Shared Regions" on page 1-5) takes `GNU` or `CONCURRENT` as a parameter.

If a `FORTRAN COMPILER` clause is specified, it is checked for consistency with any **-G**/**--gnu-f77** command line option (see "Command Syntax" on page 1-1). If the specified compilers do not match, the command line option will override the *compiler* specified in the `FORTRAN COMPILER` clause and a warning will be issued.

If a `FORTRAN COMPILER` clause is not specified, the Concurrent Fortran compiler's naming conventions are assumed for any common blocks specified in the shared region.

Initialization

An INIT USING clause, if specified, provides a way to modify the behavior of the automatic initialization that occurs on program startup under PowerMAX OS, or of the shm_init() function under Linux, for associated shared regions (see "Shared Regions" on page 1-5).

After attaching the shared memory region, callbacks are made to INIT USING subprograms if and only if the number of processes attached to the shared region is 1. This provides a means to perform more complicated initializations of data residing in a shared memory region.

Three parameters are passed to the INIT USING callback subprogram, which is expected to have the following profile:

```
void init_callback (char *   region_name,
                   void *   start_address,
                   int      length) ;
```

The *region_name* is the same name specified in the SHARED REGION clause in the **shmdefine** input file. The *start_address* and *length* define the space allocated to the shared region.

On PowerMAX OS, any initial data values are copied to the shared region before any INIT USING subprogram is called.

On Linux, the shared region will be uninitialized at the time the INIT USING subprogram is called, even if initial values were supplied in the source files defining the variables that are associated with the shared region.

The **-N** option may be used to suppress the calling of subprograms specified in an INIT USING clause (see "Command Syntax" on page 1-1).

Attributes

Attributes that can be specified in the shared memory definition (see “Shared Regions” on page 1-5) are presented in Table 1-1.

Table 1-1. Attributes

Attribute	Purpose
`ADDRESS`	Enables you to specify a starting virtual address for the shared memory segment.
`IPC`	Enables you to set the control flags for the segment.
`SHM_LOCAL`	Enables you to set the NUMA policy for the shared memory segment to the anchored soft-local policy. A soft-local policy allows pages to be allocated from global memory when pages are not available for allocation from the local memory pool. This option has no effect on systems without local memory.
`SHM_HARD`	Enables you to set the NUMA policy for the shared memory segment to the anchored hard-local policy. A hard-local policy causes a process to wait for local memory pages to become available if the pages cannot be allocated from the local memory pool when needed. This option has no effect on systems without local memory.
`KEY`	Enables you to specify a user-chosen identifier for the segment.
`MODE`	Enables you to set the permissions that are associated with the segment.
`SHM_RDONLY`	Enables you to prevent a process from writing to the segment.

Variables

The following types of variables may be associated with the shared memory segment (see “Shared Regions” on page 1-5):

- C external variables

  External variables must be declared with the type qualifier `volatile` in the C source program.

- Ada variables

  Ada variables should be declared volatile via pragma `volatile` and must be exported via pragma `export` in the Ada source program.

- Fortran common blocks

  When using Concurren Fortran, common blocks should be declared `VOLATILE` in the Fortran source program.

NOTE

GNU Fortran does not support the `VOLATILE` keyword.

- Concurrent Fortran pointer blocks
- Concurrent Fortran datapool dictionaries

The volatile declaration informs the compiler that the values of the variables may be modified in a way that is unknown to the compiler.

Variables are associated with the shared memory segment using the following variable clauses:

- `C EXTERN` *external_name* `[ SIZE` *n* `[ *` *m* `]]`

  The C external variable *external_name* is included in the current region.

- `Fortran COMMON` *common_block_name* `[ SIZE` *n* `[ *` *m* `]]`

  The Fortran common block *common_block_name* is included in the current region.

- `Fortran BLANK COMMON [ SIZE` *n* `[ *` *m* `]]`

  The Fortran unnamed ("blank") common block is included in the current region.

- `Fortran DATAPOOL` *datapool_name*`,` "*filename.o*"

  The variables from *datapool_name* listed in the Fortran datapool dictionary *filename.o* are included in the current region.

