

TRIMEDIA™
SOFTWARE
DEVELOPMENT
ENVIRONMENT



COOKBOOK

Part 1: Developing TriMedia Applications

Part 2: Programming with Peripherals

Part 3: Bootstrapping TriMedia

Part 4: Optimizing TriMedia Applications

© 1998 Philips Semiconductors

All rights reserved.

No part of this publication or the software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Philips Semiconductors, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

Intel is a registered trademark of Intel Corporation. Microsoft, MS-DOS, Windows, Windows NT, Windows 95, and ActiveMovie are registered trademarks of Microsoft Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. MKS and MKS Toolkit are registered trademarks of Mortice Kern Systems Inc. Apple, Mac OS, and QuickTime are trademarks of Apple Computer, Inc. The following are trademarks of Integrated Systems, Inc.: DocumentIt, ESp, HyperBuild, OpEN, OpTIC, pHILE+, pNA+, pREPC+, pRISM, pRISM+, pROBE+, pRPC+, pSET, pSOS, pSOS+, pSOS+m, pSOSim, pSOSystem, pX11+, RealSim, SpOTLIGHT, SystemBuild, Xmath, ZeroCopy. Other product and company names mentioned herein may be the trademarks of their respective owners.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Philips Semiconductors retains all intellectual property rights associated with the technology described in this book.

Even though Philips Semiconductors has reviewed this manual, Philips Semiconductors makes no warranty or representation, either express or implied, with respect to this manual, its quality, accuracy, merchantability, or fitness for a particular purpose. As a result, this manual is sold "as is," and you, the purchaser, are assuming the entire risk as to its quality and accuracy.

In no event will Philips be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect or inaccuracy in this manual, even if advised of the possibility of such damages.

The warranty and remedies set forth above are exclusive and in lieu of all others, oral or written, express or implied.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Printed in the United States of America

Documentation Services by
International Consulting Group
San Jose, California
(www.icg-sj.com)



About The TriMedia SDE Cookbook

Manual Contents

The TriMedia SDE Cookbook contains the following four parts:

Part 1: Developing TriMedia Applications

Developing TriMedia Applications describes the processes that users follow to compile, link and test TriMedia applications.

Part 2: Programming with Peripherals

Programming with Peripherals describes the ways in which TriMedia libraries demonstrate the use of the hardware peripherals on the TriMedia chip.

Part 3: Bootstrapping TriMedia

Bootstrapping TriMedia describes the steps that must be followed to boot TriMedia.

Part 4: Optimizing TriMedia Applications

Optimizing TriMedia Applications describes some tips and tricks which are useful when optimizing applications to run on TriMedia.

Conventions Used in This Manual

This manual uses various conventions to present information. Words that require special treatment appear in special fonts or font styles. Certain information (such as code listings) appears in special formats so that you can scan it quickly.

Fonts and Colors

All code listings, command-line options, and names of structures and of functions are shown in *Courier*.

Structure and function parameters are shown in *Courier italics*.

Filenames and filename extensions (for example, .c or .o) are shown in normal font.

All TriMedia commands are shown in **boldface** (for example, **tmcc** and **tmdbg**).

All these special font conventions do not apply to references in titles. All text in titles appears in normal font.

Code listings and command line arguments are surrounded by gray boxes.

Types of Notes

This manual contains several types of notes.

Note

A note such as this contains information that is interesting, but possibly not essential to an understanding of the text. Notes can also tell you where to look for a more detailed discussion of a particular topic. ◆

IMPORTANT

A note such as this contains information that is essential for an understanding of the text. ▲

▲ WARNING

Warnings such as this indicate potential problems you should be aware of as you design your applications. Failure to heed these warnings could result in system crashes or loss of data. ▲

Philips TriMedia SDE Cookbook

Part 1:

Developing TriMedia Applications



Table of Contents

Chapter 1 **Compiling TriMedia Applications**

Introduction.....	1-2
Build and Execution Hosts	1-4
Build Hosts	1-4
Execution Hosts.....	1-4
Defining the TCS Environment Variable.....	1-5
Using tmcc to Compile TriMedia Applications.....	1-6
Invoking tmcc.....	1-6
Using tmcc Options	1-7
Specifying Execution Hosts	1-7
Compiling TriMedia Applications to Run on the Simulator .	1-8
Compiling TriMedia Applications to Run on the Chip	1-8
Compiling Multiple Files.....	1-8
Specifying Endianness	1-10
Predefined Macros	1-10
Creating Makefiles.....	1-11
Creating pSOS Makefiles	1-13
Simple pSOS Application Makefile	1-13
Porting This Makefile to nmake	1-14
Linking With Other pSOS Libraries.....	1-15
Using the pSOS Monitor.....	1-16
Running TriMedia Applications	1-17
Running TriMedia Applications with tmgmon.....	1-17
Dumping the Trace Buffer	1-18
Example.....	1-18
Running TriMedia Applications with tmrun.....	1-20

Running TriMedia Applications with tmmon.....	1-20
Running TriMedia Applications with tmdbg.....	1-20
Running TriMedia Applications with tmmprun.....	1-20

Chapter 2 Creating a GUI Interface

Introduction.....	2-2
Windows Application Program	2-2
Makefile	2-3
Main Program.....	2-4
Callback Function.....	2-7
File Operation	2-14
Initialization.....	2-14

Chapter 3 Programming With pSOS

Introduction.....	3-2
A pSOS Beginning	3-2
The Root Function	3-2
Communication Using Semaphores	3-4
Communication Using Asynchronous Signals.....	3-4
A pSOS Ending	3-5
A pSOS+™ Based Multiprocessor Example	3-5
Starting Development	3-6
Number of Executables to Build	3-6
The Root Function	3-7
Buffer and Packet Management, Caching Issues.....	3-8
DMA Transfer	3-10

Chapter 4 Using the Dynamic Loader on TriMedia

Introduction.....4-2

Dynamic Loading Basics.....4-2

Dynamic Loader Example.....4-3

 Starting Development.....4-3

 The Root Function.....4-3

 The Application Shell.....4-5

 Running dynamic_loader_shell.....4-6

Chapter 1

Compiling TriMedia Applications

Topic	Page
Introduction	1-2
Build and Execution Hosts	1-4
Defining the TCS Environment Variable	1-5
Using tmcc to Compile TriMedia Applications	1-6
Creating Makefiles	1-11
Creating pSOS Makefiles	1-13
Running TriMedia Applications	1-17

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 1-1 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.

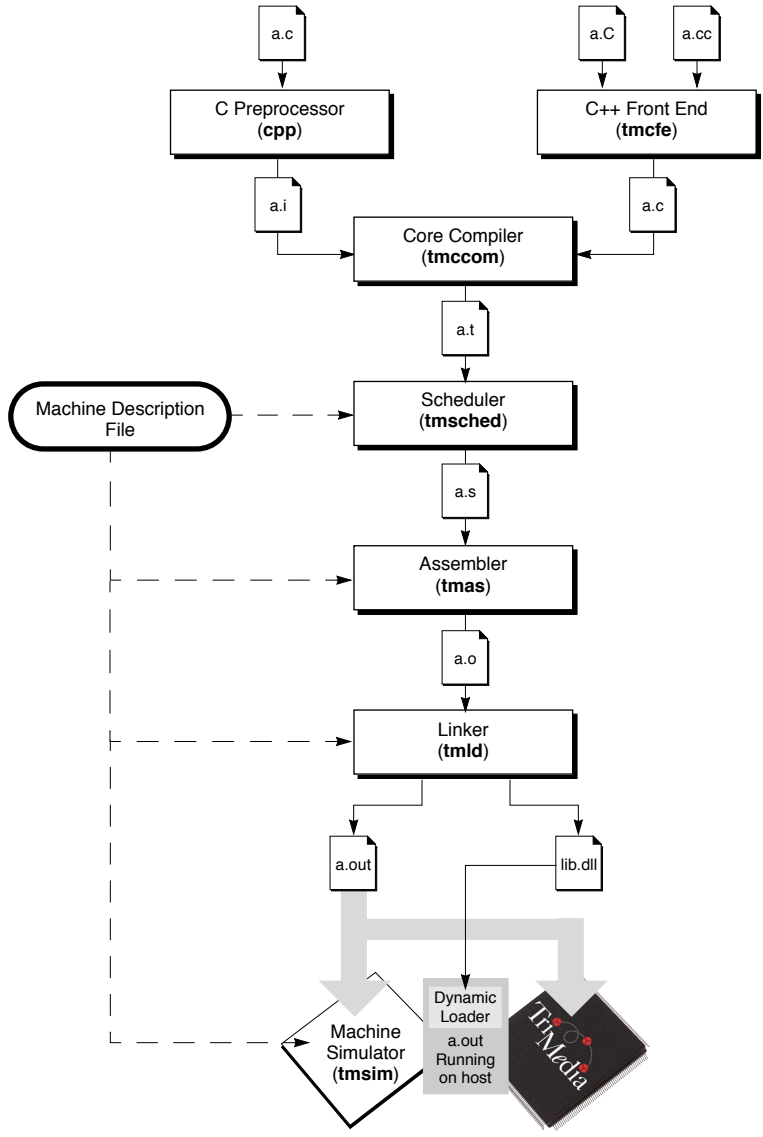


Figure 1-1 TriMedia Compilation and Simulation System

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
- 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:

- Windows 95
- Windows NT

Note

Although you can use the Windows 95 host for both building and running TriMedia applications, it is good practice 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. ♦

Defining the TCS Environment Variable

The **tmcc** compiler driver typically uses default paths to find the machine-description file, libraries, header files, and tools needed by the driver. You can control the driver configuration in several different ways.

By default, **tmcc** assumes that the TCS is installed in a directory tree rooted in the directory specified by the TCS environment variable. If the TCS is not set, **tmcc** uses the directory where it was originally installed (usually `/usr/local/tcs`) as its default value. Placing the TCS path in your system path enables you to call **tmcc** with no prefixes.

The **tmcc** compiler driver reads a *configuration file* (by default, `$TCS/tmconfig`) that specifies default options passed to the various compilation stages. You can specify an alternative location for the configuration file with the command-line option `-tmconfig=file`. For more information, Chapter 5, “Man Pages,” in Part 2 of TriMedia SDE Reference Manual I.

The **tmconfig** configuration file can specify alternative locations for the compilation tools, the machine-description file, the standard C runtime start off, and the standard C library.

The manual page for **tmcc** provides more information about the configuration file format. You can specify alternative tool locations in the **tmcc** command line. For example, option `-tmccom=/u/george/bin/tmccom` tells **tmcc** to use **tmccom** from the given location for compilation.

Note

For best results, define the TCS environment variable *only* inside makefiles. Defining the TCS environment variable in a startup script can lead to great confusion when multiple versions of the TriMedia Compilation System are resident on your computer. It is similarly wise to explicitly specify the TCS variable when calling **tmcc** from a makefile (`$(TCS)/bin/tmcc`). ♦

Using tmcc to Compile TriMedia Applications

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

IMPORTANT

The compiler front end (**tmcfe**) uses the “__0” prefix when it converts C++ names into C format. Consequently, some “valid” names may conflict with previous declarations which appear to be unrelated. For example, the following C++ program will not compile with **tmcc**:

```
float __0dDfooBx;
class foo {
    static int x;
};
```

This is because the mangled name for the static member of the class conflicts with the float declaration. We highly recommend that you follow the ANSI standard, or at least not prefix any identifiers with the special string “__0”. ♦

Invoking tmcc

You can invoke the **tmcc** (**tmCC** 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.

Extension	Description
.o	Unlinked object module. Output of the tmas assembler.
a.out	Linked executable. Output of the tmld linker.
.a	Library file. Output of the tmnar librarian.

The only difference between **tmcc** and **tmCC** is that **tmCC** assumes that compilation involves objects generated from C++ sources. Thus, **tmCC** 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 -cpp -pedantic -- foo.c
```

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

For more information about the **tmcc** options, Chapter 5, “Man Pages,” in Part 2 of TriMedia SDE Reference Manual I.

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 figures that enable you to track very precisely the performance on the simulator), 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 TM-1000 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 1-17 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 `-e1` (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 1-6. 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 host TM-1000 processor supports either big endian or little endian byte ordering, depending on the BSX bit in the PCSW. (See the *TM-1000 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 a few macros when it invokes either **cpp** (C programs) or the C++ front end **tmcfe** (C++ programs). The following macros are always defined:

Macro	Description
<code>__TCS__</code>	Defined during source file conditionalization to indicate source code specific to the TCS.
<code>__STDC__</code>	Defined to indicate compliance with the ANSI/ISO C Standard.
<code>__BIG_ENDIAN__</code>	Defined when compiling in big endian mode
<code>__LITTLE_ENDIAN__</code>	Defined when compiling in little endian mode.
<code>__TCS__host__</code>	Defined to indicate compilation for the given host <i>host</i>
<code>__TCS__target__</code>	Defined to indicate compilation for the given host <i>target</i>
<code>__cplusplus</code>	Defined by tmcfe to indicate C++ compilation.
<code>__TMSCHED__</code>	Defined by tmcc with the <code>-x</code> option when preprocessing a ".t" source file.
<code>__TMAS__</code>	Defined by tmcc with the <code>-x</code> 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 `TMCFFE_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/pSOSsystem
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 (/). It compiles the objects root.o and drv_conf.o and links them with the appropriate pSOS library, for Win95 with little endian. 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 $@ $<
```

5. 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) "
```

6. Change object file rule to

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

7. Make sure you take out all back slashes (\) for line separation
8. 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\psosSystem
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           = $(TCS)\bin\tmr
CINCS        = -I. -I$(PSOS_SYSTEM)\include
CFLAGS       =
LDLFLAGS     = -bremoveunusedcode -bcompact -bfoldcode

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

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 1-1 The Usage of pSOS in the Sample Makefile

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

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 $(LD_FLAGS) $(CFLAGS) -o
$(APPLICATION)
```

3. Remove linker optimizations in `LD_FLAGS`.

Running TriMedia Applications

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

- **tmgmon**
- **tmrn**
- **tmmon**
- **tmdbg**
- **tmmprn**

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

To simplify matters, make sure that **tmgmon** and **tmcons** are in the same directory as the program you are trying to run because **tmgmon** always starts in the directory in which it resides. This way, it is easy to locate the programs you're trying to run. ♦

Dumping the Trace Buffer

The Win95 execution host provides the DP function to be used for real-time debugging. This is described in Part 3: TriMedia Debugger in Reference Manual I. 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.

