Book 2—Cookbook

Part A:

Developing TriMedia Applications

TriMedia

Version 2.1

Table of Contents

Chapter 1 Compiling TriMedia Applications

Introduction 6

Build and Execution Hosts 8
Build Hosts 8
Execution Hosts 8

Using tmcc to Compile TriMedia Applications 10
Invoking tmcc 10
Using tmcc Options 11
Specifying Execution Hosts 11
Compiling Multiple Files 12
Specifying Endianness 13
Predefined Macros 13

Creating Makefiles 14

Creating pSOS Makefiles 15
Simple pSOS Application Makefile 15
Porting This Makefile to nmake 16
Linking With Other pSOS Libraries 17
Using the pSOS Monitor 18

Running TriMedia Applications 19
Running TriMedia Applications with tmgmon 19
Dumping the Trace Buffer 19
Example 20
Running TriMedia Applications with tmrun 21
Running TriMedia Applications with tmmon 21
Running TriMedia Applications with tmdbg 21
Running TriMedia Applications with tmmprun 22

Chapter 2 Programming With pSOS

Introduction 24
A pSOS Beginning 24
The Root Function 24
Communication Using Semaphores 25

Communication Using Asynchronous Signals 26
A pSOS Ending 26

A pSOS+™ Based Multiprocessor Example 27
Starting Development 27
Number of Executables to Build 27
The Root Function 28
Buffer and Packet Management, Caching Issues 29
DMA Transfer 30

Chapter 3 Using the Dynamic Loader on TriMedia

Introduction 34

Dynamic Loading Basics 34

Dynamic Loader Example 35
Starting Development 35
The Root Function 35
The Application Shell 36
Running dynamic_loader_shell 37

Chapter 1

Compiling TriMedia Applications

Topic	Page
Introduction	6
Build and Execution Hosts	8
Using tmcc to Compile TriMedia Applications	10
Creating Makefiles	14
Creating pSOS Makefiles	15
Running TriMedia Applications	19

Introduction

Traditionally, real-time Digital Signal Processor (DSP) and multimedia applications have been primarily implemented in assembly language. The TriMedia hardware architecture enables you to implement applications not only in assembly language, but also in high-level languages such as C and C++.

The TriMedia Compilation System (TCS) translates C and C++ programs and generates code for a machine in the TriMedia architecture family. This cookbook addresses issues related to developing applications for TriMedia in C or C++.

The TCS translates C and C++ programs and generates machine code for the TriMedia architecture family. The TriMedia **tmcc** (**tmCC** for C++) compiler driver controls program compilation and linking for the TriMedia processor. Figure 1 shows the stages in the TriMedia compilation and simulation system, as well as the information flow during the stages.

The **tmcc** compiler driver provides a natural command-line interface that makes it unnecessary for most users to understand the details of the TriMedia compiler. Some features of the **tmcc** compiler driver are useful for system software developers who must test drop-in replacements for TCS tools, while other features are useful for application developers and system architects.

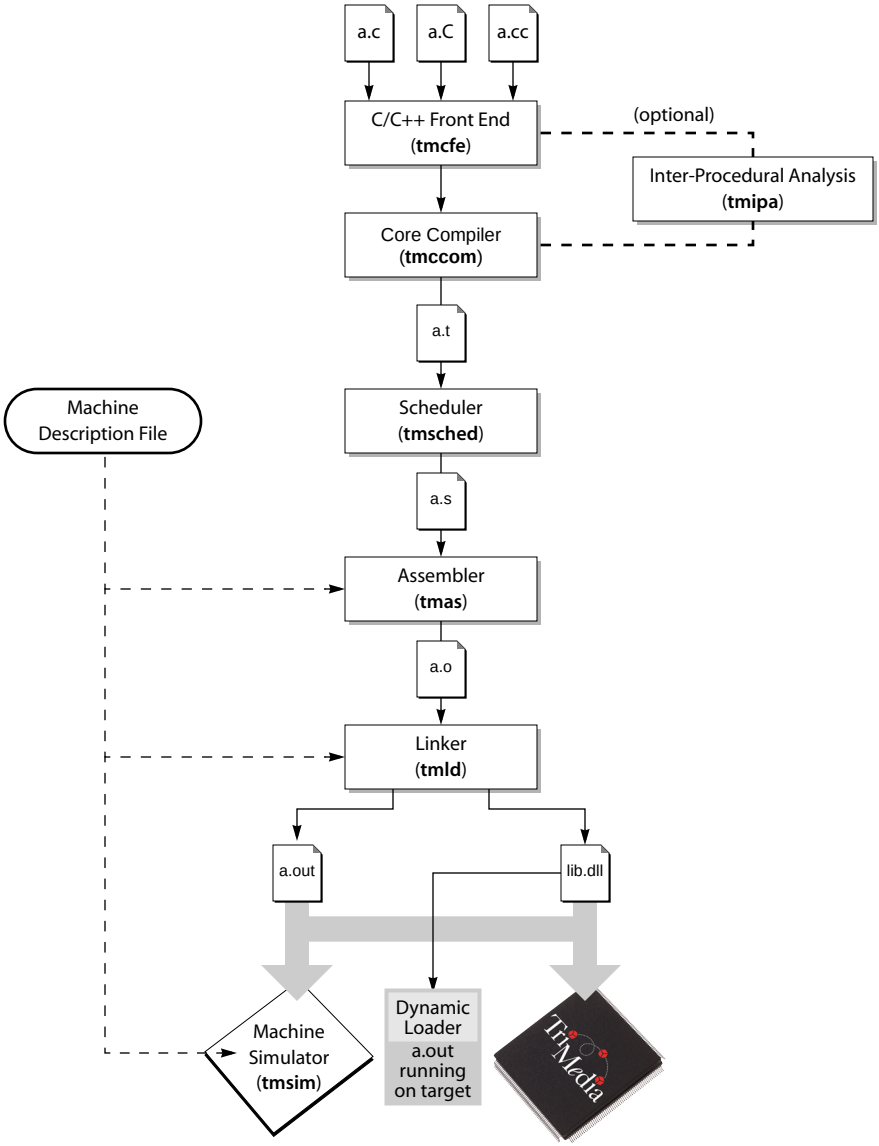


Figure 1 TriMedia Compilation and Simulation System

Build and Execution Hosts

The following two types of platforms use TriMedia applications:

- Build hosts
- Execution hosts

Build Hosts

You use build hosts to develop and compile TriMedia applications. (You must install the TCS first.) Following is a list of the build hosts:

- Solaris
- SunOS
- HP-UX
- Mac OS
- Windows 95
- Windows NT

Because the TCS works in the same way on each build host, selecting a host is a matter of personal preference. For example, some developers prefer using a UNIX-based host (Solaris, SunOS, and HP-UX) because of the following:

- Availability
- Higher performance
- Extensive experience using UNIX-based hosts

On the other hand, other developers prefer using personal computers (PCs) for building and running TriMedia applications. The choice of host is completely up to you.