NOTE

Fortran datapools are only supported by Concurrent Fortran compilers; use of this clause with other compilers may not allow the program to link

- `Fortran SIZEOFBLOCK` *name* `[ *` *count* `]  [SIZE` *n* `[ *` *m* `]]`

NOTE

Fortran pointer blocks are only supported by Concurrent Fortran compilers; use of this clause with other compilers may not allow the program to link

Reserves space the size of the Fortran pointer block *name* in the current region. The space is eight-byte aligned, the size is rounded up to an eight-byte multiple, and the start and end addresses of the space are marked with the names `sblock__`*name* and `eblock__`*name*. **`get_sblock_addr(3F)`** and **`get_eblock_addr(3F)`** return these addresses.

If *count* is specified, space is reserved in the current region for *count* contiguous copies of the Fortran pointer block *name*. Each copy is eight-byte aligned, and its size is rounded up to an eight-byte multiple. **`get_block_copy_addr(3F)`** returns the start address of a specific copy of the pointer block. **`get_block_numcopies(3F)`** returns *count*.

NOTE

The optional `SIZE` clause in the above definitions is required when generating files for a Linux operating system target, but is ignored when generating files for a PowerMAX OS target.

It is important to note that space in the shared memory segment is allocated to variables in the same order in which the variables are specified in the input to **`shmdefine`**.

PowerMAX OS Example

This example demonstrates how to enable a C program, a Fortran program, and two Ada programs to cooperate in using a shared memory segment on a PowerMAX OS target.

- Binding shared memory to physical memory (see page 1-10)
- Creating source programs (see page 1-11)
- Creating shmdefine input files (see page 1-14)
- Executing shmdefine (see page 1-14)
- Compiling the initialization output file (see page 1-14)
- Compiling and linking the source programs (see page 1-15)
- Running the programs (see page 1-16)

When executed, the C program named **generate** places data into a shared memory segment, the Fortran program named **process** performs a computation on each item of data stored in the segment, the Ada program named **init** initializes the `iready` and `oready` variables, and the Ada program named **output** writes the result of each computation to the standard output.

Binding shared memory to physical memory

*If you wish to bind the shared memory segment to a particular section of physical memory, configure the target system such that the specified shared memory segment already exists and is bound to the appropriate physical address. The PowerMAX OS command **shmconfig** can be used to accomplish this.*

Since our example does not require that the shared memory segment be bound to a section of physical memory, this step is not required.

Creating source programs

Create source programs, and include in them a volatile type declaration for each program variable that is to reside in shared memory.

For our example, create the following C, Fortran, and Ada source programs using a text editor of your choice:

- **generate.c** (see "generate.c" on page 1-11)
- **process.f** (see "process.f" on page 1-12)
- **output.a** (see "output.a" on page 1-13)

NOTE

The shm_init() procedure is automatically called during program startup.

generate.c

```
/* This program creates 10 integer values and passes the
    data to the cooperating programs using the shared memory
    structure sm_data.
*/

 #include <stdio.h>

 volatile struct sm_data {
   int   ain, aout;
   int   iready, oready;
} shared_data__ ;

 void main () {
   int   i;
   int   accum = 2;

    for (i = 1 ; i <= 10 ; ++i) {
      while (shared_data__.iready > 0) {
         sleep(1);
      }
      shared_data__.ain = accum;
      shared_data__.iready = i;
      accum *= 2;
   }
}
```

process.f

```
C   This program processes the input data and places the
C   results of the calculations in another shared memory
C   segment for output.  On input, it waits for the iready
C   variable to be equal to its count of the data.  When it has
C   processed that datum, it negates the iready variable to
C   tell the input program it is ready for another one.  A
C   similar scheme is used for communicating to the output
C   program.

      PROGRAM process
      COMMON /shared_data/ ain, aout, iready, oready
      INTEGER ain, aout
      INTEGER iready, oready
      VOLATILE shared_data
      INTEGER   i

       DO i = 1,10
         DO WHILE (iready .NE. i .OR. oready .GE. 0)
            CALL sleep(1)
         END DO
         aout =  - ain
         iready = - iready
         oready = i
      END DO
      END
```