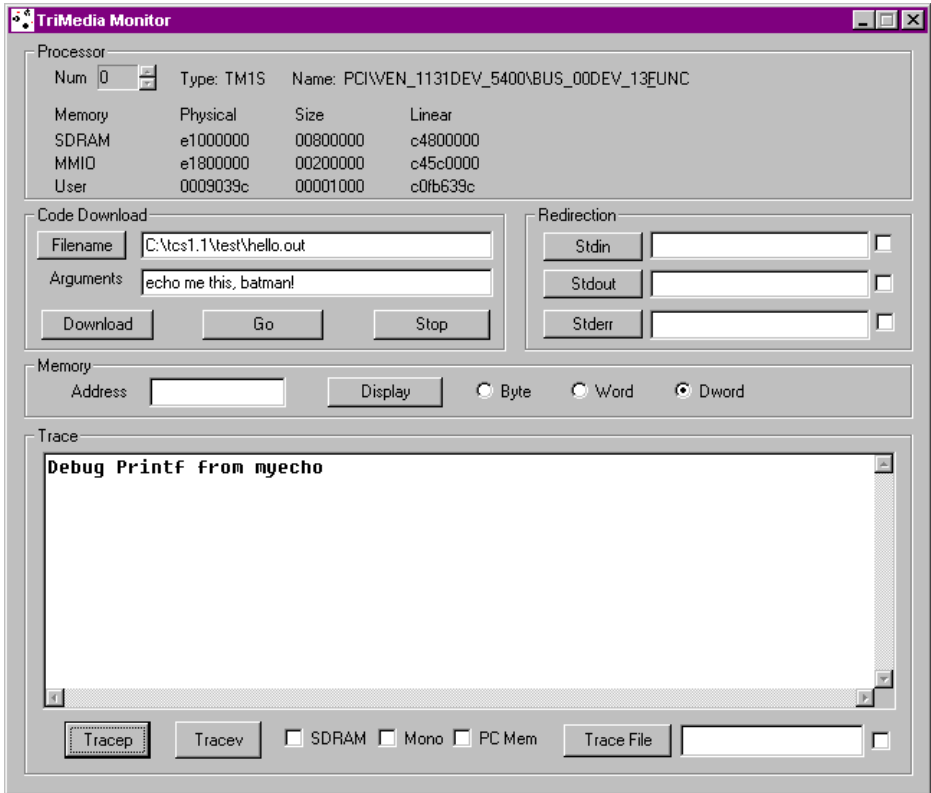
Example

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

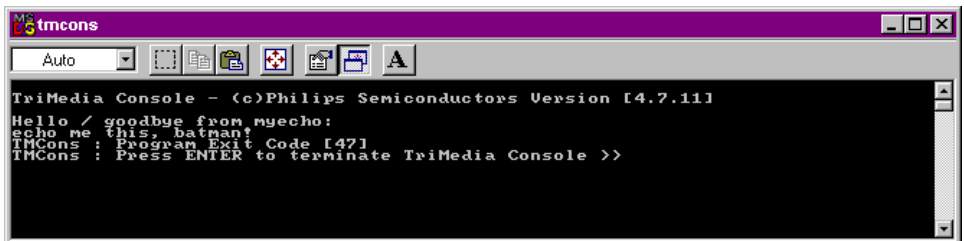
1. Compile the program to run on Windows 95.

```
#include <stdio.h>
#include <tmlib\dprintf.h>
main(int argc, char **argv)
{
    int i;
    DPmode(DP_PERSIST); /* for debugging */
    DPsize(64000);
    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.

For more information about **tmgmon**, Chapter 5, “Man Pages,” in Part 2 of TriMedia SDE Reference Manual I.

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:

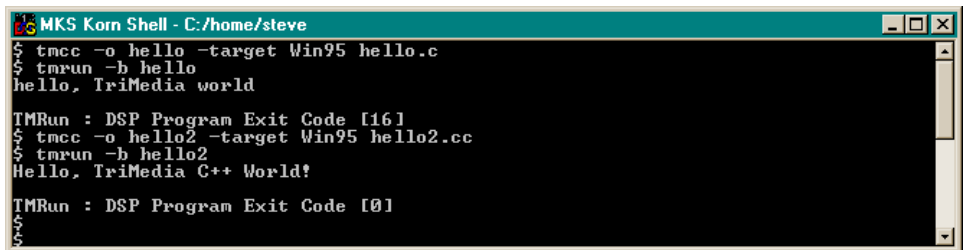
```
tmrun -b hello
```

For more information about **tmrun**, Chapter 5, “Man Pages,” in Part 2 of TriMedia SDE Reference Manual I.

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).



```
MKS Korn Shell - C:/home/steve
$ tmcc -o hello -target Win95 hello.c
$ tmrun -b hello
hello, TriMedia world
TMRUN : DSP Program Exit Code [16]
$ tmcc -o hello2 -target Win95 hello2.cc
$ tmrun -b hello2
Hello, TriMedia C++ World!
TMRUN : DSP Program Exit Code [0]
$
$
```

For more information about **tmmon**, Chapter 5, “Man Pages,” in Part 2 of TriMedia SDE Reference Manual I.

Running TriMedia Applications with tmdbg

You can run TriMedia applications using the **tmdbg** TriMedia debugger. Refer to Part 3 of Reference Manual I 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.

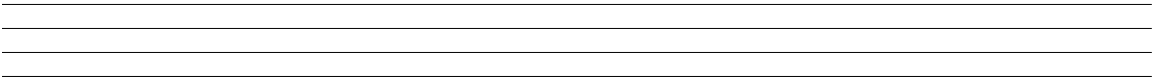
For example, to run the vivot demo application on the first processor and fplay on a second one, type the following:

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

For more information about **tmmprun**, Chapter 5, “Man Pages,” in Part 2 of TriMedia SDE Reference Manual I.

Chapter 2

Creating a GUI Interface



Topic	Page
Introduction	2-2
Windows Application Program	2-2
Makefile	2-3
Main Program	2-4
Callback Function	2-7
File Operation	2-14
Initialization	2-14

Introduction

The purpose of this program is to show how to create a GUI interface which is able to run programs on TriMedia. This program is able to load, run, or stop the execution of a program on the TriMedia. It offers the possibility of running four programs from the example directory by a choice of menu buttons.

Windows Application Program

The program was developed with Microsoft Visual C++ 4.0 (MSVC++ 4.0) but it was designed to be compiler independent. It should work with other Win32 compilers, for example Borland C++ 5.0, without problems.

Figure 1 shows a bitmap view of a fractal image. The fractal image was loaded from a file using the functionality provided in the example. In a full scale application, the fractal image could have been generated on TriMedia to take advantage of SIMD processing.

Figure 2 shows how a resource editor can be used to structure an application. Each of the objects in the window is identified by a unique prefix that is coded according to a Windows convention. The menu selections (`ID_OPEN`, `ID_SAVE`, `ID_SAVEAS`, `ID_EXIT`) are child windows of the File menu. Pressing a command button (`dma`, `files`, `sine`, `patest`) causes a TriMedia program to be started.

The resource editor can greatly simplify the creation of the GUI interface. It is possible to create a simple interface such as the one shown in an hour or less. If the presentation needs to be adjusted no recompilation is necessary. The interface can be tested for usability even without programming. To create the resource file shown, all that is necessary is knowledge of a few MSVC++ commands. The user should have no difficulty finding the necessary information.

Figure 3 shows the dialog box that is generated when About is selected from the Help menu. The information in the Window correspond to the processor state. For example, the contents of this Window could be copied and pasted for transmission to customer support.

Real time and compute-intensive parts of the application can be very effectively off-loaded from the main processor using TriMedia. However, to get the most benefit, it is essential that the application contain a host part that conforms to Windows API and quality standards. The purpose of the example program is to allow TriMedia programmers to become proficient in programming a basic Windows GUI with a minimum of effort. This will allow them to concentrate on obtaining the most added value from the power of TriMedia.

Makefile

The makefile of this program is detailed below.

```
#####
# GUI Windows App Makefile
#####
TCS      = C:\TriMedia
SDK      = C:\msdev
```

This line allows you to specify the directory where you installed TriMedia. Even if host application is being made, the header file (tmman32.h) and the library (tmman32.lib) is needed to access tmman functions. **tmman** implements host/target communication.

```
CC       = $(SDK)\bin\cl.exe
LD       = $(SDK)\bin\link.exe
RC       = $(SDK)\bin\rc.exe
```

These specify the compiler, linker, and resource compiler you are going to use. These are the values which will run with MSVC++. The Microsoft C compiler is invoked from the command line as 'cl'.

```
CFLAGS  = -c -I$(TCS)\include\Win95 -DSTRICT -zP4 -G3 -Ow
LDFLAGS = -SUBSYSTEM:windows
GUILIBS = -DEFAULTLIB:user32.lib gdi32.lib winmm.lib comdlg32.lib
          comctl32.lib \
          -LIBPATH:$(TCS)\lib\Win95 tmman32.lib
RCFLAGS = -r -DWIN32
```

The command line parameters are a bit different than other compilers. Note that the -I and -LIBPATH options are used to specify the paths for includes and libraries for TriMedia.

```
OBJ      = about.obj communication.obj error.obj files.obj trimedia.obj
```

The project consists of five C files, three H files, and one RC file.

```
trimedia.exe: $(OBJ) trimedia.res
              $(LD) $(LDFLAGS) -OUT:$@ $(OBJ) trimedia.res $(GUILIBS)
about.obj:   about.c trimedia.h resource.h
              $(CC) $(CFLAGS) about.c
communication.obj: communication.c trimedia.h
                  $(CC) $(CFLAGS) communication.c
error.obj:   error.c
              $(CC) $(CFLAGS) error.c
files.obj:   files.c trimedia.h resource.h
              $(CC) $(CFLAGS) files.c
trimedia.obj: trimedia.c trimedia.h main.h resource.h
              $(CC) $(CFLAGS) trimedia.c
trimedia.res: trimedia.rc resource.h trimedia.ico
              $(RC) $(RCFLAGS) trimedia.rc
```

```
clean:
    del *.obj
    del trimedia.res
    del trimedia.exe
```

A clean target has been added to remove unwanted files.

Main Program

Windows GUI applications are different from console applications because they are event-driven. This means that the application reacts to messages coming from the OS rather than controlling what goes on. For example, when the user clicks on a button this generates a `WM_COMMAND`. All the possible messages are defined in the `<windows.h>` header file supplied by Microsoft.

This means that any Windows GUI application is split in two parts. The `WinMain` (entry point of a Windows GUI application) function performs initialization. The `WndProc` function is a callback function which reacts to the events.

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  PSTR szCmdLine, int iCmdShow)
{
```

The main program in `trimedia.c` contains the entry point of the program. `WinMain` is the equivalent of the `main` function in an ordinary C program.

The following part is very important. It is in this part that the characteristics of the windows class (`WNDCLASSEX` struct) are set up. After that, we will be able to open as many windows as we want based on this class.

```
wndclass.cbSize      = sizeof (wndclass) ;
wndclass.style       = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = WndProc ;
```

This statement specifies the window message procedure which will be called by Windows whenever an event happens. Windows uses Hungarian notation for names. The lower case letters at the beginning of the name correspond to the type. `lpfn` means long pointer to function in this case.

```
wndclass.cbClsExtra = 0 ;
```

This statement allows the user to ask for more data for the class. `ClsExtra` is a count in bytes.

```
wndclass.cbWndExtra = DLGWINDOWEXTRA ;
```

The difference between class and window is important. The class information is shared. In this case, DGLWINDOWEXTRA extra bytes are reserved for each window but none for the class.

Windows GUI application can either be basic or Dialog based. Using a dialog based application simplifies the code because it does not have to deal with as many messages. However, more space is needed for the class. This value should be zero if the application is not dialog based. The space which is allocated is needed by Windows and does not concern the programmer.

```
wndclass.hInstance = hInstance ;
```

An instance in Windows corresponds to a system object (a process). Objects in Windows are normally referred to via a handle (a pointer to a pointer). The type (h) corresponds to the first letter. hInstance corresponds to the process ID in this case. It is given as a parameter to winMain. Handles are of integer type.

```
wndclass.hIcon = LoadIcon (hInstance,  
MAKEINTRESOURCE( IDI_TRIMEDIA ) ) ;
```

This gets a handle to the customized icon. Resources are identified in the resource file by integers to save space. The macro converts this to a string (“TriMedia in this case”).

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
```

This gets a handle to the Windows default cursor.

```
wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW ) ;
```

This gets a handle to the background brush (grey in this case).

```
wndclass.lpszMenuName = MAKEINTRESOURCE( IDR_MENU ) ;
```

Windows identifies menu names with strings. In the resource file, these are represented by integers to save space. The type is long pointer to zero terminated string (lpsz). This corresponds to a normal C string.

```
wndclass.lpszClassName = "TriMedia" ;
```

A class is a sort of template for a window. Classes are also identified by strings.

```
wndclass.hIconSm = LoadIcon (hInstance, "TriMedia") ;
```

This statement is equivalent to the LoadIcon call above.

At this point, all the resource aspects of the window have been specified. The icon is defined in the file `trimedia.ico`, and the menu is defined in the resource file `trimedia.rc`.

```
RegisterClassEx (&wndclass) ;
```

This communicates the template information to the OS. This is required before a window can be created.

```
hwnd = CreateDialog (hInstance,
                    MAKEINTRESOURCE(IDD_TRIMEDIA), 0, NULL) ;
```

This creates a window of the class we previously defined. Handles of windows are a key part of a window application.

The characteristics defined above are not enough to create an application with buttons, edit boxes, etc. There are two ways to deal with this. They can be created on the fly with some calls to `CreateWindow`. In this case, a resource file was used instead. This is much easier, especially with the resource editors available in most Win32 IDEs. `IDD_TRIMEDIA` corresponds to the name “TriMedia” defined above. A convention for Windows programming is being used here. Resources normally begin with ID with another letter to specify the type. D stands for Dialog, I for Icon, M for menu, C for control (buttons, edit boxes), etc. Figure 2-1 on page 2-6 shows what this resource looks like.

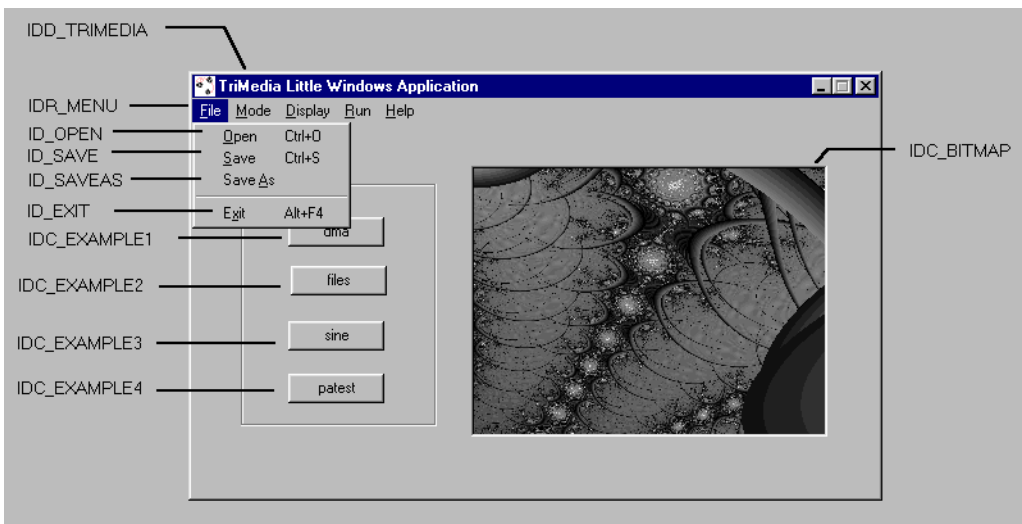


Figure 2-1

```
ShowWindow (hwnd, iCmdShow) ;
```

This specifies how to display the window (maximized, minimized) depending on the value of `iCmdShow` given as a parameter.

```
UpdateWindow(hwnd) ;
```

The effect of this function is to send to the callback function a `WM_PAINT` message. This forces Windows to display the window.

```
hAccel = LoadAccelerators (hInstance,
                          MAKEINTRESOURCE(IDR_ACCELERATOR1)) ;
```

It loads the customized accelerators which are defined in the resource file. Accelerators are basically keystrokes which can be used as shortcuts. Standard shortcuts are supported by this example (F1 to get help, Ctrl+O to open a file, Ctrl+S to save the file).

```
while (GetMessage (&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(hwnd, hAccel, &msg))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
}
```

The last part of the `WinMain` is a busy loop which waits for messages from the Operating System and then sends them to the window that is appropriate.

```
return msg.wParam ;
}
```

This terminates the main program.

Callback Function

The second part of the main program is the message callback function.

```
LRESULT CALLBACK WndProc (HWND hwnd,
```

The first parameter is the handle to the window.

```
UINT iMsg,
```

This identifies the message to be processed (`iMsg`). These are numerous and are defined in `windows.h`. All the system events are broadcast at least to the active window. An application has to handle the messages it understands. Otherwise it should request the default behavior.

```
WPARAM wParam,
LPARAM lParam)
```

The last two parameters, which allow the user to get more information about the circumstances of the message.