Execution Hosts

You use execution (host) hosts to run TriMedia applications. Following is a list of the execution hosts:

- Mac OS
- Windows 95
- Windows NT
- Windows CE

Note
Although you can use the Windows 95 host for both building and running TriMedia applications, it is sometimes useful to use different machines for building and running. For example, you might build on Windows NT and run on Windows 95. This helps you avoid problems with the building environment if the TriMedia application you are trying to build crashes.

Using tmcc to Compile TriMedia Applications

This section describes how to use the TriMedia **tmcc** compiler driver.

Invoking tmcc

You can invoke the **tmcc** (**tm**CC for C++) compiler driver using either

```
tmcc [ <option> ... ] <file> ...
```

or

```
tmCC [ <option> ... ] <file> ...
```

The command line can specify options that affect the operation of **tmcc** and must specify at least one file that **tmcc** processes. Each file argument must have one of the known extensions listed below. In keeping with standard C usage, **tmcc** passes each unrecognized argument to the **tmld** loader directly. Refer to the following table.

Extension	Description
.c	C source file.
.C, .cc, or .cpp	C++ source file (Windows 95 does not have case distinction. The extension .cc or .cpp indicates a C++ program.)
.i	Preprocessed C source file. Output of the C cpp preprocessor.
.t	Intermediate representation (decision trees.) Output of the tmccom core compiler.
.s	Assembly code. Output of the tmsched instruction scheduler.
.o	Unlinked object module. Output of the tmas assembler.
.out	Linked executable. Output of the tmld linker.
a.dyn	A linked executable designed to be loaded dynamically by a general shell. Created by tmld.
.a	Library file. Output of the tmar librarian.
.dll	A library linked to be loaded dynamically. Created by tmld.

The only difference between **tmcc** and **tm**CC is that **tm**CC assumes that compilation involves objects generated from C++ sources. Thus, **tm**CC always links with the standard C++ (libC++.a) library.

The **tmcc** compiler driver also enables you to pass command-line arguments to specified compilation stages, as described in the next section.

Using tmcc Options

The **tmcc** compiler driver enables you to pass extra arguments to compilation stages directly by specifying the name of the desired stage, followed by the desired arguments with the special terminator "--". For example, the command

```
tmcc -tmccom -xunvdl=0 -- foo.c
```

compiles the **foo.c** program and adds the **-xunvdl=0** argument to the options that **tmcc** normally passes to **tmccom**. Similarly, you can pass arguments to other compilation and linkage phases using the **-tmcfe**, **-tmccom**, **-tmsched**, **-tmas**, or **-tmld** options.

For more information about the **tmcc** options, see the **tmcc** man page.

Specifying Execution Hosts

In almost all circumstances, if you have a TriMedia board, you define the run time host as the host computer of your board. However, you can get information out of the simulator that you can't get out of the chip (for example, detailed analysis performance that enables you to observe cache behavior), so you may sometimes want to specify the simulator **tmsim** as execution host.

The configuration file defines a **HOST_DEFAULT** default host. It also contains host-specific sections, each starting with **HOST=*host*** and ending with **HOST_END**. You can specify an execution host with the **-host** option to **tmcc**, which allows host-specific compilation. The syntax is as follows:

```
tmcc -host host foo.c
```

This builds an executable suitable for the specified **host** (for example, Win95 or **tmsim**) by using the **host**-specific parts of the **tmconfig** configuration file.

Compiling TriMedia Applications to Run on the Simulator

To compile the sample program hello.c (it prints the message "hello, TriMedia world" on the screen), type the following:

```
tmcc -o hello hello.c
```

When the file compiles successfully, use the **tmsim** command to run the resulting program using the TriMedia simulator (**tmsim**) as follows:

```
tmsim hello
```

The following message appears:

```
hello, TriMedia world
```

You can also compile C++ applications for the simulator in the same way.

```
tmcc -o hello2 hello2.cc
```

When the file compiles successfully, run the resulting executable file with **tmsim.**

```
tmsim hello2
```

The following message appears:

```
Hello, TriMedia C++ World!
```

Compiling TriMedia Applications to Run on the Chip

To compile for the chip, you must specify the execution host that contains the TriMedia chip. You do this using the **-host** option (**Win95** for a Windows 95 PC as shown in the following example:

```
tmcc -o hello -host Win95 hello.c
```

To run the resulting executable file, refer to "Running TriMedia Applications" on page 19 for more information.

When you specify **Win95** as the execution host, **tmcc** selects various options from the **tmconfig** file. The **tmconfig** file sets the default endianness to **-el** (little endian), and adds a number of libraries that are Windows 95-specific.

Compiling Multiple Files

The following command compiles two files (ave1.c and ave2.c) and produces an executable ave, assuming no errors occur in any of the compilation stages:

```
tmcc -o ave ave1.c ave2.c
```

The **tmcc** compiler driver expects filenames to have one of the extensions listed on page 10. Its actions depend on the extension and the driver options specified in the command line. For example, the command

```
tmcc -o ave ave1.c ave2.i ave3.t ave4.s ave5.o
```

causes **tmcc** to

- Preprocess, compile, schedule, and assemble ave1.c to produce ave1.o
- Compile, schedule, and assemble ave2.i to produce ave2.o
- Schedule and assemble ave3.t to produce ave3.o
- Assemble ave4.s to produce ave4.o
- Link the five object files (ave1.o, ave2.o, ave3.o, ave4.o, and ave5.o) to produce the executable ave.

The **-D** option defines preprocessor macros as follows:

```
tmcc -DMAX_LEN=1024 -DFOO -o ave ave1.c ave2.c
```

In the following example, the first command line compiles a program with profiling code inserted (using the **-p** option). The second line simulates the resulting program **a.out** using **tmsim**, which generates an execution profile in the file **dtprof.out**. The third recompiles the program using the profile information (using the **-r** option).

```
tmcc -p ave1.c ave2.c
tmsim a.out
tmcc -r -o ave ave1.c ave2.c
```


The following example is identical to the previous example, except that the second compilation uses the profile information to perform grafting (using the **-G** option):

```
tmcc -p ave1.c ave2.c
tmsim a.out
tmcc -G -o ave ave1.c ave2.c
```

Specifying Endianness

The TriMedia processor supports either big endian or little endian byte ordering, depending on the BSX bit in the PCSW. (See the appropriate TriMedia data book for details.) The default is big endian. You can change the default by editing the configuration file; you can override the default with the **-eb** or **-el** command-line option. In addition, you can override the default endianness with **tmcc**'s **-host** option (**-host Win95** uses **-el** by default and **-host MacOS** uses **-eb** by default).