output.a

```
---- This file contains the source for two programs.  ”init”
---- initializes the iready and oready variables.  ”output”
---- writes the results from the process program.  It waits
---- for the oready variable to be equal to its count of the
---- the data.  When that happens, it writes the results and
---- negates the oready variable.

package external_data is
----
   type data_items is new integer range -10 .. 10 ;
   subtype data_item_id is data_items range 1..10 ;

   type common_block is
      record
         ain, aout : integer ;
         iready, oready : data_items ;
      end record ;
   shared_data : common_block ;

   pragma export (C, shared_data, "", "shared_data__") ;
----
end external_data ;

with external_data ;
procedure init is
   use external_data ;
begin
   shared_data.iready := data_items’first ;
   shared_data.oready := data_items’first ;
end init ;

with external_data ;
with ada.text_io ;
procedure output is
   use external_data ;
begin
   for id in data_item_id loop
      ---- Wait for the [next] data item
      ---- to be ready for output.
      while (shared_data.oready /= id) loop
         delay (1.0) ;
      end loop ;

      ---- Print the data.
      ada.text_io.put ("result");
      ada.text_io.put (data_items'image(id));
      ada.text_io.put (" = ");
      ada.text_io.put (integer'image(shared_data.aout));
      ada.text_io.new_line;

      ---- Inform ”process” that the data item has been output.
      shared_data.oready := -shared_data.oready ;
   ----
   end loop ;
----
end output ;
```

Creating shmdefine input files

*Create the **`shmdefine`** input file(s).*

To perform this step, create the **`shmdefine`** input file using a text editor of your choice (see "Shared Regions" on page 1-5).

Specify the `KEY` attribute with a pathname to ensure that a unique identifier for the shared memory segment is obtained and that access to the segment is limited to the cooperating programs (see "Attributes" on page 1-7).

Specify the Fortran `COMMON` block `shared_data` as the variable (see "Variables" on page 1-7).

Create the input file, **`shmdef`**

```
SHARED REGION input_output
   KEY="./generate.c"
   Fortran COMMON shared_data
END SHARED REGION
```

Executing shmdefine

*Execute **`shmdefine`** with the desired options.*

Execute **`shmdefine`** using **`shmdef`** as the input file. Note that the **`-b`** option designates **`shmdef`** as the base name for the object files produced by the utility (see "Command Syntax" on page 1-1). Also, the **`-t`** option instructs the **`shmdefine`** utility to generate output files for use on a PowerMAX OS target.

```
% shmdefine -b shmdef -t pmax shmdef
```

The initialization file (**`shmdef.sm.c`**) and the linker command file (**`shmdef.sm.ld`**) are created.

Compiling the initialization output file

*Compile the initialization output file that is produced by **`shmdefine`**.*

Compile the initialization file by invoking the C compiler. Note that a subsequent listing of your files will include the object file produced by the compiler.

```
% /usr/ccs/bin/ec -c shmdef.sm.c
```

This creates the object file **`shmdef.sm.o`**.

Compiling and linking the source programs

Compile and link the source programs with the ***shmdefine*** *initialization object file and the* ***shmdefine*** *link command output file.*

Refer to the following sections to compile and link the source programs in our example.

Compiling and linking the C program

To compile and link the program to generate the data (**generate.c**), invoke the C compiler, and specify the initialization object file and the link command file.

```
% /usr/ccs/bin/ec -o generate generate.c shmdef.sm.o \
    -Wl,-M shmdef.sm.ld --osversion=4.3 --arch=moto
```

The executable file **generate** is created.

For information specific to the C programming language, refer to the *Concurrent C/C++ Reference Manual* (0890497).

Compiling and linking the Ada programs

To compile and link the programs to initialize the **iready** and **oready** variables and write the results of the computations (**output.a**), create an Ada compilation environment, define the program to be built including the object file and link command file, as shown below.

```
% /usr/ada/bin/a.mkenv -osversion 4.3 -arch moto
% /usr/ada/bin/a.intro output.a
% /usr/ada/bin/a.partition -create active \
     -oset '-ld shmdef.sm.o -M shmdef.sm.ld' output
% /usr/ada/bin/a.partition -create active \
     -oset '-ld shmdef.sm.o -M shmdef.sm.ld' init
% /usr/ada/bin/a.build
```

The executable files **init** and **output** are created.