```
{
switch (iMsg)
{
case WM_CREATE:
```

The `WM_CREATE` message is used to for initialization. It is sent as soon as the window is created. This message is usually a good place to put initialization code.

```
Lib=LoadLibrary("tmman32.dll");
```

Windows uses dynamically linked libraries (DLLs) instead of libraries that are linked with the application. This loads the library from the filesystem. It searches in:

1. the directory from which the program was started
2. the directory where the program is stored
3. The Windows directory
4. The Windows system directory
5. from the `PATH` environment variable

`Tmman32.dll` is used for communication with the TriMedia board. It contains all the functions needed to access the board.

```
errTriMedia = InitTriMedia();
```

We initiate the communication with the TriMedia (see `communication.c`).

```
InitializeFile(hwnd);
```

This initializes the file handling (look at `files.c`).

```
return 0;
```

This terminates processing for the message. The return value indicates that the message has been handled and that no error was encountered.

To understand the next message, please refer to Figure 2-2. Menu number 2 (View) can be configured with one of three values (bitmap view, or BMP in this case).

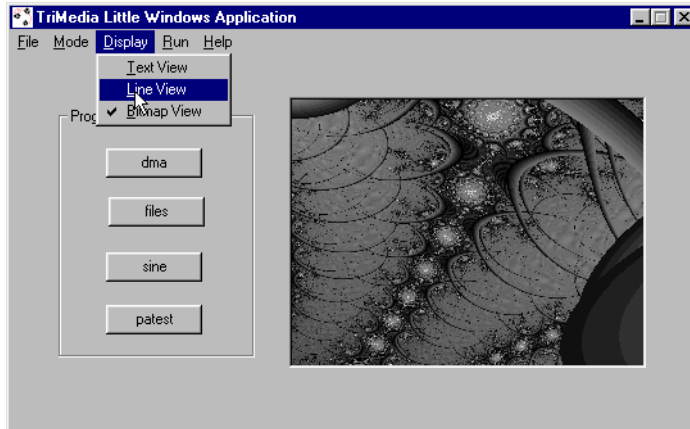


Figure 2-2

```
case WM_INITMENUPOPUP:
    switch(lParam)
```

lParam contains the number of the menu which is about to be opened.

```
    {
        case 2:          /*View Menu */
            MyCheckMenuItem(ID_VIEW_TEXT,
                            GUIFlags.View);
            MyCheckMenuItem(ID_VIEW_LINE,
                            GUIFlags.View);
            MyCheckMenuItem(ID_VIEW_BMP,
                            GUIFlags.View);

            break;
```

This compares the position with the three possible values in the figure. A check mark is added where appropriate.

```
        case 1:        /* Mode Menu */
            MyCheckMenuItem(ID_MODE_SEQ,
                            GUIFlags.Mode);
            MyCheckMenuItem(ID_MODE_TASK,
                            GUIFlags.Mode);

            break;
```

This does the same thing for the Mode Menu (not shown).

```
default:
    break;
```

Menus number 0 (File) and 3 (Run) and 4 (Help) are not configured with checkmarks, and therefore do not need any special handling.

```
    }
    return 0;

case WM_COMMAND:
```

The `WM_COMMAND` message is very general. Windows sends it when a daughter window (such as a button, an edit box, a combo box...) receives a message. The four following `IDC_EXAMPLEn` cases correspond to the four buttons. We determine which is the example to run, and then launch it. The `#define` values are defined in `main.h`.

```
switch (LOWORD(wParam))
{
```

`LOWORD(wParam)` contains the ID (as defined in the resource file) of the daughter windows. `HWORD(wParam)` contains the name of the windows message (not used).

```
case IDC_EXAMPLE1:
```

The `IDC_EXAMPLEn` values are codes for the programs to be launched.

```
    wprintf(lpszExample, "%s\\%s",
            EXAMPLE_PATH,
```

The `EXAMPLE_PATH` directory contains the programs to be launched.

```
        "dmatest",
```

The first example is the DMA test program.

```
        ".out");
```

The extension corresponds to that of a TriMedia executable.

```
case IDC_EXAMPLE2:
```

```
    ...
```

The source code for the other examples is similar.

The `WM_COMMAND` message is also used, when the user selects one of the items in the menu, or when a keystroke is issued. The value is the same regardless of whether an accelerator or a menu button has been selected. This corresponds to the following case.

```
case ID_OPEN:
    hNewBitmap=LoadFile(hwnd, FileName,
                        TitleName);
```

The `LoadFile` and `SaveFile` functions are in `files.c`. The function displays a `OpenFile` dialog box, and returns a Handle of bitmap from it.

```
if (hNewBitmap==NULL)
    break;
hStaticBitmap = GetDlgItem
                (hwnd, IDC_BITMAP);
hBitmap=(HBITMAP)SendMessage
        (hStaticBitmap,
         STM_SETIMAGE,
         (WPARAM) IMAGE_BITMAP,
         (LPARAM) (HANDLE)hNewBitmap);
```

With the `STM_SETIMAGE` message, the handle is assigned to the bitmap static box (`IDC_BITMAP`). The message is routed through Windows to the application itself. It is normal for a Windows application to generate as well as to receive messages. The parameters of `SendMessage` are exactly those of the callback function (messages).

```
if (hBitmap!=NULL)
    DeleteObject(hBitmap);
hBitmap=hNewBitmap;
break;
```

A bitmap handle also stores the bitmap data. It is especially important to delete the handle as this is a lot of space. Under Windows, garbage collection is not handled automatically and applications are required to clean up after themselves.

```
case ID_SAVE:
case ID_SAVEAS:
    SaveFile(hwnd, FileName, TitleName);
    break;
```

This calls the `SaveFile` function which just pops up the `OpenFile` dialog box. The appropriate code can be added at this point.

```
case ID_ABOUT:
    DialogBox (hInstance,
              MAKEINTRESOURCE (IDD_ABOUT),
              hwnd,
              AboutDlgProc) ;
    break;
```

This creates a child dialog box, with `IDD_ABOUT` resource. The `AboutDlgProc` callback function is defined in `about.c`. The function will not exit until the user closes the dialog box.

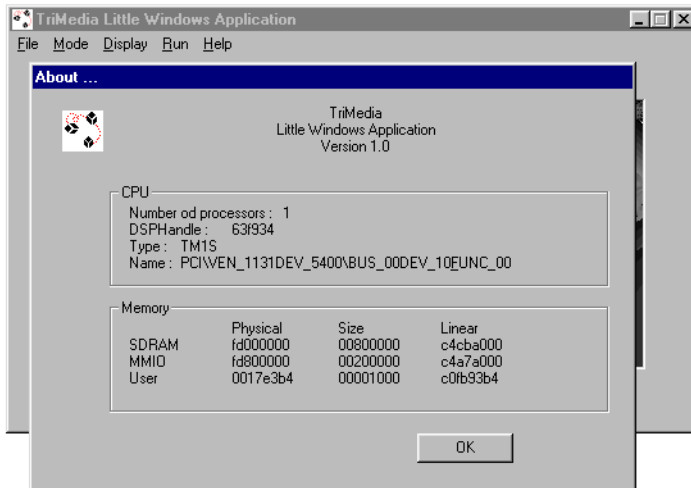


Figure 2-3

```

case ID_MODE_SEQ:
case ID_MODE_TASK:
    GUIFlags.Mode=LOWORD(wParam);
    break;

case ID_VIEW_TEXT:
case ID_VIEW_LINE:
case ID_VIEW_BMP:
    GUIFlags.View=LOWORD(wParam);
    break;

```

These messages are sent when the user clicks on one of the menu items in the View, or the Mode menu. The value is intercepted, so that we can internally store the current value in the `GUIFlags` struct. The same values are used for the ID of the menu (`ID_XXX_YYY`), and for the internal struct to simplify processing.

```

case ID_EXIT:
    SendMessage(hwnd,WM_CLOSE,0,0);
    break;

```

This case corresponds to an “exit” menu selection being made. A Windows program cannot terminate by calling the `exit` function directly. A `WM_CLOSE` message must be sent

which results in many other messages. The last of these is `WM_DESTROY` (see below). The first parameter is the window. The two final parameters must be zero.

```

        }
        return 0;

    case WM_DESTROY :

```

All the resources must be freed when the `WM_DESTROY` message is processed.

```

        if (hBitmap!=NULL)
            DeleteObject(hBitmap);

```

This frees the space corresponding to the window (if any).

```

        FreeLibrary(Lib);

```

This liberates the DLL.

```

        if (errTriMedia==0)
            ExitTriMedia();

```

This terminates the c communication with TriMedia (see the `ExitTriMedia` function in `communication.c`). This code is very important because Windows is unable to understand just what resources need to be freed. For a library, Windows will not be able to get rid of it as long as the process owns a lock (even if the process is dead). Also, the number of handles for resources in Windows is limited. This can result in having to reboot the system, because of resource exhaustion.

```

        PostQuitMessage (0) ;

```

The `PostQuitMessage` is the Windows equivalent of the exit function.

```

        return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;

```

This last line is very important. Although we cannot process every Windows message (there are hundreds). we still have to tell windows to process the messages we chose not to process.

File Operation

We will now explain how to create the standard `FileOpen` popup windows. If you have used several applications, You will probably have noticed that the `OpenFile` dialog boxes look the same. This is because they use the `OPENFILENAME` object which is defined in `<commdlg.h>`. Font selection and color selection (Choose Font, Choose Color) are defined in this file also. All the files operations are in the `files.c` file.

```
static OPENFILENAME ofn;
```

This declares an `OPENFILENAME` object. Since we are going to use it for Load and Save operations, it is declared as global.

The file I/O can be done with the classic `fopen/fclose` functions. These functions are not as high level as the Windows specific ones. However, they do the job, and are a lot simpler.

Initialization

The initialization function is called from `WndProc` when a `WM_CREATE` message is received. It loads the default values for the `OPENFILENAME` object.

```
void InitializeFile(HWND hwnd)
{

static char szFilter[] = "Bitmap Files (*.BMP)\0*.bmp\0" \
                        "Text Files (*.TXT)\0*.txt\0" \
                        "ASCII Files (*.ASC)\0*.asc\0" \
                        "All Files (*.*)\0*.*\0\0" ;
```

This array corresponds to a standard format for the `OpenFile` function. The first string element corresponds to a menu selection by the user for the file type. The second is used by Windows to display the appropriate files in the window. As many file types as necessary can be supported.

```
memset((char *)&ofn, 0, sizeof(OPENFILENAME));
```

Most of the members of the `OPENFILENAME` struct are zero by default.

```
ofn.lStructSize      = sizeof (OPENFILENAME) ;
ofn.hwndOwner        = hwnd ;
```

The first two parameters correspond to the size of the structure and a handle for the window.

```
ofn.lpstrFilter      = szFilter ;
```

This specifies that the filter for filenames is the array defined above.

```
ofn.nMaxFile        = _MAX_PATH ;
ofn.nMaxFileName    = _MAX_FNAME + _MAX_EXT ;
```

The first value corresponds to the maximum path name length. The second corresponds to the maximum file name length (name + extension). The values defined in `stdio.h` are used.

```
ofn.lpstrDefExt     = "bmp" ;
}
```

This specifies the extension by default (“bmp”). This corresponds to a bitmap.

The two following functions will do a specific setup of the `OPENFILENAME` struct according to whether we want to save or load a file.

```
BOOL FileOpenDlg (HWND hwnd, PSTR pstrFileName, PSTR pstrTitleName)
{
    ofn.hwndOwner      = hwnd ;
    ofn.lpstrFile       = pstrFileName ;
    ofn.lpstrFileTitle  = pstrTitleName ;
    ofn.Flags           = OFN_HIDEREADONLY | OFN_CREATEPROMPT ;

    return GetOpenFileName (&ofn) ;
}
```

This function will fill the `pstrFileName` and the `pstrTitleName` according to the file that the user selected. It also says, that the read-only files will not be displayed (`OFN_HIDEREADONLY`), and that if the user type a file which is not in the list, a message box will popup asking if he wants to create this file (`OFN_CREATEPROMPT`). Note that the Dialog Box will be displayed only when the `GetOpenFileName` function is called. The return value of `GetOpenFileName` is used to check if the user did not press the Cancel button.

```
BOOL FileSaveDlg (HWND hwnd, PSTR pstrFileName, PSTR pstrTitleName)
{
    ofn.hwndOwner      = hwnd ;
    ofn.lpstrFile       = pstrFileName ;
    ofn.lpstrFileTitle  = pstrTitleName ;
    ofn.Flags           = OFN_OVERWRITEPROMPT ;

    return GetSaveFileName (&ofn) ;
}
```

This is similar to the previous function. Here, we use the `OFN_OVERWRITEPROMPT` flag: the user will be prompted whether he wants to overwrite the file he is specifying if the file already exists. Note that, in these two functions, we use `GetSaveFileName` and the `GetOpenFileName` functions which are responsible for displaying the classic `OpenFile` dialog box.

The following two functions, are the functions which are directly called from the main program. These functions will call the two functions we have just explained. Note that we check the return value of these functions, since if these functions return `FALSE`, it means a problem occurred: the user may have pressed `CANCEL`, or did not want to overwrite a file when prompted.

In these functions, you should add your own code. At this point, this code will load the file, and display it in the edit box.

```
int LoadFile(HWND hwnd, char *FileName, char *TitleName)
{
    ...

    if (FALSE==FileOpenDlg(hwnd, FileName,TitleName))
        return 1;
    ...

int SaveFile(HWND hwnd, char *FileName, char *TitleName)
{
    ...
    if (FALSE==FileSaveDlg(hwnd,FileName,TitleName))
        return 1;
    ...
}
```

This document does not describe `communication.c` since this file deals with communication with `TriMedia`. The setup and the use of the `TriMedia` are detailed in part 3 of the Cookbook, chapter 2: *Bootstrapping TriMedia in Host-Assisted Mode*. The functions are basically the same. For further information refer to the source of `communication.c`. This source is documented. For more information about creating a GUI, the best place to start is *Programming Windows 95*, by Charles Petzold (Microsoft Press).

Chapter 3

Programming With pSOS

Topic	Page
Introduction	3-2
A pSOS+™ Based Multiprocessor Example	3-5

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;
    ULONG rc, ioretval, iopb[4];

    int i;

    ULONG 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, aaa, 0 );
t_start( task2, T_PREEMPT | T_TSLICE | T_ASR | T_ISR, catch, 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 “cat” 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 catch() {
    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 task 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/misc/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 any number of processors larger than or equal to 2. Any additional processor will automatically become a consumer. Figure 3-1 shows an N- node configuration.

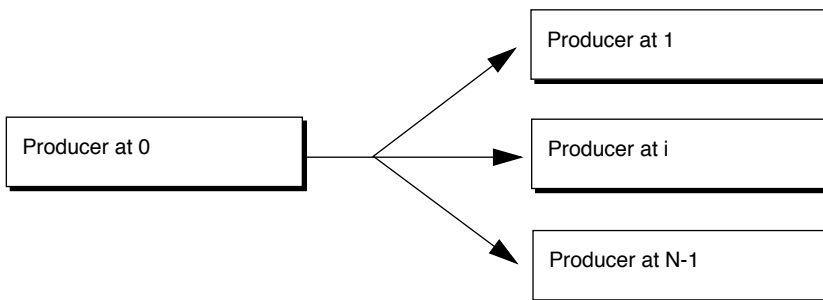


Figure 3-1 N-Node Configuration

Starting Development

To start development, you must first set up a pSOS+™ application. This is started by copying directory \$(TCS)/OS/pSOS/pSOSsystem/apps/demo_1 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 consumers. 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 TM-1000 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 will be filled with produced packets by node 0; all other nodes will obtain their packets by reading from this queue. The second queue is used for returning 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 ULONG full_packets;
static ULONG empty_packets;

void root()
{
    void *dummy;
    ULONG 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 TM-1000 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++) {
        ULONG message[4];

        message[0]= (ULONG)_cache_malloc(PACKET_SIZE);

        q_send(empty_packets, message);
    }
}

static void produce()
{
    while (True) {
        ULONG 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) {
        ULONG 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 4

Using the Dynamic Loader on TriMedia

Topic	Page
Introduction	4-2
Dynamic Loading Basics	4-2
Dynamic Loader Example	4-3

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/misc/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:

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

to **tmcc** when linking the executable, which will automatically specify `-btype dynboot`. (Refer to `$(TCS) /examples/misc/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/misc/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;
    ULONG     rc, ioretval, iopb[4];

    Int       i;

    ULONG     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);

    while (1) {
        ULONG     task;
        ULONG     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) {

```

◆ **Note:** Code is continued on the next page.


```

/*
 * 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] = (ULONG) argc;
arguments[1] = (ULONG) 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
```

Refer to the dynlink_demo in the \$(TCS)/examples/psos/psos_dynlink_demo, as well.