Predefined Macros

The **tmcc** compiler driver automatically defines several macros when it invokes **tmcfe** (C/C++ programs). The following macros are always defined:

Macro	Description
__TCS__	Defined during source file conditionalization to indicate source code specific to the TCS.
__STDC__	Defined to indicate compliance with the ANSI/ISO C Standard.
__BIG_ENDIAN__	Defined when compiling in big endian mode .
__LITTLE_ENDIAN__	Defined when compiling in little endian mode.
__TCS__host__	Defined to indicate compilation for the given host ***host.***
__TCS__target__	Defined to indicate compilation for the given host ***target.***
__cplusplus	Defined to indicate C++ compilation.
__TMSCHED__	Defined by **tmcc** with the **-x** option when preprocessing a ".t" source file.
__TMAS__	Defined by **tmcc** with the **-x** option when preprocessing a ".s" source file.

Note
You can specify additional predefined macros for C source compilation on the **CPP_ARGS** line of the tmconfig configuration file. You can specify additional predefined macros for C++ source compilation on the **TMCFE_ARGS** line.

Creating Makefiles

A makefile is a useful way to organize information about programs, especially if you have complicated programs. It enables you to include device libraries and define options that you use frequently in your program without having to remember this information every time you recompile your programs. Following is an example of a standard UNIX makefile for compiling the hello.c sample program:

```
CC=tmcc or CC=$(TCS)/bin/tmcc
CFLAGS=-host Win95

hello.out:      hello.o
                $(CC) $(CFLAGS) -o $@ hello.o

hello.o:        hello.c
                $(CC) $(CFLAGS) -c -o $@ hello.c
```

Note
The **$@** symbols represent the program that you are trying to build (in this case, hello.out or hello.o). This makefile runs transparently on a Windows 95 platform using Microsoft's NMAKE.

To run this file, type **make** and press Enter.

```
make
tmcc -host Win95 -c -o hello.o hello.c
tmcc -host Win95 -o hello hello.o
```

IMPORTANT
Makefiles might not necessarily be portable across different build hosts. For example, UNIX makefiles use forward slashes in path information, while makefile utilities on the Windows platform, such as nmake and gnumake, use back slashes (in addition to other differences). In the case of simple makefiles, you might be able to easily modify makefiles to work on one platform or the other. However, when dealing with long and complicated makefiles, Philips highly recommends that you use utilities such as the Mortice Kern Systems (MKS) toolkit. This third-party utility adds UNIX-compatible commands (including the make command) to the PC's command line and recognizes forward slashes and backward slashes equally. This enables you to run UNIX-based makefiles on the PC.

```
#
#   NMAKE compatible makefile for myecho
#
TCS = c:\tcs1.1
CC = $(TCS)\bin\tmcc
CFLAGS = -host Win95

myecho.out : myecho.o
      $(CC) $(CFLAGS) -o $@ myecho.o
myecho.o : myecho.c
      $(CC)  $(CFLAGS) -c -o $@ myecho.c
clean :
      del myecho.o
      del myecho.out
```


Creating pSOS Makefiles

Simple pSOS Application Makefile

The following is an example of a simple pSOS application makefile for use in the Unix-like make environment, including the MKS toolkit on Windows. Minor changes can be made to use it with Microsoft's NMAKE.

```
# Fill in these appropriately for your application and host configuration
# HOST: Win95, WinNT, MacOS, tmsim, nohost
# ENDIAN: el, eb

TCS         = /usr/local/tcs
HOST        = Win95
ENDIAN      = el
APPLICATION = a.out
OBJECTS     = root.o drv_conf.o
target: $(APPLICATION)

# You normally should not need to change the following

PSOS_SYSTEM = $(TCS)/OS/pSOS/pSOSystem
PSOS_DEFS   = -DSC_PSOS=YES -DSC_PSOSM=NO -DSC_PNA=NO -DSC_PPP=NO
CC          = $(TCS)/bin/tmcc -host $(HOST) -$(ENDIAN) $(PSOS_DEFS)
LD          = $(TCS)/bin/tmld
AR          = $(TCS)/bin/tmar
CINCS       = -I. -I$(PSOS_SYSTEM)/include
CFLAGS      =
LDFLAGS     = -bremoveunusedcode -bcompact -bfoldcode

$(APPLICATION): bsp.a $(OBJECTS) Makefile
        @ echo "Linking $(APPLICATION)"
        @ $(CC) \
                $(OBJECTS) $(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).o bsp.a \
                $(LDFLAGS) $(CFLAGS) -o $(APPLICATION)

bsp.a:
        @ make -f $(PSOS_SYSTEM)/configs/Makefile \
                PSOS_SYSTEM="$(PSOS_SYSTEM)" \
                AR="$(AR)" CC="$(CC)" CFLAGS="$(CFLAGS)"

%o: %c
        @ echo "Compiling $(*)c"
        @ $(CC) $(CFLAGS) $(CINCS) -c $(*)c -o $@

clean:
        rm -fr $(APPLICATION) *.o bsp.a
```

The macros that should be customized for a specified build environment are **TCS**, **HOST**, **ENDIAN**, **APPLICATION**, and **OBJECTS**. **CINCS**, **CFLAGS**, and **LDFLAGS** can also be customized, but it is not necessary.

This makefile assumes the TCS compiler tools are located at "/usr/local/tcs." When using MKS, it should be like "C:/TriMedia/bin", with forward slashes (/). The makefile compiles local objects, like root.c. Then it builds a pSOS board support package as a library and this is linked together with the appropriate version of the standard pSOS library to create

the executable program. At least two components of the pSOS BSP might be overridden with application-specific versions. These are sys_conf.h, and drv_conf.c. Its resulting executable is called "a.out" in the local directory.

Porting This Makefile to nmake

To use this makefile with Microsoft's nmake, follow the step listed below (also found in $(TCS)/examples/psos/psos_demo1/Makefile.simple).