NOTE

The values specified for the **-osversion** and **-arch** options to **a.mkenv** depend on the architecture and version of PowerMAX OS running on the target system. The values used in the above example may not be correct for the system you are targeting.

Refer to the *MAXAda Reference Manual* (0890516) for further information.

Compiling and linking the Fortran program

To compile and link the program to process the data (**process.f**), invoke the Fortran compiler, and specify the initialization object file and the link command file.

```
% /usr/ccs/bin/xf77 -o process process.f shmdef.sm.o \
     -M shmdef.sm.ld --osversion=4.3 --arch=moto
process.f:
```

The executable file **process** is created.

NOTE

In this example, the cross-compiler running on a Linux host is used. Native compilations on a PowerMAX OS system should invoke **hf77**.

For information specific to Fortran and the use of shared memory, refer to the *hf77 Fortran Reference Manual* (0890240).

Running the programs

You are now ready to run the programs on a PowerMAX OS system. Note that each program performs its operations asynchronously.

```
% ./init
% output &
[1] 5515
% process &
[2] 5526
% generate
result 1 = -2
result 2 = -4
result 3 = -8
result 4 = -16
result 5 = -32
result 6 = -64
result 7 = -128
result 8 = -256
result 9 = -512
result 10 = -1024
[2] + Done                 process &
[1] + Done                 output &
```

The first program to start executing will create the shared memory section and initialize the data items with any compile-time initial values as defined by the source code for that program. Subsequent invocations of any programs sharing the data will not re-initialize the data items unless the shared memory segment was first removed.

Linux Example

This example demonstrates how to enable two Fortran programs to cooperate using shared data.

This example utilizes a feature of Fortran called datapools. Concurrent Fortran supports datapools, but the GNU Fortran compiler does not.

NOTE

If you do not have the Concurrent Fortran compiler, you can still follow this example and substitute common blocks for datapools. At each step, the example text will instruct you as to how to make the appropriate substitutions.

The example consists of a **`work`** program which does a simple numeric calculation and then exports the calculation by writing to the variable `calculation`. The second program, **`report`**, prints the value of `calculation`.

- Creating source programs (see page 1-18)
- Creating shmdefine input files (see page 1-21)
- Compiling the datapool definition file (see page 1-22)
- Executing shmdefine (see page 1-22)
- Compiling the initialization output file (see page 1-22)
- Compiling and linking the source programs (see page 1-23)
- Running the programs (see page 1-23)

Creating source programs

Create source programs.

For our example, create the following Fortran source files using a text editor of your choice:

- **`work.f`** (see "work.f" on page 1-18)
- **`report.f`** (see "report.f" on page 1-19)
- **`pool.dp`** (see "pool.dp" on page 1-20)

NOTE

If you do not have the Concurrent Fortran compiler, uncomment the lines currently commented out in the **`work.f`** and **`report.f`** source files (as indicated in red) and comment the line containing the datapool statement (as indicated in blue). In this case, skip the creation of **`pool.dp`** as well.

work.f

```
      program work
      real*4 calculation
      real*4 data
C     common /blk/ data, calculation
      datapool /pool/ data, calculation
      call shm_init
      data = 1.0
10    continue
      calculation = data * 1.3
      data = calculation
      call sleep(1)
      goto 10
      end
```

report.f

```
      program report
      real*4 calculation
C     real*4 data
C     common /blk/ data, calculation
      datapool /pool/ calculation
      call shm_init
10    continue
      call sleep(1)
      write (6,*) calculation
      goto 10
      end
```

pool.dp

```
real*4   data
real*4   calculation
datapool /pool/ data, calculation
end
```

Creating shmdefine input files

Create the ***shmdefine*** *input file(s).*

To perform this step, create the **shmdefine** input file using a text editor of your choice (see "Shared Regions" on page 1-5).