Philips TriMedia SDE Cookbook

Part 2:

Programming with Peripherals



Table of Contents

Chapter 1 Programming TriMedia Video Applications

Introduction.....	1-2
TSSA Video Modules.....	1-2
The Video Digitizer	1-2
The Video Renderer	1-3
The exoVrendVO Example Program	1-3
Include Files.....	1-3
Definitions	1-4
Specifying the Packet Format	1-4
Static Parameters and Function Prototypes	1-5
The Main Program	1-5
Variables	1-6
DP Debug Information	1-6
Check Capabilities	1-7
Read Command Line Parameters	1-7
Open the Components.....	1-7
Make the Connection Between the Two Components	1-8
Setup the Video Digitizer and Renderer	1-10
Starting the Component Instances	1-11
User Input	1-12
Stop and Shutdown	1-13
TriMedia Video-In Operation	1-15
Full-Resolution Capture Mode.....	1-15
Full-Resolution Capture Mode.....	1-16
Half-Resolution Capture Mode	1-16
Raw Capture Mode.....	1-16
Message-Passing Mode	1-17

TriMedia Video-Out Operation	1-17
Image Transfer Mode	1-18
Data Transfer Modes	1-18
Data-Streaming Mode.....	1-18
Message-Passing Mode	1-19
Using the TriMedia Video-In/Video-Out Device Library	1-19
Guidelines for Use of the Video-In/Video-Out APIs	1-20
Vivot Demonstration Program Overview.....	1-20
C Program Includes	1-21
Main Program	1-21
Vivot Demonstration Program (Vivorun)	1-22
Image Representation	1-22
Buffer Allocation (vivoAlloc)	1-23
Cache Management.....	1-23
viOpenAPI - level 1 initialization for VI	1-25
voOpenAPI - level 1 initialization for VO	1-26
Field Capture versus Frame Capture.....	1-27
Running in CIF Resolution (vivoRunCIF).....	1-27
Running in Full Resolution (vivoRunFullRes)	1-29
Initialization With Alpha Overlay (vivoRunOverlay).....	1-29
Setup Input and Begin Capture (viYUVOpenAPI)	1-31
Start Outputting an Image To Video Out (voYUVAPI)	1-32
Initialize Overlay Mode (voOverlayAPI)	1-35
Inputting an Image for Display on VO (readYUVfiles)	1-36
ICP Setup	1-37
Buffer Processing for Full Resolution and CIF	1-39
Buffer Processing for Overlay (mmOvlyBufUpdate).....	1-39
VI Interrupt Service Routine (viTestISR).....	1-40
Querying the Configuration.....	1-43

Chapter 2 Programming TriMedia Video Applications Using the ICP TSSA API

Introduction.....	2-2
--------------------------	------------

The exoIVtransICP Example Program	2-3
Include Files	2-3
Definitions.....	2-4
Static Variables.....	2-5
Specifying the Packet Format.....	2-5
Specifying the Output Format.....	2-6
Packet Defines and Function Prototypes.....	2-7
Variables.....	2-8
Initialization.....	2-10
Get Capabilities	2-11
Make the Connection Between the Two Components	2-12
Create the Video Transformer Control Descriptor.....	2-13
Setup the Video Digitizer	2-13
Setup the Video Transformer	2-15
Starting the Component Instances	2-16
User Input.....	2-17
Stop and Shutdown	2-21
Application Progress Function.....	2-22
Application Completion Function.....	2-22

Chapter 3 Programming TriMedia Audio Applications

Introduction.....	3-2
TSSA Audio Modules	3-3
The Audio Renderer	3-3
Check Capabilities:	3-6
Open the Components:.....	3-6
Make the Connection Between Each Pair of Components:	3-7
Setup the File Reader.....	3-8
Setup the Audio Renderer	3-8
Start	3-8
Stop and Shutdown	3-9

Advanced Features.....	3-10
Audio Digitizer	3-11
CopyAudio Example	3-12
Create the Components:.....	3-12
Create and Populate the Queues	3-13
Set Up the Components	3-14
Modifying the Copy Component:	3-14
Audio Mixer.....	3-15
Audio Decoders	3-15
Audio Device Library.....	3-16
Audio Hardware Overview	3-16
Capture/Transmission by DSPCPU	3-16
Using the TriMedia Audio-In/Audio-Out API.....	3-17
Guidelines for Use of the Audio-In/Audio-Out APIs	3-17
Restrictions	3-18
Demonstration Programs.....	3-18
Playing an Audio File	3-19
Interrupt Routine fplayISR	3-20
Recording an Audio File	3-22
sthru Demonstration Program.....	3-22
Setting Audio Parameters	3-23
Board Support Package.....	3-27

Chapter 1

Programming TriMedia Video Applications

Topic	Page
Introduction	1-2
TSSA Video Modules	1-2
TriMedia Video-In Operation	1-15
TriMedia Video-Out Operation	1-17
Using the TriMedia Video-In/Video-Out Device Library	1-19
Vivot Demonstration Program Overview	1-20

Introduction

This chapter describes how to write video applications using several programming interfaces available on TriMedia. For a detailed description of these APIs, refer to Reference Manuals I and II of the Philips TriMedia SDE.

This chapter begins by describing the high level interface to the Video-In and Video-Out peripherals. These interfaces are provided using the Video Digitizer and Video Renderer components and enable an application to be written without requiring knowledge of the underlying hardware peripherals. An overview of the operation of the TriMedia Video-In and Video-Out units is then presented. This provides background material which is useful when understanding the use of the low-level Video-In/Video-Out device libraries which will then be described.

TSSA Video Modules

The high level interface is supported using modules which conform to the TriMedia Streaming Software Architecture (TSSA). This software architecture is documented in Reference manual I, Part 4. There are several TSSA compliant modules which support video data; examples of interest include the Video Digitizer, the Video Renderer, and the Video Transformer. The Video Digitizer and Video Renderer will be discussed in this chapter, while the Video Transformer is discussed in the next chapter.

The Video Digitizer

The video digitizer supports video capture using data streaming (*pull* mode) operation; it is described in more detail in Chapter 9, “TriMedia Video Digitizer API”, of Reference Manual II Part 2. In the *pull* mode of operation, the component obtains an empty packet using the *datain* callback function from an operating system message queue (the *empty* queue). It then captures a video frame, and using the same *datain* callback function, places the full packet onto another message queue (the *full* queue). This streaming operation is supported in both the AL and OL layers; the AL API layer assumes no operating system dependencies, while the OL API layer does.

The application can specify parameters which include the video standard (NTSC, PAL, or SECAM), the adaptor type (CVBS or SVIDEO), and the size of the frame to capture.

The `exolVrendVO` example demonstrates how the Video Digitizer can be used to capture video data. This example will be described in detail after the reader has been introduced to the Video Renderer component.

The Video Renderer

The VrendVO Video Renderer component is used to display video images using the TriMedia Video-Out peripheral. The component supports non-streaming (*push* mode) and streaming (*pull* mode) operation in the AL Layer. It also supports streaming operation in the OL Layer. In *push* mode, the application calls a Video Renderer function which will display the frame, i.e. the application *pushes* the frame to the renderer.

The Video Renderer supports several video standards and adaptor types. It is also capable of combining the main video with an overlay image with alpha blending. The `exolVrendVO` example shows the use of this component and will be described next.

The exolVrendVO Example Program

The `exolVrendVO` example demonstrates the use of the OL Layer Video Digitizer and Video Renderer. As it uses OL versions of the APIs, data streaming is used to transfer data packets between components. The example simply connects an instance of the Video Digitizer to an instance of the Video Renderer. The digitizer captures live data using the Video-In device while the renderer displays these images using the Video-Out device. The example enables the user to specify parameters such as the video standard (NTSC or PAL), the adaptor type (CVBS or SVIDEO), and whether to use full resolution or SIF resolution images.

The source code for this example is contained within the `examples/exolVrendVO` directory of the application tree. This example will now be described in detail.

Include Files

```
#include <tml/tmAvFormats.h>
#include "tmos.h"
#include "tmolVrendVO.h"
#include "tmolVdigVI.h"
#include <stdio.h>
#include <tmlib/dprintf.h> /* for debugging with DP(()) */
#include "sys_conf.h"
```

The `tmAvFormats.h` file contains definitions for the packets which are used to store the video data. The `tmos.h` file abstracts the underlying operating system; this enables the code to be ported to different operating systems by simply changing this file. The type definitions and function prototypes for the two video components are defined in the `tmolVrendVO.h` and `tmolVdigVI.h` files respectively.

Definitions

```
#define IMAGE_NTSC_HEIGHT 480
#define IMAGE_PAL_HEIGHT 576
#define IMAGE_WIDTH 720
#define IMAGE_STRIDE 768
#define NUMPACKETS 4
#define NEW_PARAMETER 0
```

The size of the captured and displayed video frame height is defined. Note that the `IMAGE_STRIDE` is larger than the `IMAGE_WIDTH`. This is because the stride must be a multiple of 64 bytes; as the image width is 720 bytes, the nearest 64 byte multiple which is greater than or equal to this is 768.

`NUM_PACKETS` defines the number of packets which will be used to transfer data between the two components. The `NEW_PARAMETER` is used in the user interface code to determine if a new command should be processed.

Specifying the Packet Format

```
static tmVideoFormat_t digitizer_format = {
    sizeof(tmVideoFormat_t), /* size */
    0, /* hash */
    0, /* referenceCount */
    avdcVideo, /* dataClass */
    vtfYUV, /* dataType */
    vdfYUV422Planar, /* dataSubtype */
    vdfInterlaced, /* description */
    IMAGE_WIDTH, /* imageWidth */
    IMAGE_NTSC_HEIGHT, /* imageHeight */
    IMAGE_STRIDE /* imageStride */
};
```

This structure defines the format of the data contained in the packets. The `hash` and `referenceCount` fields must be set to zero, and should never be modified by the application. They are used by the format manager which ensures that connected components are compatible.

The `dataClass` and `dataType` must always be set to `avdcVideo` and `vtfYUV`. These specify that the class of data is video and is YUV. The `dataSubtype` is set to `vdfYUV422Planar` and specifies the sub-type of YUV data. The Video Digitizer can capture either `vdfYUV422Planar` or `vdfYUV422Interspersed` video; both types store the Y, U, and V components in separate buffers. The `vdfYUV422Planar` sub-type has the chrominance samples co-sited with the luminance data, while `vdfYUV422Interspersed` has the chrominance located mid-way between luminance samples.

The `description` field is set to `vdflInterlaced` to indicate that the Video Digitizer is capturing interlaced video. The digitizer will store the two fields in a single buffer, with the top field being on the even lines.

Finally, the size of the video frame is specified.

Static Parameters and Function Prototypes

```
static tmVideoAnalogStandard_t vidStd      = vasNTSC;
static tmVideoAnalogAdapter_t  vidAdapter = vaaCVBS;

extern int __argc;
extern char **__argv;

/* ----- function prototypes ----- */
static int DoCommand ( char *command );
static int CheckArgcv(int argc, char **argv);
static void PrintUsage(void);

/* setup parameters */
int vResolution    = viFULLRES;
int acquStartX     = 0;
int acquStartY     = 0;
int scaleUp        = False;
int voStartX       = 0;
int voStartY       = 0;
```

The global variables used for the video configuration are declared and initialized. These are used to enable the user to change the configuration by entering commands on the console. The default settings for the video standard and adaptor are NTSC and CVBS. The digitizer will capture full resolution images, and the renderer will perform no upscaling on the output.

The function prototypes are for the user interface code. This will not be described.

The Main Program

The following code is contained within the `main{ }` function.

Variables

```

tmLibappErr_t      rval;
Int                digitizerInstance;
Int                vrendInstance;
char               ins[80];
ptmolVdigVICapabilities_t  digCap;
ptmolVrendVOCapabilities_t  rendCap;
ptmolVrendVOInstanceSetup_t  vrend_inst_setup;
ptmolVdigVIIInstanceSetup_t  digitizer_inst_setup;
ptsaInOutDescriptorSetup_t  iodSetup;
ptsaInOutDescriptor_t      iod;

```

The `rval` variable is used to store the value returned whenever a call is made to the Video Digitizer, Video Renderer, or `tsaDefaults` API. The returned value is always of type `tmLibappErr_t` and will have a value of `TMLIBAPP_OK` if there is no error. It is important to check the returned value whenever a call to a component API is made.

The `digitizerInstance` and `vrendInstance` variables are used to store the instance id's when the digitizer and renderer instances are opened. These id's are unique and must be used whenever the application calls a component API function.

The `ins[80]` character array is used to store user command typed in at the keyboard.

Before two components are connected together to form a data flow, the application uses the format manager to determine if they are compatible. Each component has a capabilities structure which specifies what formats it can understand. The `digCap` and `rendCap` variables are used to point to these capabilities structures.

Each component must also be setup before it is used. The `vrend_inst_setup` and `digitizer_inst_setup` are pointers to the instance setup structures.

The connection between two component instances is described using a `tsaInOutDescriptor`. When a descriptor is created, it requires a setup structure which specifies information about the two components being connected and the packets which will be used. The `iodSetup` variable is used to point to this setup information.

DP Debug Information

```

DPmode(DP_PERSIST);
DPsize(1024*1024);

```

The TriMedia SDE provides a mechanism where debug information can be written to SDRAM by the application and component libraries. This can then be read either during execution if the debugger is being used, or after the program has completed.

Check Capabilities

```

rval = tmlVdigVIGetCapabilities(&digCap);
rval = tmlVrendVOGetCapabilities(&rendCap);

printf("TriMedia OS Video Renderer Demo. v1.0\n");
printf("\nThis program uses the video digitizer v%d.%d.%d\nand video
renderer v%d.%d.%d\n",
digCap->defaultCapabilities->version.majorVersion,
digCap->defaultCapabilities->version.minorVersion,
digCap->defaultCapabilities->version.buildVersion,
rendCap->defaultCapabilities->version.majorVersion,
rendCap->defaultCapabilities->version.minorVersion,
rendCap->defaultCapabilities->version.buildVersion);
printf(" to pass video from video-in to video-out.\n");

```

The capabilities of the two components to be connected together are obtained using the respective GetCapabilities functions. This information will be used by the format manager to ensure the two components are compatible. The versions of the two components are printed on the console.

Read Command Line Parameters

```

tmosInit();

if ( CheckArgcv(__argc, __argv) != 0 )
    tmosExit(0);

```

The multitasking operating system is initialized and the command line arguments are checked.

Open the Components

```

rval = tmlVdigVIOpen(&digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmlVrendVOOpen(&vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);

```

Before a component can be used, it must first be opened using the respective open function. The relevant function will open an instance of the component, and store a unique instance id in the pointer parameter. The application must use the instance id when calling the components API. It is important to check the return value to ensure that an error did

not occur during the open operation. For example, the Video Digitizer and Video Renderer only support a single instance to be open, if the application incorrectly tries to open a second instance then the function will return an error.