1. Copy this file to Makefile.win
2. In $(PSOS_SYSTEM)/config, copy Makefile to Makefile.win
3. Replace all forward slashes (/) with back slashes (\) in both files
4. Change the default rule to

```
{$(SRC)\}.c.o:
        @ echo "Compiling $<"
        $(ECHO_OPTION) $(CC) -c $(CFLAGS) $(CINCS) -o $@ $<
```

   Change make command for target bsp.a below to

```
@ nmake /f $(PSOS_SYSTEM)\configs\Makefile.win
PSOS_SYSTEM="$(PSOS_SYSTEM)" APPDIR="." AR="$(AR)" CC="$(CC)"
CFLAGS="$(CFLAGS)"
```

5. Change object file rule to

```
.c.o:
        @ echo "Compiling $*.c"
        @ $(CC) $(CFLAGS) $(CINCS) -c $*.c -o $@
```

6. Make sure you take out all back slashes (\) for line separation.
7. Invoke this makefile by typing at a MS-DOS prompt:

```
nmake /f Makefile.win
```

The resulting version of the above makefile is listed below.

```
# Fill in these appropriately for your application and host configuration
# HOST: Win95, WinNT, MacOS, tmsim, nohost
# ENDIAN: el, eb

TCS         = C:\TriMedia\bin
HOST        = Win95
ENDIAN      = el
APPLICATION = a.out
OBJECTS     = root.o drv_conf.o
target: $(APPLICATION)

# You normally should not need to change the following

PSOS_SYSTEM = $(TCS)\OS\pSOS\pSOSystem
PSOS_DEFS   = -DSC_PSOS=YES -DSC_PSOSM=NO -DSC_PNA=NO -DSC_PPP=NO
CC          = $(TCS)\bin\tmcc -host $(HOST) -$(ENDIAN) $(PSOS_DEFS)
LD          = $(TCS)\bin\tmld
AR          = $(TCS)\bin\tmar
CINCS       = -I. -I$(PSOS_SYSTEM)\include
```



```
CFLAGS      =
LDFLAGS     = -bremoveunusedcode -bcompact -bfoldcode

$(APPLICATION): bsp.a $(OBJECTS) Makefile
        @ echo "Linking $(APPLICATION)"
        @ $(CC) $(OBJECTS) $(PSOS_SYSTEM)\sys\os\psos_tm_$(ENDIAN).o bsp.a
$(LDFLAGS) $(CFLAGS) -o $(APPLICATION)

bsp.a:
        @ nmake /f $(PSOS_SYSTEM)\configs\Makefile.win
PSOS_SYSTEM="$(PSOS_SYSTEM)" APPDIR="." AR="$(AR)" CC="$(CC)"
CFLAGS="$(CFLAGS)"

.c.o:
        @ echo "Compiling $*.c"
        @ $(CC) $(CFLAGS) $(CINCS) -c $*.c -o $@

clean:
        rm -fr $(APPLICATION) *.o bsp.a
```

Notice that Steps 2, 3, 4, and 7 are also to be applied to the Makefile in $(PSOS_SYSTEM)/config. The resulting makefile there should also be called Makefile.win (see step 2. above).

Linking With Other pSOS Libraries

Note that only with care can this makefile be made to link with special pSOS libraries, such as pSOS+m, dynamic linking, and pSOS networking modules (pNA, PPP). For such advanced compilation, the comprehensive makefile in $(TCS)/examples/psos/psos_demo1/Makefile should be used. Instructions to use that makefile are found in it. Below are a few instructions on how to change this makefile to link with pSOS+m, dynamic linking, and pSOS networking libraries.

To use pSOS+m, switch on pSOS+m and switch off pSOS in the definition of **PSOS_DEFS** (**-DSC_PSOS=NO -DSC_PSOSM=YES**). Then change psos_tm_$(ENDIAN).o to psosm_tm_$(ENDIAN).o, under the rule for $(APPLICATION).

To use the pSOS library compiled for dynamic linking, replace $(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).o with -bimmediate $(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).dll. To use pSOS+m with dynamic linking, follow the steps above for pSOS+m after applying the steps for dynamic linking.

To use pNA, switch on the PNA flag with -DSC_PNA=YES in the definition of **PSOS_DEFS**, and add $(PSOS_SYSTEM)/sys/os/pna_tm_$(ENDIAN).o after $(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).o. To use pNA with dynamic linking, instead of above, add -bimmediate $(PSOS_SYSTEM)/sys/os/pna_tm_$(ENDIAN).dll.

Apply the same steps for PPP as for pNA.

Table 1 The Usage of pSOS in the Sample Makefile

	PSOS_DEFS	$(APPLICATION)
pSOS	-DSC_PSOS=YES -DSC_PSOSM=NO	$(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).o
pSOS+m	-DSC_PSOS=NO -DSC_PSOSM=YES	$(PSOS_SYSTEM)/sys/os/psosm_tm_$(ENDIAN).o
dll, pSOS	-DSC_PSOS=YES -DSC_PSOSM=NO	-bimmediate$(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).dll
dll,pSOS+m	-DSC_PSOS=NO -DSC_PSOSM=YES	-bimmediate$(PSOS_SYSTEM)/sys/os/psosm_tm_$(ENDIAN).dll
pNA	add -DSC_PNA=YES	add $(PSOS_SYSTEM)/sys/os/pna_tm_$(ENDIAN).o
PPP	add -DSC_PPP=YES	add $(PSOS_SYSTEM)/sys/os/ppp_tm_$(ENDIAN).o
pNA, dll	add -DSC_PNA=YES	add -bimmediate $(PSOS_SYSTEM)/sys/os/pna_tm_$(ENDIAN).dll
PPP, dll	add -DSC_PPP=YES	add -bimmediate $(PSOS_SYSTEM)/sys/os/ppp_tm_$(ENDIAN).dll

Using the pSOS Monitor

To use compile this makefile with the pSOS monitor for debugging in **tmdbg**, follow the steps below.

1. Add **-g** to CFLAGS.
2. Add $(TCS)/lib/$(ENDIAN)/psosmon.o to the link line of your application:

```
  @ $(CC) \
        $(OBJECTS) $(PSOS_SYSTEM)/sys/os/psos_tm_$(ENDIAN).o bsp.a \
        $(TCS)/lib/$(ENDIAN)/psosmon.o $(LDFLAGS) $(CFLAGS) -o
$(APPLICATION)
```

3. Remove linker optimizations in **LDFLAGS**.

Running TriMedia Applications

You can run TriMedia applications on PC hosts (running Windows 95 or Windows NT) with any of the following tools:

- **tmgmon**
- **tmrun**
- **tmmon**
- **tmdbg**
- **tmmprun**

Running TriMedia Applications with tmgmon

The **tmgmon** tool is a GUI-based Win32 application (built on top of **tmmon**) that uses the TriMedia Manager Host Application Programming Interface (API). It provides an interactive user interface for downloading and running TriMedia executables on the TriMedia processor. You can access all options by selecting the appropriate option from the window. Scrollable views are provided for the trace and memory window to aid in debugging.

Note
By default, tmgmon will switch to the directory of the program you are trying to run. So if your program accesses any data files, they are relative to the program's executable directory. However, you can disable this option and can specify a different working directory.

Dumping the Trace Buffer

The Win95 execution host provides the DP function to be used for real-time debugging. This is described in Part C of Book 4, *Software Tools*. The **tmgmon** tool enables you to dump the DP buffer. The Tracep button at the lower left of the **tmgmon** TriMedia Monitor window initiates a dump.