Create the input file, **shmdefine.input**

```
shared region share
   key = "/tmp/key"
   # fortran common blk size 8
   fortran datapool pool, "pool.o"
end shared region
```

The choice of the filename associated with the `KEY` attribute is arbitrary. It is used to ensure that a unique identifier for the shared memory segment is obtained and that access to the segment is limited to the cooperating programs (see "Attributes" on page 1-7).

The name of the shared region is arbitrarily chosen by the user as well.

NOTE

If you do not have the Concurrent Fortran compiler, uncomment the line:

```
fortran common blk size 8
```

and comment out the line:

```
fortran datapool pool, "pool.o"
```

in **shmdefine.input**.

One advantage of using Fortran datapools is that **shmdefine** can automatically determine the size of the datapool by reading the object file corresponding to the datapool definition file.

For Fortran common blocks, it is the user's responsibility to specify the correct size of the common block in the **shmdefine** input file. In such cases, if you have an object file which describes the entire common block, the script **shmdefine.size** (shipped with the **shmdefine** utility) can help you in obtaining the size.

NOTE

shmdefine.size works only on Linux ELF 32-bit LSB relocatable object files.

Compiling the datapool definition file

NOTE

If you do not have the Concurrent Fortran compiler, skip this step.

Compile the datapool definition file using the following command.

```
% /usr/ccs/bin/cf77 -c -g pool.dp
```

Executing shmdefine

*Execute **shmdefine** with the desired options.*

Execute **shmdefine** using **shmdefine.input** as the input file. Note that the **-b** option designates **shmdef** as the base name for the object files produced by the utility (see “Command Syntax” on page 1-1). Also, the **-t** option instructs the **shmdefine** utility to generate output files for use on a Linux target.

```
% /usr/bin/shmdefine -b shmdef -t linux shmdefine.input
```

NOTE

If you do not have the Concurrent Fortran compiler, add the **--gnu-f77** option to the above command.

The initialization file (**shmdef.sm.c**) and the linker command file (**shmdef.sm.ld**) are created.

Compiling the initialization output file

*Compile the initialization output file that is produced by **shmdefine**.*

Compile the initialization file by invoking the C compiler. Note that a subsequent listing of your files will include the object file produced by the compiler.

```
% /usr/bin/gcc -c shmdef.sm.c
```

This creates the object file **shmdef.sm.o**.

When linking programs that use **shmdefine**, you must specify the linker configuration file generated by **shmdefine** as well as the compiled form of the C source file generated by **shmdefine**.

Compiling and linking the source programs

Compile and link the source programs with the ***shmdefine*** *initialization object file and the* ***shmdefine*** *link command output file.*

Compile and link the work and report programs by invoking Concurrent Fortran compiler in the following manner:

```
% /usr/ccs/bin/cf77 -g -o work work.f shmdef.sm.ld \
                                   shmdef.sm.o pool.o
% /usr/ccs/bin/cf77 -g -o report report.f shmdef.sm.ld \
                                   shmdef.sm.o pool.o
```

If you do not have the Concurrent Fortran compiler, use the following invocations of the GNU Fortran compiler:

```
% /usr/bin/g77 -g -o work work.f shmdef.sm.ld shmdef.sm.o
% /usr/bin/g77 -g -o report report.f shmdef.sm.ld shmdef.sm.o
```

Running the programs

Invoke the **work** program in the background. It updates the variable `calculation` in the datapool `pool` once a second.

```
% ./work &
```

Invoke the **report** program. It prints the value of `calculation` once a second.

```
% ./report
2.8561
3.7129
4.8268
6.2749
8.1573
...
```

NOTE

The **report** program only refers to the variable `calculation`; it does not need to have a complete description of the datapool. This is the major advantage of datapools over `COMMON` blocks - the program only needs to reference the variables with which it is concerned.

Terminate the **report** program using the Ctrl-c keyboard sequence.

Terminate the **work** program by typing:

```
% fg
```

and then using the Ctrl-c keyboard sequence.

Spine for 1/2” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

shmdefine

Quick Reference

0898010