Make the Connection Between the Two Components

```
iodSetup = (ptsaInOutDescriptorSetup_t)
    malloc(sizeof(tsaInOutDescriptorSetup_t)+2*sizeof(UINT32));
iodSetup->format          = (ptmAvFormat_t)&digitizer_format;
iodSetup->flags           = tsaIODescSetupFlagCacheMalloc;
iodSetup->fullQName       = "full";
iodSetup->emptyQName      = "mpty";
iodSetup->queueFlags      = tmosQueueFlagsStandard;
iodSetup->senderCap        = digCap->defaultCapabilities;
iodSetup->receiverCap     = rendCap->defaultCapabilities;
iodSetup->senderIndex     = VDIGVI_MAIN_OUTPUT;
iodSetup->receiverIndex   = VRENDVO_MAIN_INPUT;
iodSetup->packetBase      = 0;
iodSetup->numberOfPackets = NUMPACKETS;
iodSetup->numberOfBuffers = 3;
iodSetup->bufSize[0]      = IMAGE_STRIDE * IMAGE_PAL_HEIGHT;
iodSetup->bufSize[1]      = IMAGE_STRIDE * IMAGE_PAL_HEIGHT / 2;
iodSetup->bufSize[2]      = IMAGE_STRIDE * IMAGE_PAL_HEIGHT / 2;
rval = tsaDefaultInOutDescriptorCreate(&iod, iodSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
```

The dataflow path connecting two components is specified using an InOutDescriptor. Before this descriptor is created, a structure specifying the connection must be initialized with information which describe the capabilities of the two components and information about the packets which will be placed in the queue.

The first step is to create the setup structure using the standard malloc function. The amount of the memory requested is equal to the size of the `tsaInOutDescriptorSetup_t` structure plus the number of buffers per packet minus one. As the packets store YUV data, three buffers are required per packet, so the application needs to add two extra `UINT32` fields to the allocated memory which will be used to store the U and V buffer sizes. By default, the `tsaInOutDescriptorSetup_t` has space for one buffer size.

The format of the packets which will be placed in the full queue are specified by passing the address of the `digitizer_format` structure. This information is used by the format manager to check that the components can accept this type of packet. It will also be placed automatically on packets when the sender instance (the Video Digitizer in this case) places a packet onto the full queue.

The `flags` parameter is set to `tsaIODescSetupFlagCacheMalloc`. This indicates to the `tsaDefaultInOutDescriptorCreate()` function that the packet buffers which it creates must be cache aligned.

Information concerning the queues which will be automatically created are then initialized. The full and empty queues are given names which can be used during debugging; any four letter name can be used. The `queueFlags` parameter provides information which will be used when the full and empty queues are created. The `tmosQueueFlagsStandard` specifies that the queues will be local to the processor (i.e. they do not connect processors) and there is no limit to the number of messages which can be placed on them.

The capabilities of the two components which will be connected together will be checked by the format manager to ensure that they are compatible. The `senderCap` is set to the address of the digitizer capabilities, while the `receiverCap` is set to the renderer capabilities. The component capabilities were obtained previously using the `tmolVdigVIGetCapabilities()` and `tmolVrendVOGetCapabilities()` functions.

The `senderIndex` and `receiverIndex` fields specify the output and input pins which will be used for the connection. Each component instance uses input and/or output pins for communication to neighboring component instances; each pin represents the full/empty message queue where packets are exchanged. The Video Digitizer has a single output pin referenced by the index value `VDIGVI_MAIN_OUTPUT`. The Video Renderer has two input pins, one for the main video input (`VRENDVO_MAIN_INPUT`) and one for the overlay input (`VRENDVO_OVERLAY_INPUT`). The `receiverIndex` is set to `VRENDVO_MAIN_INPUT` as this pin will receive the video packets for display.

The next set of fields will be used to provide information about the packets which will be automatically created. The `packetBase` field is used to specify an identification number to the packets that are placed in the queues. The application can use any number; the first packet will contain this value, with subsequent packets containing id's with ascending values. In the example, the first packet will have an id of zero, the second packet will be one, the third will be two, and the fourth packet will have an id of three. This can be useful for debugging to identify where the packets are being held. The `numberOfPackets` specifies the number of packets which must be created and stored in the empty queue. The `numberOfBuffers` specifies the number of data buffers per packet. As the components are using YUV data, three buffers are required per packet to store the Y, U, and V data. Each buffer has a corresponding `buffSize` value which specifies the size of the buffer. As the packets are hold YUV data, the first `bufferSize` is set to the size of the luminance component, with the subsequent `bufferSize` values set to the size of the Chrominance components. As the data is YUV422, the chrominance is half the size of the luminance.

Finally, the `InOutDescriptor` is created using `tsaDefaultInOutDescriptorCreate()`. This creates the descriptor, the message queues, and the associated packets.

Setup the Video Digitizer and Renderer

```

rval = tmolVdigVIGetInstanceSetup(digitizerInstance,
                                  &digitizer_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVrendVOGetInstanceSetup(vrendInstance,
                                    &vrend_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);

digitizer_inst_setup->instSetup>outputDescriptors[VDIGVI_MAIN_OUTPUT]
                    = iod;

digitizer_inst_setup->videoStandard = vidStd;
digitizer_inst_setup->videoAdapter  = vidAdapter;
digitizer_inst_setup->capSizeFlag   = vResolution;
digitizer_inst_setup->startX        = acquStartX;
digitizer_inst_setup->startY        = acquStartY;

vrend_inst_setup->instSetup->inputDescriptors[VRENDVO_MAIN_INPUT]
                = iod;

vrend_inst_setup->videoStandard  = vidStd;
vrend_inst_setup->adapterType    = vidAdapter;
vrend_inst_setup->scaleUp        = scaleUp;
vrend_inst_setup->imageHorzOffset = voStartX;
vrend_inst_setup->imageVertOffset = voStartY;

rval = tmolVdigVIInstanceSetup(digitizerInstance, digitizer_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer initialized.\n");

rval = tmolVrendVOInstanceSetup(vrendInstance, vrend_inst_setup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("renderer initialized.\n");

```

Before an instance of a component is used it must first be setup. The first step is to obtain a pointer to the instance setup structure; this is achieved by calling the `tmolVdigVIGetInstanceSetup()` and `tmolVrendVOGetInstanceSetup()` respectively.

The Video Digitizer structure is setup first. The instances output descriptor is set to point to the `InOutDescriptor` which was created in the last section of code. The input `videoStandard` specifies either PAL or NTSC; by default, this is set to NTSC. The input `videoAdaptor` indicates the adaptor type and can be CVBS or SVIDEO, with the CVBS being set by default. The `capSizeFlag` indicates whether to perform full resolution or half resolution video capture; by default this will be full resolution. Finally, the `startX` and `startY` fields are used to specify the location in the incoming field where video capture will start. The two values are zero by default.

The Video Renderer parameters are then initialized. The instances main image input descriptor is set to the `InOutDescriptor` which was created before. The output video

standard and adaptor type are setup in similar fashion to the Video Digitizer. The `scaleUp` flag is used to specify that the input image should be scaled up by the video-out hardware. If full resolution images are captured, this value should be set to false. Half resolution images may be scaled up to full resolution by setting this value to true. Finally, the `imageHorzOffset` and `imageVertOffset` specify the starting pixel and line in the active output video area where the image will be displayed. These are set to zero by default.

Once the setup structures have been initialized, the `tmolVdigVIInstanceSetup()` and `tmolVrendVoInstanceSetup()` functions are called to pass the information to the two instances.

Starting the Component Instances

```
DP(("Starting Video Renderer\n"));
rval = tmolVrendVOSTart(vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("renderer started.\n");

DP(("Starting Video Digitizer\n"));
rval = tmolVdigVISTart(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer started.\n");
```

Data streaming between the two component instances will begin once both have been started. The `tmolVdigVISTart()` and `tmolVrendVOSTart()` functions will initiate data streaming for each instance. Both components execute in interrupt service routines.

User Input

```

printf("\nVideo Renderer demo started.\nVideo input is being echoed to
video output.\n");
PrintUsage();
printf("Enter Command:\n");

while (1)
{
printf(">");
gets(ins);
rval = DoCommand(ins);
if ( rval == NEW_PARAMETER ) {
if (rval = tmlVdigVISTop(digitizerInstance))
printf("exolVrendVO: tmlVdigVISTop error %s\n",rval);
if (rval = tmlVrendVOSTop(vrendInstance))
printf("exolVrendVO: tmlVrendVOSTop error %s\n",rval);
digitizer_inst_setup->videoStandard = vidStd;
digitizer_inst_setup->videoAdapter = vidAdapter;
digitizer_inst_setup->capSizeFlag = vResolution;
digitizer_inst_setup->startX = acquStartX;
digitizer_inst_setup->startY = acquStartY;
vrend_inst_setup->videoStandard = vidStd;
vrend_inst_setup->adapterType = vidAdapter;
vrend_inst_setup->scaleUp = scaleUp;
vrend_inst_setup->imageHorzOffset = voStartX;
vrend_inst_setup->imageVertOffset = voStartY;
tsaDefaultInstallFormat (iod,
                        (ptmAvFormat_t)&digitizer_format);
if (rval = tmlVdigVIInstanceSetup(digitizerInstance,
                                digitizer_inst_setup))
printf("exolVrendVO: tmlVdigVIInstanceSetup error %s\n", rval);
if (rval = tmlVrendVOInstanceSetup(vrendInstance,
                                vrend_inst_setup))
printf("exolVrendVO: tmlVrendVOInstanceSetup error %s\n",rval);
if (rval = tmlVrendVOSTart(vrendInstance))
printf("exolVrendVO: tmlVrendVOSTart error %s\n",rval);
if (rval = tmlVdigVISTart(digitizerInstance))
printf("exolVrendVO: tmlVdigVISTart error %s\n",rval);
}
else if (rval == -1){
continue;
}
else {
break;
}
}

```

The example enables the user to enter commands via the console which alter the digitizer and renderer parameters. While the two video instances are streaming data, the default task will wait for the user to type a command. Once a valid command has been entered, the two

instances are stopped by calling `tmolVdigVISTop()` and `tmolVrendVOSTop()` respectively; this will terminate data streaming. The new instance values are then assigned to the respective instance setup structures and each component instance is setup. Finally, data streaming is restarted for both the digitizer and renderer.

If the user types 'exit' at the console, then the 'while' processing loop will be exited, and the shutdown sequence of command will be executed.

Stop and Shutdown

```
printf("\nStopping Everything:\n");
DP("\nStopping Everything:\n");
rval = tmolVdigVISTop(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVrendVOSTop(vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVdigVIClose(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVrendVOClose(vrendInstance);
tmAssert((rval == TMLIBAPP_OK), rval);

rval = tsaDefaultCheckQueues(iod);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("tsaDefaultCheckQueues returned 0x%x\n", rval);

printf("Destroying InOutDescriptor\n");
rval = tsaDefaultInOutDescriptorDestroy(iod);
tmAssert((rval == TMLIBAPP_OK), rval);

DP("Demo Complete.\n");
printf("Demo Complete. \n");
tmosExit(0);
}
```

Component instances should be stopped before they are closed. The `tmolVdigVISTop()` and `tmolVrendVOSTop()` functions will cause the two instances to stop data streaming and return any packets that they may be holding. The low level video-in and video-out devices will also be stopped within these functions.

After use, each component instance should be closed. For the Video Digitizer and Renderer which only allow one instance to be opened, this will enable other applications or tasks to use the components. The `tmolVdigVOClose()` and `tmolVrendVOClose()` will free any memory that was being used by the instances.

The `InOutDescriptor` full and empty queues can be checked using the `tsaDefaultCheckQueues()` function. This function should be used during debugging and checks the queues to ensure that the correct number of packets have been returned to them.

Finally, the `InOutDescriptor` should be destroyed by calling `tsaDefaultInOutDescriptorDestroy()`. This will remove the packets contained within the queues, free the memory allocated to the packets, and free the memory allocated to the `InOutDescriptor`.

TriMedia Video-In Operation

The TriMedia Video-In unit provides digital video input in YUV 4:2:2 with 8-bit resolution, multiplexed in CCIR656 format from a digital camera or CCIR656-capable video decoder (such as a Philips SAA7111), across an 8-bit wide interface.

The Video-In unit can operate in any one of the following modes:

- Full-resolution capture
- Half-resolution capture
- Raw capture (raw8, raw10s, and raw10u)
- Message passing

An operation in each of these modes is given below. For more information, refer to chapters 6 and 7 of the Data Book.

Full-Resolution Capture Mode

In Full-resolution Capture mode, the Video-In unit receives all three video components (Y, U, and V), as well as synchronization information, on the 8-bit wide interface in CCIR656 format. The Y, U, and V video components are separated into three different streams. Each component is written in packed form into Y, U, and V buffers in the SDRAM. This is commonly called a *planar format*.

The DSPCPU initiates capture by setting the `CAPTURE_ENABLE` flag to 1. The Video-In unit captures video data and stores it in the SDRAM, at the locations defined by the storage parameters defined in the MMIO registers. When capture is complete (that is, any internal Video-In buffers have been flushed and the entire captured image is in local SDRAM), Video-In sets the `STATUS` register flag to `CAPTURE_COMPLETE`. This causes a DSPCPU interrupt to be requested. The Video-In unit resumes capture as soon as the DSPCPU acknowledges the previously captured image by deactivating `CAPTURE_COMPLETE`.

You can program the `Y_THRESHOLD` field to generate pre-completion (or post-completion) interrupts. Whenever `CUR_Y` reaches the `Y_THRESHOLD`, the `THRESHOLD_FLAG` in the status register is set. If enabled in the Video-In control register, this event causes a DSPCPU interrupt request.

If the Video-In internal buffers overflow because of insufficient internal data-highway bandwidth allocation, the `HIGHWAY_BANDWIDTH_ERROR` condition is raised in the Video-In status register (`VI_STATUS`). If enabled, this causes a DSPCPU interrupt request. Capture continues at the correct memory address as soon as the internal buffers can be written to memory, but one or more pixels may be lost, and the corresponding memory locations are not written.

Full-Resolution Capture Mode

Full-Resolution Capture mode is illustrated in the vivot example that follows.

Half-Resolution Capture Mode

Half-Resolution Capture mode is identical in operation to full-resolution capture mode, except that horizontal resolution is reduced by a factor of 2 on both luminance and chrominance data.

Half-Resolution Capture mode is used for CIF format in the vivot example that follows.

Raw Capture Mode

All Raw Capture modes (`raw8`, `raw10s`, and `raw10u`) behave similarly. The video data is captured at the rate of the sender's clock, without interpretation or start/stop on the basis of the data values.

The DSPCPU initiates capture by providing two empty buffers and putting their base addresses and sizes in the `BASEn` and `SIZEn` registers. It does so by writing a base address and size to MMIO control fields. After two buffers are assigned, capture is enabled by setting `CAPTURE_ENABLE` to 1. The Video-In unit starts capturing video data in `buffer1` (the active buffer). It continues until capture is disabled or `buffer1` fills up. If `buffer1` fills up, capture continues (without missing a sample) in `buffer2`. At the same time, `BUF1FULL` is asserted, which causes an interrupt on the DSPCPU.

`buffer2` then becomes the active buffer and the loop repeats. In normal operation, the DSPCPU before `buffer2` fills up, the DSPCPU must assign a new, empty buffer `BASE1`, `SIZE1` and perform an `ACK1` operation. If the DSPCPU fails to assign a new `buffer1` before `buffer2` fills up, the `OVERRUN` condition is raised, bringing a temporary halt to capture. Capture resumes as soon as the DSPCPU makes one or more new buffers available through an `ACK1` or `ACK2` operation.