If the DP buffer is small, you can dump the DP buffer to the scrollable buffer on screen. If the DP buffer is large (greater than 64K), the on-screen buffer is too small and Philips recommends dumping the DP buffer to a file. You can select file output by typing a filename in the Trace File field of the TriMedia Monitor window and checking the box to its right.

Example

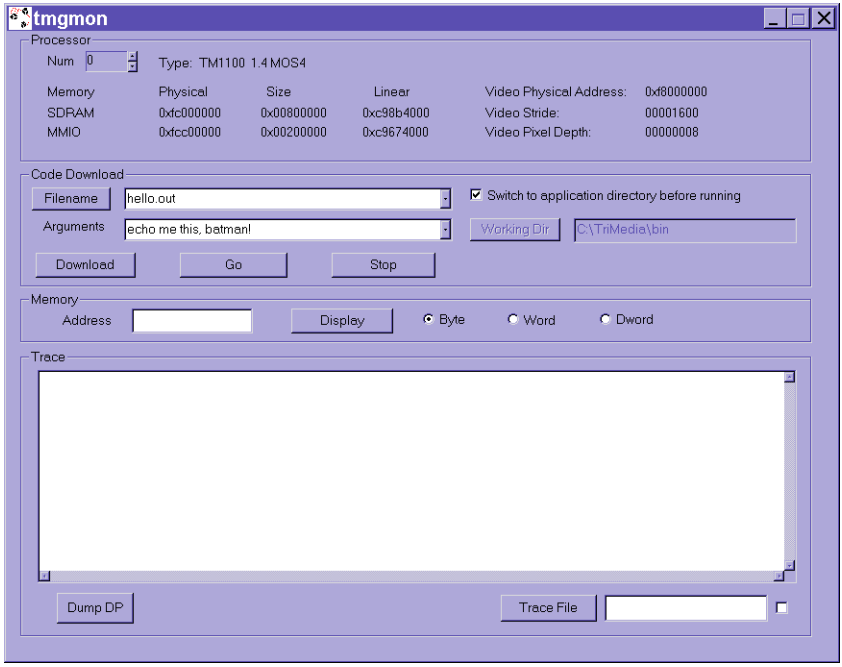
The following steps show you how to use **tmgmon** to run TriMedia programs and dump the DP buffer:

1. Compile the program to run on Windows 95.

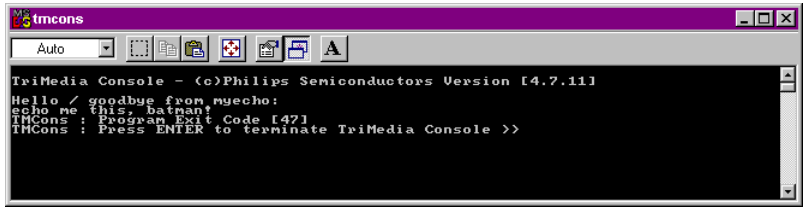
```
#include <stdio.h>
#include <tmlib\dprintf.h>
main( int argc, char **argv ){
   int   i;

   DP_START( 64000, Null );
   DP(("Debug Printf from myecho\n"));
   printf( "\nhello / goodbye from myecho: \n" );
   for( i=1; i<argc; i++ )
      printf("%s ", argv[i]);
   return (0x47);
}
$ tmcc -o myecho -host Win95 myecho.c
```

2. Enter the name of the program in the Filename field.
3. Enter arguments in the Arguments field.
4. Click Go.
5. Click Tracep.



The “Hello, goodbye from myecho” message appears on the screen.



You can also use **tmgmon** to pass arguments. For example, to run the sample program "myecho," type myecho in the Filename field, enter the arguments in the Arguments field, and specify the standard output file (optional) in the Stdout field.

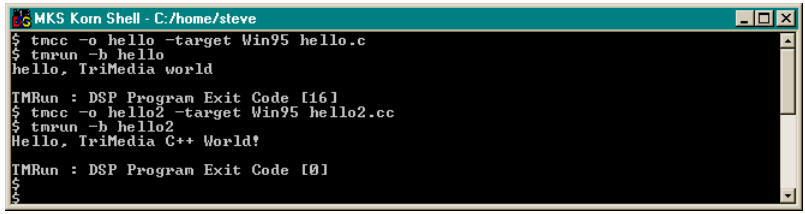
Running TriMedia Applications with tmrun

Use **tmrun** to download programs and run them on the TriMedia processor. This is a Win32 console application that enables you to run programs in batch mode.

For example, to run the hello program, type the following:

```
tmrun -b hello
```

Running TriMedia Applications with tmmon



The **tmmon** tool is a Win32 console mode application that provides a command-based interface for executing programs on the TriMedia processor. It performs its functions through calls to the documented TriMedia Manager interface.

When the program "myecho" compiles successfully, launch **tmmon** and load the program using the **ld** command (type the arguments to pass after the program name).

Running TriMedia Applications with tmdbg

You can run TriMedia applications using the **tmdbg** TriMedia debugger. Refer to Part C of Book 4, *Software Tools* for more information about using **tmdbg**.

Running TriMedia Applications with tmmprun

The **tmmprun** application allows a multiprocessor application to be downloaded to a set of IREF boards. This is a Win32 console application that enables you to run programs in batch mode.

For example, to run the vivot demo application on the first processor and fplay on a second one, type the following:

```
tmmprun -exec vivot -exec fplay
```


Chapter 2

Programming With pSOS

Topic	Page
Introduction	24
A pSOS+™ Based Multiprocessor Example	27

Introduction

This section describes an example of a simple pSOS application that creates pSOS tasks then uses semaphores and asynchronous signals for communication between the tasks. The example can be found in the TCS release, in the following directory:

$TCS/examples/psos/psos_demo1

A pSOS Beginning

This example demonstrates the communication between the root task and two other tasks via semaphores and asynchronous signals.

Note
The pSOS system timer must be started with a call to **de_init** in order to use timed events, timeslicing, or the system clock. The root task will then print "Hello, world."