If insufficient bandwidth is allocated from the internal data highway, the Video-In internal buffers might overflow. This leads to assertion of the `HIGHWAY BANDWIDTH ERROR` condition. One or more data samples are lost. Capture resumes at the correct memory address as soon as the internal buffer is written to memory.

Message-Passing Mode

In Message-Passing mode, the Video-In unit receives 8-bit message data across the 8-bit wide interface. It writes the message data in packed form (four 8-bit message bytes per 32-bit word) to the SDRAM. Message data capture starts on receipt of a `START` event and continues until either the receive buffer is full, or the `EndOfMessage` event is received. `OVERFLOW` is raised if a receive buffer is full and no `EndOfMessage` event has been received. If enabled, it generates a DSPCPU interrupt. Detection of overflow leads to total halt of capture of this message. Capture resumes in the next buffer on receipt of the next `START` event.

The TriMedia Video-In APIs provide the necessary interface for video applications to access the TriMedia Video-In unit hardware.

TriMedia Video-Out Operation

The TriMedia Video-Out unit connects to an off-chip video subsystem, such as a digital video encoder chip (DENC), a digital video recorder, or the video input of another TriMedia system through a CCIR656-compatible byte-parallel video interface.

The Video-Out unit outputs digital video in YUV 4:2:2 co-sited format with 8-bit resolution multiplexed in CCIR656 format. It can drive a CCIR656-compatible digital video encoder across an 8-bit wide interface. It can also drive other CCIR656-compatible devices, such as digital video cassette recorders (VCRs) and the Video-In unit of other TriMedia chips. For example, in Video-In Diagnostic Mode, the Video-Out unit of one TriMedia supplies video data to the Video-In unit of a second TriMedia system.

The Video-Out unit can operate in either *image transfer* or *data transfer* (data streaming or message-passing) mode. The TriMedia DSPCPU programs the Video-Out unit by setting the `Mode` field to the appropriate transfer mode, setting the appropriate addresses, address deltas, image-timing registers, and associated control bits in the control register. Setting `VO_ENABLE` in the `VO_CNTRL` register starts the Video-Out unit, which transfers the image or messages as commanded.

In image-transfer and data-streaming modes, the Video-Out unit runs continuously. It issues an interrupt to the DSPCPU at the end of each field. To maintain continuous video output, the DSPCPU updates the Video-Out image data pointers with pointers to the next field during the vertical blanking interval. In message-passing mode, the Video-Out unit runs until the message has been transferred.

Image Transfer Mode

In Image Transfer mode, the Video-Out unit continuously transfers an image from the SDRAM to the Video-Out port. The mode field in the `VO_CTL` register defines the image input data format and whether or not the Video-Out unit is to perform horizontal upscaling. The Video-Out unit accepts memory image data in YUV 4:2:2 co-sited, YUV 4:2:2 interspersed, and YUV 4:2:0 co-sited image output streams.

During image transfer, the `YTR` bits are set in the status register when the Image Line Counter reaches the `Y_THRESHOLD` value. When an image field has been transferred, the `BFR1_EMPTY` bit is set in the status register. The DSPCPU is interrupted when either the `YTR` or the `BFR1_EMPTY` flag is set and its corresponding interrupt is enabled.

To maintain continuous transfer of image fields, the DSPCPU supplies new pointers for the field following each `BFR1_EMPTY` interrupt. If the DSPCPU does not supply new pointers before the next field, the `URUN` bit is set, and the Video-Out unit uses the same pointer values until they are updated.

Image Transfer mode is illustrated in the vivot example that follows.

Data Transfer Modes

There are two modes for transferring data:

- Data-Streaming mode
- Message-Passing mode

Data-Streaming Mode

In the Data-Streaming mode, the Video-Out unit generates a continuous stream of byte data using internal or external clocking. Dual buffers facilitate continuous data streaming by allowing the DSPCPU to set up the next buffer while the first one is being emptied by the Video-Out unit.

The data is stored in the DRAM in two buffer tables. When the Video-Out unit has transferred the contents of one table, it interrupts the DSPCPU and begins transferring the contents of the second table. The DSPCPU supplies pointers to both tables. The Video-Out unit supplies a continuous stream of data to the video device, provided the DSPCPU updates the pointer to the next table before the Video-Out starts transferring data from the next table.

When each buffer has been transferred, the corresponding buffer empty bit is set in the status register. The DSPCPU is interrupted if the buffer empty interrupt is enabled. To maintain continuous transfer of data, the DSPCPU supplies new pointers for the next data buffer following each buffer empty interrupt. If the DSPCPU does not supply new pointers

before the next field, the Video-Out unit uses the same pointer values until they are updated.

Message-Passing Mode

In the Message-Passing mode, messages can be sent to one or more TriMedia Video-In units. Start and end-of-message signals are provided in this mode to synchronize message passing to the other TriMedia message receivers. Video data is stored in the DRAM in one buffer table.

Setting `VO_ENABLE` in the `VO_CNTRL` register starts the Video-Out in Message-Passing mode. The Video-Out unit sends a `Start` condition on `VO_IO1`. When the Video-Out unit has transferred the contents of the buffer table, it sends an `End` condition on `VO_IO2`, sets the `BFR1_EMPTY` bit, and interrupts the DSPCPU. The Video-Out unit stops. No further operation takes place until the DSPCPU sets `VO_ENABLE` for another message or another Video-Out operation.

The TriMedia Video-Out APIs provide the necessary interface for video applications to access the TriMedia Video-Out unit hardware.

Examples of the use of Raw and Message-Passing modes are to be found in the Power On Self Test (POST) sources.

Using the TriMedia Video-In/Video-Out Device Library

The APIs provided in the TriMedia Video-In/Video-Out Device Library enable you to access both the Video-In and Video-Out hardware units of TriMedia. The Video-In/Video-Out device library provides functions for controlling video encoders and decoders. It can be linked with other programs, providing you with total control of the hardware by enabling you to

- Optimize ISRs to meet application requirements.
- Create vendor-specific initialization and configuration routines for on-board chips (such as a decoder that works with the TriMedia Video-In component and an encoder that works with the TriMedia Video-Out component).

Guidelines for Use of the Video-In/Video-Out APIs

General guidelines for using the TriMedia Video-In/Video-Out APIs are as follows:

- Use the archive version (libdev.a), rather than building the library yourself. (The Video-In/Video-Out device library is archived in libdev.a).
The source for the Video-In/ Video-Out device library is included in the TCS. This makes it easier to incorporate new versions of the library as they become available.
- Pass the specific instance when making subsequent calls.
The Video-In/Video-Out Device Library operates as an exclusive device driver, and, as such, can service only one task at a time. This is enforced through the instance identifier, which is returned by all the initialization functions.
- Modify the functions, viOpen() and voOpen() using interfaces provided in the Board Support API.
The viOpen() and voOpen() functions call the initialization routines for the analog I/O hardware on the board. The board library provides support for default boards (For example, the TriMedia debug board and IREF board).
It provides the initialization routine for the decoder on the debug board (SAA7111) and IREF board (SAA7111A), and for the encoder on the debug board (SAA7185) and IREF board (SAA7125).
For more information about the Board Support API, refer to Reference Manual II of the Philips TriMedia SDE.
- Check the error values returned by the initialization functions. Most of the Video library functions return zero on success, or nonzero error codes.
- Use the debug version of the libdev.a library during development. Many functions check and report the use of sizes and alignments that the hardware cannot support.

Vivot Demonstration Program Overview

The vivot example is intended as an example to allow the user to gain familiarity with the techniques necessary to program the video capabilities of the TM-1000 architecture. The program demonstrates how the video input and output modes can be reprogrammed dynamically on Trimedia.

First, the image captured on VI is converted to CIF and a quarter sized image on the middle of the screen (vivoRunCIF).

Next, a full screen image is displayed and captured (vivoRunFullres). These two processes are executed for 1000 frames each.

Finally, a quarter sized image is displayed on the middle of the screen on top of the Trimedia logo (vivoRunOverlay), thus illustrating the overlay feature.

It is useful to know a certain number of “video programming tricks” when using TriMedia; for example, CIF conversion, as well as general device issues on TriMedia (such as buffer alignment, or cache coherency). The data book provides the functionalities but it does not go into detail. The purpose of this is to explain them.

The programmer needs to be aware of a certain number of implementation choices while studying the code. For example, the buffer scheme being used indexes buffers from a table and references them circularly using modulo addressing. Using a linked list of buffers can be preferable. Optimizing the code could reduce the buffer space requirements or execution time. For example, a single buffer can be used for Video-In (VI) and Video-Out (VO). This was not done for clarity reasons. (For example, the overlay buffer is used for the captured frames and the VO buffer for the logo). Busy waiting is used for buffer and ICP processing, instead of a semaphore.

C Program Includes

Most C programs include `<stdio.h>` and `<stdlib.h>`. Trimedia-specific C library functions are in `<tmlib/tmlibc.h>`. The standard C header files such as `<assert.h>` and `<ctype.h>` can be included also. MMIO registers are defined in `<tml/mmio.h>`.

Custom ops are defined in `<ops/custom_defs.h>`. Multimedia formats (such as `vaaNTSC`, `vaaPAL`, `vaaCVBS`, `vaaSvideo`) are defined in `<tml/tmAvFormats.h>`.

A program that uses VO should include `<tml/tmVO.h>` and `<tml/tmVOMmio.h>`. To use the Image Co-Processor (ICP), include `<tml/tmICP.h>`. A program that uses VI should include `<tml/tmVI.h>` and `<tml/tmVImmio.h>`.

To find out the clock speed or the processor type, include `<tml/tmProcessor.h>` (`procGetCapabilities`) and/or `<tml/tmBoard.h>`.

For definitions associated with interrupts, include `<tml/tmInterrupts.h>`. To use the DP debug printing facility of **tmgmon**, include `<tmlib/dprintf.h>`

Main Program

The first line initializes the DP printing facility of **tmgmon**. The next two lines print a start-up message, using DP and `printf`. The call to `ReportSys` is to find out the processor clock frequency and the version of the processor (for work-arounds).

Video out bug 21727 was present in versions of the processor prior to TM1S1.1. Video out bug 3056 is less important but the two exacerbate each other. The call to `vivoDetectworkarounds` detects which of these bugs are present and positions the flag `DummyCode`. We will assume in what follows that `DummyCode` is zero.

The call to `vivoCheckArgcv` sets the adapter type (S-video, composite) and the video standard (PAL or NTSC). The call to `vivoRun` contains the main program.

```
int
main(int argc, char **argv)
{
  SetDP();
  DP((Header));
  printf(Header);
  reportSys();
  vivoDetectworkarounds();

#ifdef __TCS_nohost__
  vivoCheckArgcv(argc, argv);
#endif

  vivoRun();

  exit(0);
}
```

Vivot Demonstration Program (Vivorun)

- Buffer allocation is ensured by `vivoAlloc`.
- `vivoOpenAPI` calls `viOpenAPI` and `voOpenAPI` for API initialization.
- `vivoCloseAPI` frees the buffers (the name is a misnomer).

The code for `vivoRun` is shown below.

```
vivoRun()
{
  vivoAlloc();
  vivoOpenAPI();

  vivoRunCIF();
  vivoRunFullRes();
  vivoRunOverlay();
  vivoCloseAPI();
}
```

Image Representation

The output format is defined by the width, the height, and the stride. The stride is different from the width because lines need not be contiguous and because of alignment of lines to cache boundaries.

The dimensions are 352 x 240 for CIF (`cifwidth`, `cifHeight`) and 720 x 480 (full Width, full Height) for full resolution. The image buffer sizes are 384 x 240 for CIF and 768 x 576 for full resolution.

The strides (`cifStride`, `fullStride`) differ from the widths because VI requires that image lines begin on a cache line (modulo 64 bytes) boundary. The image is represented in planar format using separate Y, U, and V pointers in the `vbuf` structure.

Buffer Allocation (`vivoAlloc`)

`vivoAlloc` calls `allocCif422`, `allocFullres`, and `allocBkBuf` to allocate the CIF, full resolution, and overlay buffers.

Table 1-1 summarizes the buffer allocation in the demonstration program.

- The buffer allocation scheme is fixed and there is no sharing.
- Buffers are addressed via an index modulo the total number (4).
- The buffers are circulated between VI, VO and processing.
- Pointer advancement corresponds to inputs and completion of processing.
- The flag field of the `vbuf` structure identifies the state at any given point in time (`VID_RDY_VI`, `VID_RDY_VO`, `VID_RDY_MM`).
- 5 megabytes of memory are required for the buffers in total.
- The dimensions in full resolution are large enough to contain either a PAL (704 x 576) or NTSC (720 x 480) image.

Table 1-1 Buffer Allocation in Demonstration Program

CIF	384	240	2	4	737280
full res	768	576	2	4	3538944
overlay	768	576	2	4	884736
Total					5160960

Cache Management

Cache coherency between the DSPCPU and the peripheral units is managed in software on the TM-1000. The program contains routines to allocate a buffer, to update the cache to memory, and to remove stale data.

These routines deal with cache lines (blocks whose sizes are a multiple of 64 bytes beginning at a modulo 64 boundary). `allocSz` calls the library routine `_cache_malloc` (Refer to code insert below).

The second parameter indicates the set number from which to allocate (0-31 or -1 if any is acceptable). Refer to Chapter 3 of the Trimedia Software Cookbook for information about the incidence of this on performance.

The pointer returned by `_cache_malloc` begins at a modulo 64 boundary. The size is rounded up also.

```
UInt32
allocSz(int bufSz)
{
    UInt32    temp;
    int       i;

    if ((temp = (UInt32) _cache_malloc(bufSz, -1)) == Null)
        my_abort("_cache_malloc", 0);

    memset(temp, 0, bufSz);
    _cache_copyback(temp, bufSz);;
    return temp;
}
```


viOpenAPI - level 1 initialization for VI

The code for viOpenAPI is shown in two sections below. The call to viOpen acquires the peripheral; accesses to a peripheral have to be exclusive. This returns an “instance” in viInst corresponding to the peripheral.

```
void viOpenAPI()

    tmLibdevErr_t err;

    if (err = viOpen(&viInst))
        my_abort("viOpen", err);
```

The call to viInstanceSetup programs the 7111 and associates the interrupt service routine viTestISR at interrupt priority level 3 with the device.

```
memset((char *) (&viInstSup), 0, sizeof (viInstanceSetup_t));

viInstSup.interruptPriority = intPRIO_3;
viInstSup.isr = viTestISR;
viInstSup.videoStandard = videoStandard;
viInstSup.adapterType = adapterType;

if (err = viInstanceSetup(viInst, &viInstSup))
    my_abort("viInstanceSetup", err);
}
```

voOpenAPI - level 1 initialization for VO

The code for voOpenAPI follows the same general structure as viOpenAPI. An instance is allocated and then setup.

```

void
voOpenAPI()
{
    tmLibdevErr_t err;
    pprocCapabilities_t procCap;

    if (err = voOpen(&voInst))
        my_abort("voOpen", err);

    memset((char *) (&voInstSup), 0, sizeof (voInstanceSetup_t));

    voInstSup.interruptPriority = intPRIO_6;
    voInstSup.isr = voTestISR;
    voInstSup.videoStandard = videoStandard;
    voInstSup.adapterType = adapterType;

    procGetCapabilities(&procCap);
    /* see formula on VO, Figure 7.6 in the data book */
    voInstSup.ddsFrequency = (unsigned int)
        (0.5 + (1431655765.0 * 27000000 / procCap->cpuClockFrequency));

    voInstSup.hbeEnable = True;
    voInstSup.underrunEnable = True;

    if (err = voInstanceSetup(voInst, &voInstSup))
        my_abort("voInstanceSetup", err);
}

```

The interrupt service routine is voTestISR and the interrupt is at level 6. The Highway Bandwidth Error (HBE) interrupt is enabled. This corresponds to VO not getting data from the highway in time to continue transfer. The Underrun interrupt is enabled. This corresponds to the CPU not updating the buffer pointer in time (excessive interrupt latency). For more information on these, refer to section 7.12.3 of the data book.

The initialization of the DDS frequency merits some explanation. Section 7.4 of the data book defines the clock frequency at the output by the following equation:

$$f_{DDS} = \frac{3 \times \text{FREQUENCY} \times f_{DSPCPUCLK}}{2^{32}}$$

The value for fdds is twice the video clock frequency of 13.5 Mhz. The input divider for the clock frequency divides by two (see Table 7-7 of the data book, default values for the

PLL fields in VO_CTL). The frequency of 27 Mhz corresponds for PAL to an image format of 864 pixels, 625 lines, at a 25 Hz frame rate (50 Hz interlaced). The image format parameters for NTSC vary, but the clock frequency is identical. So the value for fdds needs to be 27 Mhz.

The code in voOpen corresponds to a rearrangement of the terms to obtain the ddsfrequency of the equation above.

```
voInstSup.ddsFrequency = (unsigned int)
    (0.5 + (1431655765.0 * 27000000 / (float) procCap-
    >cpuClockFrequency));
```

The number 1,431,655,765 (referenced in the code above) equals

$$\frac{2^{32}}{3}$$

Field Capture versus Frame Capture

The dimensions of image being captured depend on the output resolution. In full resolution, the two fields are assembled together to form a frame. Consecutive lines from different fields are assembled together to form an image by using a stride equal to twice the line stride and setting buffer pointers.

In CIF resolution, the buffer consists of a single field and has half the height of the image. The frame rate for output is the same as in full resolution since one of out two fields is discarded. For the horizontal resolution the HALFRES mode of the Video Out unit is used.

Running in CIF Resolution (vivoRunCIF)

The code for vivoRunCIF begins by initializing the capture buffer pointers.

```
void
vivoRunCIF()
{
    tmLibdevErr_t err;

    printf("\nStarting CIF resolution mode\n");

    cpGenBuf(cif422Buf, VID_NUMBUFS, VID_RDY_VI);
    viNum = mmNum = voNum = 0;
```

The resolution for U and V strides are half that of Y so the stride must be divided by two also.

```
yFieldStride = cifStride;
uvFieldStride = (cifStride >> 1);
overlayFieldStride = 0;
```

Setting `capField` tells `viTestISR` *not* to assemble fields into frames. This has the effect of dividing by two the vertical resolution.

```
capField = True;
firstField = False;
```

The arguments to `viYUVAPI` indicate that the image is being captured starting at line 11, pixel 4, with field capture. The `HALFRES` mode is used, dividing in effect by two the horizontal resolution.

```
viYUVAPI(viHALFRES, cifWidth, cifHeight, cifStride, 1, 4, 11,
         (Pointer) (cif422Buf[0].Y),
         (Pointer) (cif422Buf[0].U),
         (Pointer) (cif422Buf[0].V));
```

The arguments to `voYUVAPI` indicate that the image is offset to line 64, pixel 128. The output format is in 4:2:2 format with cosited sampling for luminance and chrominance.

This corresponds to the format used by VI (CCIF 656 standard). The VO unit has the capacity to upscale the image by two but this is not used.

```
voYUVAPI(vo422_COSITED_UNSCALED, cifWidth, cifHeight, cifStride, 64, 128,
         (Pointer) cif422Buf[0].Y,
         (Pointer) cif422Buf[0].U,
         (Pointer) cif422Buf[0].V);
```

1000 frames are copied from VI to VO. This corresponds to approximately 33 seconds at 60 Hz.

```
for (voISRCCount = 0; voISRCCount < loopCount;) {
    mmBufUpdate();
}
```

After 1000 frames, we shut down image display and capture.

```
if (err = viStop(viInst))
    my_abort("viStop", err);
if (err = voStop(voInst))
    my_abort("voStop", err);
```

Running in Full Resolution (vivoRunFullRes)

In full resolution, two fields are assembled to form a frame together. The global variables `ScanWidth` and `uvScanWidth`. These values are used in the ISR to adjust the buffer pointers for the second field of capture. They correspond to the offset in bytes between fields for the Y and U, V buffers.

```
yScanWidth = fullStride;
uvScanWidth = (fullStride >> 1);
```

This corresponds to forming the frame by reassembling consecutive lines from different fields together. The arguments to `viYUVAPI` indicate that the image is being captured starting at line 21, pixel 0, with a frame mode of capture, (as explained previously).

```
viYUVAPI(viFULLRES, fullWidth, fullHeight, fullStride, 0, 0, 21,
         (Pointer) (fullResBuf[0].Y),
         (Pointer) (fullResBuf[0].U),
         (Pointer) (fullResBuf[0].V));
```

The arguments to `voYUVAPI` correspond to those used in CIF mode except that the image is offset at (0, 0).

```
voYUVAPI(vo422_COSITED_UNSCALED, fullWidth, fullHeight, fullWidth, 0, 0,
         (Pointer) fullResBuf[0].Y,
         (Pointer) fullResBuf[0].U,
         (Pointer) fullResBuf[0].V);
```

Initialization With Alpha Overlay (vivoRunOverlay)

In overlay mode the TM-1 logo is displayed on video out. The image from video in is converted to overlay mode and output.

```
void vivoRunOverlay()
{
    tmLibdevErr_t err;

    printf("\nStarting overlay mode\n");

    cpGenBuf(cif422Buf, VID_NUMBUFS, VID_RDY_VI);
    runningOverlay = True;
    cpUsize = ((cifStride * cifHeight) >> 1);
```

The image is captured starting at line 12 (hex C), pixel 12, with field capture.

```
viYUVAPI(viHALFRES, cifWidth, cifHeight, cifStride, 1,
         0xc,          /* x offset */
         0xc,          /* y offset */
         (Pointer) (cif422Buf[0].Y),
         (Pointer) (cif422Buf[0].U),
         (Pointer) (cif422Buf[0].V));
```

The output format is the same as in full resolution mode.

```
voYUVAPI(vo422_COSITED_UNSCALED, fullWidth, fullHeight, fullWidth, 0, 0,
         (Pointer) fullResBuf[0].Y,
         (Pointer) fullResBuf[0].U,
         (Pointer) fullResBuf[0].V);
```

voOverlayAPI is then called. The buffer pointer corresponds to the first CIF buffer. In overlay mode, the VO buffer points to the TriMedia logo. The overlaid zone is at offset (64, 128) from the left hand corner of the active video area and has the width and height a CIF image (350 x 240).

The stride value of a single alpha value of 64 (50 percent) is used over the entire display image. The CIF buffer Y pointer points to the converted overlaid image. A single pointer is used as the overlaid image is in YVYU format.

The stride of 1408 is four times the width of the image because in the buffer both the even and odd fields and the luminance and chrominance data are interspersed.

```
voOverlayAPI(64, 128, 352, 120, 64, 64, 1408, (Pointer) cif422Buf[0].Y);
```

1000 buffers are copied from VI to VO.

```
for (voISRCCount = 0; voISRCCount < loopCount;) {
    mmOvlyBufUpdate();
}
```

Image capture and display are stopped as previously.

```
if (err = viStop(viInst))
    my_abort("viStop", err);
if (err = voStop(voInst))
    my_abort("voStop", err);
}
```

Setup Input and Begin Capture (viYUVOpenAPI)

The beginning of the code for viYUVAPI is shown below. The VI unit has two interrupt modes corresponding to when a scan line is reached (thresholdReached) and to capture complete (end of an image, beginning of vertical sync interval). The capture mode is used. Cosited sampling is used.

```
void
viYUVAPI(int mode, int width, int height, int stride, int fieldBuf,
         int startx, int starty, Pointer yBase, Pointer uBase, Pointer vBase)
{
    tmLibdevErr_t err;

    memset((char *) (&viYUVSup), 0, sizeof (viYUVSetup_t));

    viYUVSup.thresholdReachedEnable = False;
    viYUVSup.captureCompleteEnable = True;
    viYUVSup.cositedSampling = True;
```

The threshold register is set to line 0. The startx, and startY values correspond to the line and pixel number to begin image capture. The width parameter corresponds to the number of pixels after the starting pixel for line capture

```
viYUVSup.mode = viFULLRES;
viYUVSup.yThreshold = 0;
viYUVSup.startX = startx;
viYUVSup.startY = starty;
viYUVSup.width = width;
```

The next three instructions initialize the VI's units buffers pointers.

```
viYUVSup.yBase = yBase;
viYUVSup.uBase = (DummyCode) ? (Pointer) MMIO(DRAM_BASE) : uBase;
viYUVSup.vBase = vBase;
```

Depending on whether the captured image is in full resolution, or in CIF mode, it must be assembled from fields to frames by the VI ISR.

The first case corresponds to CIF mode. Interlacing is used in this mode to divide the vertical resolution and the second field is eliminated. The delta values for U and V are divided because they have half the resolution.

```
if (fieldBuf) {
    viYUVSup.height = height;
    viYUVSup.yDelta = (stride - width) + 1;
    viYUVSup.uDelta = ((stride - width) >> 1) + 1;
    viYUVSup.vDelta = ((stride - width) >> 1) + 1;
}
```

“Delta” corresponds to the difference between the last pixel of a line and the first pixel of the following line. This corresponds to the difference between “stride” and “width” (the space necessary so that the next line can begin on a mod 64 boundary).

The “+1” comes from the definition of Delta (the pointer stops incrementing at the last pixel). The second case corresponds to full resolution mode. The value for height corresponds to the number of lines in a field (half that of a full image). The extra space of “stride” bytes corresponds to the corresponding line from the other field of the image.

```
else {
    viYUVSup.height = (height >> 1);
    viYUVSup.yDelta = (stride - width) + stride + 1;
    viYUVSup.uDelta = ((stride - width) >> 1) + (stride >> 1) + 1;
    viYUVSup.vDelta = ((stride - width) >> 1) + (stride >> 1) + 1;
}
```

The `viYUVSetup` routine initializes the video parameters.

```
if (err = viYUVSetup(viInst, &viYUVSup))
    my_abort("viYUVSetup", err);
```

The `viStart` routine initializes image capture.

```
if (err = viStart(viInst))
    my_abort("viStart", err);
}
```

Start Outputting an Image To Video Out (voYUVAPI)

The routine begins by initializing the video mode. The VO unit supports three output modes: cosited 4:2:2, interspersed 4:2:2, and 4:2:0 (see section 7-8 of the data book).

In cosited 4:2:2, the chrominance values (U and V) correspond to the first of two luminance values. In interspersed 4:2:2, they correspond to the midpoint between the two pixels. In 4:2:0 mode, there are four times fewer U and V than Y values (half as many as in

4:2:2). The chrominance values correspond to the point in the center of the square formed by consecutive horizontal and vertical pictures.

The VO unit has two interrupt modes. An interrupt can be generated at the end of the image area or when the scan line reaches a given value. The first is used.

```
void
voYUVAPI(voYUVModes_t mode,
         int imageWidth, int imageHeight, int imageStride,
         int imageVertOffset, int imageHorzOffset,
         Pointer yBase, Pointer uBase, Pointer vBase)
{
    tmLibdevErr_t err;

    memset((char *) (&voYUVSup), 0, sizeof (voYUVSetup_t));
    voYUVSup.mode = mode;
    voYUVSup.buf1emptyEnable = True;
    voYUVSup.yThresholdEnable = False;
    voYUVSup.yThreshold = False;
```

The next three lines set the Y, U, and V image pointers.

```
voYUVSup.yBase = yBase;
voYUVSup.uBase = uBase;
voYUVSup.vBase = vBase;
```

The `imageVertOffset` and `imageHorzOffset` correspond to the offset of the image from the top left hand corner of the active video area (see figure 7-12 of the data book).

```
voYUVSup.imageVertOffset = imageVertOffset;
voYUVSup.imageHorzOffset = imageHorzOffset;
```

The image height needs to be divided by two for interlaced scan. Lines of one field are interspersed with lines of another, so the stride needs to be doubled. The stride for U and V is half that for the Y pixels.

```
voYUVSup.imageHeight = (imageHeight >> 1);
voYUVSup.yStride = (2 * imageStride);
voYUVSup.uStride = imageStride;
voYUVSup.vStride = imageStride;
```

The mode supplied to `voYUVSetup` is a combination of the VO mode and the use of 2x horizontal upscaling. The width of the image is halved in the presence of scaling.

```
switch (mode) {
case vo422_COSITED_UNSCALED:
    case vo422_INTERSPERSED_UNSCALED:
    case vo420_UNSCALED:
        voYUVSup.imageWidth = imageWidth;
        break;
case vo422_COSITED_SCALED:
case vo422_INTERSPERSED_SCALED:
case vo420_SCALED:
default:
    voYUVSup.imageWidth = imageWidth << 1;
    break;
}
```

The call to `voYUVSetup` programs the 7185 registers.

```
if (err = voYUVSetup(voInst, &voYUVSup))
    my_abort("voYUVSetup", err);
```

The call to `voStart` begins image display.

```
if (err = voStart(voInst))
    my_abort("voStart", err);
}
```

Initialize Overlay Mode (voOverlayAPI)

voOverlayAPI copies the arguments into a structure and calls voOverlaySetup.

The TM-1000 VO unit allows the display buffer to be overlaid with a raster in memory using alpha blending. Since the TM-1000 uses a raster overlay the user has full control over the contents. For example, the overlay can contain a graphic logo as well as characters for teletext. The dimensions and position of the overlay with respect to the active image area are programmable.

The degree of blending is determined by the top three bits of two eight bit registers (GLOBAL ALPHA 0, GLOBAL ALPHA 1), as indicated in Table 7-4 of the data book. The TM-1000 display buffer has separate Y, U, and V planes but the overlay raster is interspersed. Overlay images are stored in YVYU format. Figure 7-20 of the data book shows the format. The U and V values are the same for the two Y pixels.

The low order bit of U determines the alpha value for (Y0, U, V) (ALPHA 1, ALPHA 0) The low order bit of V determines the alpha value for (Y1, U, V) similarly.

The arguments to voOverlayAPI are the offset of the overlay from the left hand corner (sLine, sPixel), the size of the overlay (width, height), the values for alpha blending (alpha0, alpha1).