The Root Function

The root task creates two tasks, **task1** and **task2**, and two semaphores, **sem_enter** and **sem_exit**. In this file, **sem_enter** and **sem_exit** are stored as global variables, so that the other functions can use them without first having to do **sm_ident** to get the semaphore IDs. The two semaphores are initially set to 1 and decremented to 0 immediately by the root task. Then **task1** and **task2** are started, which will produce the outputs "aaaaaaa" from **task1** and "catcher active" from **task2**.

```
void root(void){
   void *dummy;
   UInt32 rc, ioretval, iopb[4];
   int    i;
   UInt32 task1, task2;

/* Start the pSOS system timer. This is almost always necessary, since
 * otherwise it is not possible to use timed events and timeslicing,
 * or the system clock */
   de_init( DEV_TIMER, 0, &ioretval, &dummy );

   printf( "Hello, world\n");
   t_create( "aaaa",
             4,
             10000,
             10000,
             0,
             &task1
             );
   t_create( "catc",
             100,
             10000,
             10000,
             0,
             &task2
             );
```



```
   sm_create( "semp",
              1,
              SM_PRIOR,
              &sem_enter
              );
   sm_create( "semv",
              1,
              SM_PRIOR,
              &sem_exit
              );
   sm_p( sem_exit,  SM_WAIT, 0 );
   sm_p( sem_enter, SM_WAIT, 0 );

   t_start( task1, T_PREEMPT | T_TSLICE | T_ASR | T_ISR, aaaa, 0 );
   t_start( task2, T_PREEMPT | T_TSLICE | T_ASR | T_ISR, catc, 0 );

   for( i=1; i<10; i++ ){
      sm_p( sem_enter, SM_WAIT, 0 );
      sm_v( sem_exit );
      printf( "TOKEN RECEIVED\n");
      as_send( task1, 1 );
      as_send( task2, 1 );
   }

   sm_delete( sem_enter );
   sm_delete( sem_exit  );

   printf( "Goodbye, world\n");

   t_suspend( 0L );
}
```

Communication Using Semaphores

The root task and **task1** (“aaaa”) will toggle back and forth ten times, as task1 increments **sem_enter** and decrements **sem_exit**, and as the root task decrements **sem_enter** and increments **sem_exit**. **task1** prints “TOKEN SENT” and the root task prints “TOKEN RECEIVED” in each iteration.

```
void aaaa(){
   int err;

   printf( "aaaaaaa\n" );
   as_catch( handler, T_PREEMPT | T_TSLICE | T_ASR | T_ISR );

   do{
      err  = sm_v( sem_enter );
      err |= sm_p( sem_exit,   SM_WAIT, 0 );
      printf( "TOKEN SENT\n" );
   } while( !err );

   printf( "bbbbbbb\n" );
   _psos_exit( 0 );
}
```

Communication Using Asynchronous Signals

At the same time, the root task communicates with both tasks, “aaaa” and “catch” via asynchronous signals. At each iteration of the above loop, the root task sends an asynchronous signal to each task, which is caught by **handler**. The handler prints “BOEM” and exits via **as_return**.

```
void handler(){
   fprintf( stderr, "******************BOEM\n" );
   as_return();
}
void catc(){
   printf( "catcher active\n" );
   as_catch( handler, T_PREEMPT | T_TSLICE | T_ASR | T_ISR );
   t_suspend( 0L );
}
```

A pSOS Ending

This demo ends when the root tasks comes out of the for loop and deletes the two semaphores. **task1** (“aaaa”) then get errors accessing **sem_enter** and **sem_exit**, and also exits its do-while loop, indicated by its output “bbbbbbb”. After printing “Goodbye, world,” the root task suspends itself, and **task1** finishes the demo by calling **_psos_exit**, so that pSOS will kill all tasks and exit.

A pSOS+™ Based Multiprocessor Example

This section will describe an example of a pSOS+™ based multiprocessor application that uses pSOS queues and DMA for passing data streams between nodes. The example can be found in the TCS release, in the following directory:

$TCS/examples/multiprocessing/data_streamer

The global structure of the application will be as follows. One of the nodes (number 0) will serve as a producer of a stream of fixed size packets. All other nodes will consume the packets that they can get from this stream. Hence, the application can be run with two or more processors. Figure 2 shows an *n*-node configuration.

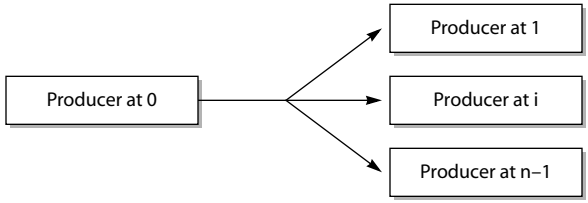


Figure 2 N-Node Configuration

Starting Development

To start development, you must first set up a pSOS+™ application. This is started by copying directory $(TCS)/examples/psos/psos_demo1 into a new development directory.

First, the Makefile is adapted to point to the used TCS installation by changing the TCS macro: TCS = /t/qasoft/build/SunOS.

Next, the desired application name and multiprocessor pSOS are selected by setting:

```
APPLICATION = data_streamer.out
PSOS        = psosm
```

Finally, a DMA transfer function is placed into a separate C file called *transfer.c*. Hence, the corresponding object name is added to the **OBJECTS** macro so that it can be compiled and linked in building the application.

```
OBJECTS= \
$(OBJDIR)/root.o \
$(OBJDIR)/drv_conf.o \
$(OBJDIR)/transfer.o
```

Number of Executables to Build

Although the multiprocessor data streamer application can be run with an arbitrary number of processors, it is not necessary to create more than two executables. We need one executable defining the producer and one generic executable defining all of the con-

sumers. In this two-executable configuration, we would be able to start execution by loading the producer at node 0 and copies of the consumer at all other nodes.

The following example shows the **tmmprun** command that starts a 3-node system with two such executables. This command allocates three TriMedia processors (defined by the number of **-exec** options numbers them 0, 1 and 2), and starts them with the specified executables:

```
tmmprun -exec producer.out -exec consumer.out -exec consumer.out
```

Note
The node numbering provided by **tmmprun** is only a logical numbering: producer.out runs at node 0 only because it was named in the first exec option. Similarly, the two consumers run at nodes 1 and 2.

In this example, however, only one executable will be developed. This executable will make use of the global variable **_node_number**, which is set by **tmmprun** in the downloaded executable for each of the nodes, in order to hold its own logical node number.

Based on the node number, the executable will configure itself as the producer, or as one of the consumers. Note that the advantage of this decision is that only one executable need be maintained, but at the cost of some redundant code on each node (producer nodes will have unused consumer code loaded, and vice versa). The **tmmprun** command for one generic executable will be as follows:

```
tmmprun -exec data_streamer.out  -exec data_streamer.out
        -exec data_streamer.out
```

The Root Function

The decisions in the previous section will shape the root function as listed below. Only one task is needed per node, so the root function does not create tasks. Instead, the root task itself will do the producing/consuming work (note that one copy of pSOS including the root task will be started for each of the nodes).

For communication and synchronization, two pSOS queues are needed. One queue is filled with produced packets by node 0; all other nodes obtain their packets by reading from this queue. The second queue returns the emptied packets to node 0. During initialization, the empty queue is pre-filled with a fixed number of packets.

Node 0 will create the two queues, each as a **Q_GLOBAL**, because they need to be accessed from other nodes. Specifying **Q_GLOBAL** in **q_create** will register the queue names in the pSOS global name table, so that other nodes can look them up using **q_ident**. Note that the producer nodes start with polling until the queues have been created by node 0.

```
extern Int   _node_number;
static UInt32 full_packets;
static UInt32 empty_packets;
void root(){
   void    *dummy;
   UInt32   ioretval;
```



```
    de_init(DEV_TIMER, 0, &ioretval, &dummy);

    if( _node_number == 0 ){
    /* Create queues named "EMPT" and "FULL" on node #0 */
       q_create( "EMPT", 0, Q_GLOBAL|Q_NOLIMIT|Q_FIFO, &empty_packets );
       q_create( "FULL", 0, Q_GLOBAL|Q_NOLIMIT|Q_FIFO, &full_packets  );

       create_empty_packets();
       produce();
    } else {   /* On all other nodes */
    /* wait until the send/receive queues have been received */
       while(  q_ident("EMPT",0,&empty_packets)
            || q_ident("FULL",0,&full_packets )
       ){}
       consume();
    }
}  /* never terminates */
```

Buffer and Packet Management, Caching Issues

Although pSOS queues can be used for data transfer, they are not recommended for high volume data streams. For this reason, only pointers to packet buffers are passed in this example. Packet buffers are allocated in the SDRAM of node 0, and it is the responsibility of the consuming nodes to copy the packet buffer in their own efficient way after they have read a buffer address from the global queue. As will be described in the next section, they will use the **tmDMA** device library for this.

Because the packet buffers will be read over the PCI bus from node 0's SDRAM by the consuming nodes, node 0 must take care to flush its data cache each time after having "filled" a buffer. In case it would forget to do so, part of the data written to the buffer might remain pending in node 0's data cache, resulting in stale data being read by the consumer who gets the buffer.

Conversely, because the consumers are going to use DMA for transferring the packet data to their local copy buffers, these local buffers must be cache-invalidated. The TriMedia DMA engine transfers to SDRAM without informing the data cache. Failing to invalidate the cache of the local buffers might result in stale data cache contents being read instead of the new SDRAM contents.

Flushing and invalidating the data cache contents that correspond to a memory range can be performed using TCS library functions **_cache_copyback** and **_cache_invalidate**. These functions are only allowed for memory ranges that do not share data cache pages with system data, or data from other pSOS tasks.

A safe way to obtain such memory ranges is by function **_cache_malloc**. Therefore, the producer node uses **_cache_malloc** for creating the packet buffers, and for similar reasons, the consumer nodes use this function for allocating their local copy buffers.

This gives rise to the following implementation of the packet create function, and of the producer-and-consumer loop. The producer continuously gets an empty packet from the empty packet queue, "fills" it, flushes the data cache, and puts the packet address on to the full packet queue. Each consumer continuously gets the address of a next full packet

on node 0, "transfers" its contents to its local copy buffer, and "uses" it. The next section describes how quick, DMA-based "transfer" can be accomplished. Functions "fill" and "use" are not described any further in this document (the example program "fills" with dummy data, while "use"checks whether the proper data has been received).

Further note that **q_send** actually sends a 4-word message. Since this example only sends pointers, only the first word of the message is used.

```
static void create_empty_packets(){
   Int i;

   for (i=1; i<=NROF_PACKETS; i++) {
      UInt32 message[4];
      message[0]= (UInt32)_cache_malloc(PACKET_SIZE);
      q_send(empty_packets, message);
   }
}
static void produce(){
   while (True) {
      UInt32 message[4];
      Char  *packet_ptr;

      q_receive (empty_packets, Q_WAIT, 0, message);
      packet_ptr= (Char*)message[0];
      fill            ( packet_ptr              );
      _cache_copyback ( packet_ptr, PACKET_SIZE );

      q_send ( full_packets,  message );
   }
}
static void consume(){
   Char *local_buffer;

   local_buffer= (Char*)_cache_malloc(PACKET_SIZE);
   _cache_invalidate(local_buffer,PACKET_SIZE);

   while (True) {
      UInt32 message[4];
      Char *packet_ptr;

      q_receive ( full_packets, Q_WAIT, 0, message );
      packet_ptr = (Char*)message[0];
      transfer( local_buffer, packet_ptr, PACKET_SIZE );
      use ( local_buffer );
      q_send ( empty_packets, message );
   }
}
```

DMA Transfer

This section concludes with a specialized DMA transfer function for this multiprocessing example. It is specialized, because it makes the following assumptions:

1. The parameter **local** is an address in the SDRAM of the processor calling this function.
2. The parameter **remote** is an address in the PCI space of the processor calling this function.

3. The memory range defined by parameters **local** and **size** does not have a pending write in the data cache.
4. The memory range defined by parameters **local** and **size** can be invalidated by this function (e.g, it has been obtained by **_cache_malloc**).

Note
The example uses this function according to these assumptions. Particularly, assumption 3 is fulfilled because the local copy buffer is invalidated immediately after allocation, and is never written to afterwards.

If the memory range defined by parameters **local** and **size** does not have a pending write in the data cache cannot be guaranteed, a cache invalidation is necessary also *before* the DMA dispatch, because otherwise a data cache page replacement could cause memory contents which was just placed into SDRAM by the DMA engine to be overwritten by stale memory updates.

```
static Int  dma_instance;
static Bool dma_opened = False;

void transfer( Char *local, Char *remote, Int size ){
   dmaRequest_t request;

   if (!dma_opened) {
      dmaOpen(&dma_instance);
      dma_opened= True;
   }
   request.slack_function      = Null;
   request.completion_function = Null;
   request.nr_of_descriptions  = 1;
   request.mode                = dmaSynchronous;
   request.done                = False;

   request.descriptions->direction       = dmaPCI_TO_SDRAM;
   request.descriptions->source          = remote;
   request.descriptions->destination     = local;
   request.descriptions->length          = size;
   request.descriptions->nr_of_transfers = 1;

   dmaDispatch(dma_instance, &request);
   _cache_invalidate(local,size);
}
```


Chapter 3

Using the Dynamic Loader on TriMedia

Topic	Page
Introduction	34
Dynamic Loading Basics	34
Dynamic Loader Example	35

Introduction

This section describes an example of an application using dynamic loading, in the form of a simple pSOS-based command dispatcher. The example can be found in the TCS release, in the following directory:

$ (TCS) /examples/dynamic_loading/dynamic_loader_shell

Dynamic Loading Basics

A dynamic loading code segment is specified by passing **-btype dynboot** to **tmld**. When **-btype** is not specified, **tmld** produces a **boot** code segment by default. All TriMedia programs that do not use dynamic loading are boot code segments.

To use dynamic loading, specify **-btype dynboot** as an option to **tmcc** when you compile a TriMedia executable. When you use dynamic loading with pSOS, first set the macro **DYNAMIC** in the pSOS application makefile to **dynamic**. This will add an option

```
-bembed $ (PSOS_SYSTEM) /sys/os/ $ (PSOS) _tm_$ (ENDIAN).dll
```

to **tmcc** when linking the executable, which will automatically specify **-btype dynboot**. (Refer to **$ (TCS) /examples/dynamic_loading/dynamic_loader_shell/Makefile**).

A **dynboot** code segment has the ability to load **app** and **dll** code segments, whereas a **boot** code segment cannot. The difference between an **app** and a **dll** is that an app must be explicitly loaded, while a dll is implicitly loaded when its exported symbols are accessed from another code segment (**dynboot**, **app**, or **dll**).

A **dynboot** code segment can load **app** code segments explicitly by a call to **DynLoad_load** from the DownLoader API specified in tmlib/DownLoader.h. Similarly, a call to **DynLoad_unload** will unload the **app** code segment. (Refer to **$ (TCS) /examples/dynamic_loading/dynamic_loader_shell/root.c**).

Note
Code segments for dynamic loading (dynboot, app, dll) cannot be compiled with **-g** for debugging because tmdbg does not currently support dynamic loading.

Dynamic Loader Example

This demo contains a simple pSOS based command dispatcher (*root.c*), in addition to three sample demo commands (*latency.c*, *task_demo.c* and *print_args.c*).

Root.c will be compiled and linked with pSOS into a code segment of type **dynboot**, and is able to load, run, and unload code segments of type **app**. The **app** programs have an entry point similar to **main (argc,argv)**, and are *not* self-contained; they must be loaded by a tm1 program (a **dynboot** code segment) that is currently running. (Refer to the implementation of **latency**, **task_demo**, and **print_args**.) The **app** programs cannot be loaded and run directly using **tmsim** or **tmmon**. Note that **apps** lack all of the system libraries (libraries for I/O using **printf**, or the pSOS library). For example, **tmsize** on **print_args** will reveal that its text segment contains only 512 bytes, which is considerably smaller than normal executables of type **boot** or **dynboot**. System libraries are contained by the command dispatcher, and will be connected during dynamic loading. After they are connected, they can be used normally by the application.

Starting Development

Since **dynamic_loader_shell** is a pSOS application, dynamic loading can be set up by setting the macro **DYNAMIC** to **dynamic** in the Makefile (as shown below).

```
DYNAMIC  = dynamic
```

The macro value **dynamic** links in the dynamic loader, allowing the command dispatcher to dynamically load the application files.

```
print_args.app: $ (OBJDIR) /print_args.o Makefile
                @ echo “Linking print_args.app”
                $ (CC) $ (CINCS) -btype app
                $ (OBJDIR) /print_args.o \
                $ (LDFLAGS) $ (CFLAGS) -o print_args.app
```

The **app** code segments to be dynamically loaded are **print_args**, **task_demo**, and **latency**. They are compiled with **-btype app** as options to **tmcc**, which will pass it directly to **tmld**.

The Root Function

In the **root** function, the command dispatcher repeatedly accepts a command string, and interprets the first word in this string as the name of an object file ending with a .app extension. A task is created for running the command, and the command string is passed in **argc**/**argv** format to this task.

```
void
root(void){
   void  *dummy;
   UInt32 rc, ioretval, iopb[4];
   Int    i;
   UInt32 task1, task2;
```

```
/* Start the pSOS system timer. This is almost always necessary, because
 * otherwise it is not possible to use timed events and timeslicing, or the
 * system clock: */

   de_init( DEV_TIMER, 0, &ioretval, &dummy );

   while (1) {
      UInt32  task;
      UInt32  arguments[4];
      Char    buffer[200];
      Int     argc;
      String *argv;

   /* Retrieve next command */
      printf( "> " );
      fflush(stdout);
      gets(buffer);
      strcpy(&buffer[strlen(buffer)]," ");

   /* Count number of arguments on command line */
      argc = count_words( buffer );

      if( argc > 0 ){
      /* Allocate space for argv plus a copy of the command string; this
       * will be passed to the application shell task and can be deallocated
       * as one unit */
        argv = (Pointer)malloc(argc*sizeof(Pointer)+strlen(buffer));
        strcpy( (Pointer)(argv+argc), buffer );
        get_words( (Pointer)(argv+argc), argv );

      /* After that, create a new task for the command to run on, and pass
       * it the argc/argv pair; Give it a priority of 231, which is higher
       * than this root task, otherwise the task will never run in tmsim
       * with its blocking input. The 10000's are the required sizes for
       * user- and system stack: */
        arguments[0] = (UInt32) argc;
        arguments[1] = (UInt32) argv;

        t_create( "aaaa ", 231, 10000, 10000, 0, &task );
        t_start( task, T_PREEMPT | T_TSLICE | T_ASR | T_ISR,
                 application_shell, arguments );
      }
   }
   _psos_exit(0);
}
```

The Application Shell

The task started by **root** for each command entered is **application_shell**. After it starts, it will attempt to load the code from **argv[0]**, and when it succeeds, it will call its main function with **argc**/**argv**. When **application_shell** terminates, the exit status is printed, and the corresponding code is unloaded. In case pSOS tasks are created by the loaded

command, it is the responsibility of the command itself to make sure that all tasks have been deleted (and certainly are not still executing) before **root** terminates.

```
void
application_shell(Int argc, String * argv){
   Int                          run_status;
   DynLoad_Status               load_status;
   DynLoad_Code_Segment_Handle module;

   load_status = DynLoad_load( argv[0], &module );

   if (load_status != DynLoad_OK) {
      printf( "** loading of `%s` failed with status %d\n ",
      argv[0], load_status);
   }else{
      run_status = ((Main_Function) module->start) (argc, argv);
      printf( "** `%s` done with status %d\n ", argv[0], run_status);
      DynLoad_unload(module->name);
   }
   free(argv);
   t_delete(0);
}
```

Running dynamic_loader_shell

In addition to the three .app programs provided in this example, (print_args.app, task_demo.app, and latency.app), any .app file in the other examples can be started using the command dispatcher; the .app file will run in parallel with applications that have previously started (but have not yet terminated), and will be scheduled by pSOS. When running one or more applications, the interrupt latencies can be sampled by running the provided latency.app for a specified number of seconds (default duration is 10 seconds):

```
> latency.app 100
```

will check the interrupt latency for 100 seconds.

The following are more examples on how to run .app files in this command dispatcher:

```
> vivot.app
```

and

```
> patest.app
```

Also refer to other examples in the **$ (TCS) /examples/dynamic_loading** directory.