Because the overlaid image data is interspersed there is a single buffer pointer and stride (base and offset).

```
voOverlayAPI(int sLine, int sPixel, int width, int height,
             UInt alpha0, UInt alpha1, int offset, Pointer base)
{
    tmLibdevErr_t err;

    memset((char *) (&voOverlaySup), 0, sizeof (voOverlaySetup_t));
    voOverlaySup.overlayEnable = True;
    voOverlaySup.overlayStartY = sLine;
    voOverlaySup.overlayStartX = sPixel;
    voOverlaySup.overlayWidth = width;
    voOverlaySup.overlayHeight = height;
    voOverlaySup.alpha0 = alpha0;
    voOverlaySup.alpha1 = alpha1;
    voOverlaySup.overlayStride = offset;
    voOverlaySup.overlayBase = base;

    if (err = voOverlaySetup(voInst, &voOverlaySup))
        my_abort("voOverlaySetup", err);
}
```

Inputting an Image for Display on VO (readYUVfiles)

There are no VO alignment constraints in image mode.

vivoAlloc calls readYUVfiles to read a 720 x 480 image in "tmlogo" into bkbuf.

```
err = readYUVFiles("tmlogo", 720, 480,
                  bkBuf[0].Y, bkBuf[0].U, bkBuf[0].V);

readYUVFiles(char *baseName, int hsize, int vsize,
             UInt32 ybuf, UInt32 ubuf, UInt32 vbuf)
{
```

```
int          count, ySize, uvSize, row;
char         fn[80];
unsigned char *pb;
FILE        *fp;
```

The size of the UV buffer is half that of the Y buffer after conversion.

```
ySize = hsize * vsize;
uvSize = (ySize >> 1);
```

The Y data is in "tmlogo.y" The file is opened in binary mode.

```
sprintf(fn, "%s.y", baseName);
fp = fopen(fn, "rb");
if (!fp)
    return (4);
```

The data is read into the buffer. For VO lines do not need to be aligned on cache line boundaries.

```
count = fread((char *) ybuf, 1, ySize, fp);
fclose(fp);
```

The data is flushed back to the cache.

```
_cache_copyback(ybuf, ySize);
```

The U data is read from “tmlogo.u” in a similar fashion.

```
sprintf(fn, "%s.u", baseName);
fp = fopen(fn, "rb");
if (!fp)
    return (4);
pb = (unsigned char *) ubuf;
count = 0;
```

There are half as many lines on the file as for the Y data. This is because the data is in 4:2:0 format.

```
for (row = 0; row < (vsize >> 1); row++) {
```

There are half as many pixels per line as for the Y data also.

```
count += fread(pb, 1, (hsize >> 1), fp);
```

The data from the odd field is reproduced for the even field also. The pointer is incremented to point to the next image. The data is flushed back for the cache.

```
memcpy(pb + (hsize >> 1), pb, (hsize >> 1));
pb += hsize;
}
_cache_copyback(ubuf, uvSize);
fclose(fp);
```

The code for reading the V data is the same as for the U data. The function returns zero to indicate successful completion.

```
return (0);
}
```

ICP Setup

A color conversion filter is used to convert the captured image to overlay format. The input image is in CIF with the strides corresponding. The output stride is double the input stride because of 2x upscaling.

ICP setup requires opening an ICP instance (`icpOpen`) and associating an interrupt with it (`icpInstanceSetup`).

```

static void
SetupICP()
{
    tmLibdevErr_t err;

    if (err = icpOpen(&icpInst))
        my_abort("icpOpen", err);
    memset((char *) &icpInstSup, 0, sizeof (icpInstanceSetup_t));
    icpInstSup.interruptPriority = intPRIO_4;
    icpInstSup.isr = NULL;
    if (err = icpInstanceSetup(icpInst, &icpInstSup))
        my_abort("icpInstanceSetup", err);
}

```

The stride for Y is twice that for U and V since the image is in 4:2:2 format. The output stride is double that of the input since we are upscaling horizontally from 352 to 704.

```

memset((char *) &icpImage, 0, sizeof (icpImageColorConversion_t));
icpImage.yInputStride = cifStride;
icpImage.uvInputStride = (cifStride >> 1);
icpImage.inputHeight = cifHeight;
icpImage.inputWidth = cifWidth;
icpImage.outputStride = cifWidth<<1;
icpImage.outputHeight = cifHeight;
icpImage.outputWidth = cifWidth;

```

The `filterBypass` field can be `icpFILTER` or `icpBYPASS`. Bypass mode corresponds to simply picking the nearest pixel in the input for the output.

```
icpImage.filterBypass = icpFILTER;
```

The output is interspersed and the input is planar. The byte ordering is little endian on a Windows host, otherwise it is big endian. Output is to the SDRAM.

```

icpImage.outputPixelOffset = 0;
icpImage.inFormat = vdfYUV422Planar;
icpImage.outputDestination = icpSDRAM;
#ifdef __TCS_Win95__
    icpImage.littleEndian = True;
#else
    icpImage.littleEndian = False;
#endif
icpImage.outFormat = vdfYUV422Sequence;
}

```

Buffer Processing for Full Resolution and CIF

Buffer processing is performed at the main level. The `mmBufUpdate` routine is called to process a captured image. the routine `mmBufUpdate` is used. It is called in the main level busy wait loop. `mmBufUpdate` first checks for a buffer ready.

```
void
mmBufUpdate()
{
    int            mmtmpNum;

    mmtmpNum = (mmNum + 1) % VID_NUMBUFS;
    If the buffer is ready
    if (genBuf[mmtmpNum].flag == VID_RDY_MM)
    {
```

The output buffer is made available for VO and the pointer is advanced to the next buffer. it is made available for VO

```
    genBuf[mmNum].flag = VID_RDY_VO;
    mmNum = mmtmpNum;
}
```

Buffer Processing for Overlay (mmOvlyBufUpdate)

`mmOvlyBufUpdate` first checks for a buffer.

```
void
mmOvlyBufUpdate()
{
    int            mmtmpNum;
    tmLibdevErr_t err;
```

The arguments to the color conversion filter are the Y U and V pointers of the input buffer.

```
    mmtmpNum = (mmNum + 1) % VID_NUMBUFS;
    if (genBuf[mmtmpNum].flag == VID_RDY_MM) {
```

The input image in buffer `mmNum+1` is in planar format.

```
    icpImage.yBase = (Pointer)genBuf[mmtmpNum].Y;
    icpImage.uBase = (Pointer)genBuf[mmtmpNum].U;
    icpImage.vBase = (Pointer)genBuf[mmtmpNum].V;
```

The output image in buffer `mmNum` is in interspersed format. A color conversion filter is used to convert. A busy wait loop is used to check for termination.

```
icpImage.outputImage = (Pointer)genBuf[mmNum].Y;
    if (err = icpColorConversion(icpInst, &icpImage))
        my_abort("icpColorConversion", err);
    while (icpCheckBUSY());
```

The output buffer is made available for VO and the pointer is advanced to the next buffer.

```
genBuf[mmNum].flag = VID_RDY_VO;
    mmNum = mmtmpNum;
}
}
```

VI Interrupt Service Routine (viTestISR)

The code has been edited to remove work-arounds for bugs for clarity. The function of `viTestISR` is to position a captured frame in the circular queue as ready for processing (`VID_RDY_MM`) as long as there is a buffer available for capture.

```
void
viTestISR()
{
    unsigned long vi_status = MMIO(VI_STATUS);
    int          oddField;
    int          vitmpNum;
```

The VI interrupt service routine is a non interruptible handler.

```
#pragma TCS_handler
```

We determine whether the field is an even or an `oddField`.

```
oddField = viExtractODD(vi_status);
```

Potential interrupt sources include capture complete, under run, and highway bandwidth error. If this is a highway bandwidth error, we return without doing anything.

```
if (viHBE(vi_status)) {
    viAckHBE_ACK();
    return;
}
```


capField corresponds to CIF capture and overlay. If capField is non zero the even field is eliminated, effectively dividing by two the vertical resolution. The buffer pointer is advanced on reception of the odd field as long as there is an available buffer.

```

if (capField) {
    vitmpNum = (viNum + 1) % VID_NUMBUFS;
    if (oddField & (genBuf[vitmpNum].flag == VID_RDY_VI)) {
        genBuf[viNum].flag = VID_RDY_MM;
        viNum = vitmpNum;
        viYUVChangeBuffer(viInst,
                           genBuf[viNum].Y,
                           genBuf[viNum].U,
                           genBuf[viNum].V);
    }
}

```

The captured buffer is acknowledged terminating interrupt processing.

```

viAckCAP_ACK();
    return;
}

```

The rest of the routine corresponds to capField being zero. The code depends on whether we are processing the first (odd) or second (even) field of a frame. The following code corresponds to the case of an odd field.

```

if (firstField) {

```

The field flag is toggled.

```

    firstField = False;

```

The if corresponds to a dropped field, which is an exception. This is the case if firstField and oddField differ. If this is so, the field is dropped, synchronizing the VI unit and the software.

```

if (!oddField) {
    /* skip even field to get sync */
    firstField = True;
} else {

```

The buffer pointers for the odd and even fields have a separation of one scan line.

```

/* always start with odd field */
    viYUVChangeBuffer(viInst,
                      genBuf[viNum].Y + yScanWidth,
                      genBuf[viNum].U + uvScanWidth,
                      genBuf[viNum].V + uvScanWidth);
    }
}

```

The else case corresponds to the case of an even field. The buffer pointer is advanced if there is an available buffer.

```

else {
    vitmpNum = (viNum + 1) % VID_NUMBUFS;
    if (genBuf[vitmpNum].flag == VID_RDY_VI) {
        genBuf[viNum].flag = VID_RDY_MM;
        viNum = vitmpNum;
    }
}

```

The buffer pointers are reset to the beginning of the buffer

```

viYUVChangeBuffer(viInst,
                  genBuf[viNum].Y,
                  genBuf[viNum].U,
                  genBuf[viNum].V);

```

The field flag is toggled.

```

firstField = 1;
}

```

The capture is acknowledged ending interrupt processing. The video out interrupt source routine is similar to the one explained in the ICP example.

```

viAckCAP_ACK();
}

```

The video out interrupt service routine is similar to the one explained in the ICP example.

Querying the Configuration

Reportsys calls the HAL functionality `procGetCapabilities` to identify the processor type. `procGetCapabilities(&procCap)`; The following structure is returned.

The fields of the data structure identify the processor type (TM1000, TM1100), the processor version (TM1000, TM1100), the revision ID, and the clock frequency in hertz.

By Trimedia API convention, there are two types that are defined (for the structure and for a pointer); the version number is the first word of each structure. The last three fields identify the type of host and the processor configuration, for a multiprocessor.

```
typedef struct
{
    tmVersion_t    version;           /* version of this sw module */

    procDevice_t   deviceID;          /* for implemented functionality */
    procRevision_t revisionID;        /* for bugs, performance, etc. */
    UInt32         cpuClockFrequency; /* in Hz */
    UInt32         nodeNumber;        /* node number in case of multiple
TMs */
    UInt32         numberOfNodes;      /* number of TMs in system */
    tmHostType_t   hostID;            /* tmInvalidHost, tmNoHost,
tmTmSimHost,
                                     tmWin32Host, or tmMacOSHost */
} procCapabilities_t, *pprocCapabilities_t;
```

This terminates the vivot example. Examples of how to use VI and VO in raw and message-passing modes are available in the Power on Self Test (POST). The following chapter contains more information on how to use the video units with the ICP and VGA cards.

Chapter 2

Programming TriMedia Video Applications Using the ICP TSSA API

Topic	Page
Introduction	2-2
The exoIVtransICP Example Program	2-3

Introduction

This chapter describes how to write video applications using the ICP-based Video Transformer. For a detailed description of this API, refer to Chapter 15-1 “TriMedia Image Co-Processor (ICP) API” of Reference Manual II.

The Video Transformer is designed to simplify the use of the Image Co-Processor (ICP) peripheral. This component offers a number of advantages over the tmICP device library. Several tasks may each open an instance of the Video Transformer and issue requests for video filtering; the component library will queue up the requests and issue them one by one to the ICP. The required vertical, horizontal, and color conversion filter operations to perform a transformation are automatically calculated and issued to the ICP. All buffers required to store scaled intermediate images are created and destroyed automatically. The component also supports antiflicker filtering for DSPCPU generated graphics and deinterlacing for interlaced to progressive scan conversion.

The AL layer supports non-data streaming (*push* mode), while the OL layer supports data streaming (*pull* mode).

The exoVtransICP Example Program

The exoVtransICP example demonstrates the use of the OL layer of the Video Transformer. The example simply connects an instance of the Video Digitizer to an instance of the Video Transformer. The digitizer captures live data using the video-in device while the transformer scales the image, converts it from YUV to RGB, and then displays it on the PC screen via the PCI interface. The user may specify parameters on the console to enable antiflicker filtering and deinterlacing.

The source code for this example is contained within the `examples/exoVtransICP` directory of the application tree. The example will now be described, with emphasis placed on the Video Transformer aspects. We recommend that you first read Chapter 1 “Programming TriMedia Video Applications” as it describes the use of the Video Digitizer. Chapter 10 “TriMedia Video Transformer API” of Reference Manual II, Part 2 provides additional information on this example and a separate AL layer example (`examples/exalVtransICP`).

Include Files

```
#include <tml/tmAvFormats.h>
#include "tmos.h"
#include "tmolVtransICP.h"
#include "tmolVdigVI.h"

#include <stdio.h>
#include <tmlib/dprintf.h>      /* for debugging with DP(()) */

#include "sys_conf.h"
```

The `tmAvFormats.h` file contains the definitions for the packets which are used to store video data. The type definitions and function prototypes for the Video Transformer are defined in `tmolVtransICP.h`.

Definitions

```
#define VIDEO_ADDR 0xe0000000 /* Default Start address of the screen */
#define VIDEO_STRIDE 2048 /* For 24 bit video it is 3x screen width */
#define VIDEO_MODE 3 /* RGB15+Alpha */

/*
 * video in image format
 */
#define INPUT_HEIGHT 480
#define INPUT_WIDTH 720
#define INPUT_STRIDE 768

/*
 * Video out image format
 */
#define OUTPUT_HEIGHT 360
#define OUTPUT_WIDTH 540
#define OUTPUT_STRIDE VIDEO_STRIDE
```

The default address of the PCI video card, the stride of the video card, and the RGB format are defined. Note that these are simply the default parameters—the user must specify the correct parameters via the command line.

The height, width, and stride of the captured image are defined using the `INPUT_HEIGHT`, `INPUT_WIDTH`, and `INPUT_STRIDE` respectively. Note that the stride must be a multiple of 64 bytes. This is a requirement of the ICP device when it is performing any vertical filtering (vertical scaling, deinterlace, or antiflicker).

The `OUTPUT_WIDTH` and `OUTPUT_HEIGHT` specify the size of the image which will be displayed on the PC screen. The `OUTPUT_STRIDE` will be equal to the stride of the PCI video card.

Static Variables

```

/*
 * These command line args come from the modified sysinit.c
 * which allows the task to read the required parameters.
 */
extern int __argc;
extern char **__argv;

/*
 * ----- locals -----
 */
static tmVideoAnalogStandard_t vidStd      = vasNTSC;
static tmVideoAnalogAdapter_t vidAdapter = vaaCVBS;
static int framesPerSecond = 30;

```

The `__argc` and `__argv` variables are used to pass command line arguments to the application. These arguments will consist of the PCI video card address, the display stride, and the display RGB format.

The video standard is stored in the `vidStd` variable, which is NTSC by default. This may be changed to `vasPAL` for PAL cameras.

The `vidAdaptor` variable stores the video adaptor type and may be either `vaaCVBS` or `vaaSvideo`.

Finally, `framesPerSecond` specifies the capture rate used by the video digitizer.

Specifying the Packet Format

```

static tmVideoFormat_t digitizerFormat = {
sizeof(tmVideoFormat_t), /* size      */
0,                        /* hash     */
0,                        /* referenceCount */
avdcVideo,               /* dataClass */
vtfYUV,                  /* dataType  */
vdfYUV422Planar,         /* dataSubtype */
vdfInterlaced,           /* description */
INPUT_WIDTH,             /* imageWidth; */
INPUT_HEIGHT,            /* imageHeight; */
INPUT_STRIDE,            /* imageStride; */
};

```

This structure defines the format of the packets used to transfer data between the Video Digitizer and Video Transformer. The `hash` and `referenceCount` fields are used exclusively by the format manager, and must be set to zero.

The `dataClass` and `dataSubtype` must be set to `avdcVideo` and `vtfYUV` respectively. The `dataSubtype` may be set to either `vdfYUV422Planer` or `vdfYUV420Planer`. For this example, YUV422 video is used.

The `description` field specifies that the data stored in the packet buffers is interlaced. The even and odd fields are stored in the same buffer using an interleaved format.

Finally, the captured frame height, width, and stride are defined.

Specifying the Output Format

```
static tmVideoFormat_t outputFormat = {
    sizeof(tmVideoFormat_t), /* size          */
    0,                       /* hash         */
    0,                       /* referenceCount */
    avdcVideo,              /* dataClass    */
    vtfRGB,                 /* dataType     */
    vdfRGB15Alpha,         /* dataSubtype  */
    0,                     /* description  */
    OUTPUT_WIDTH,         /* imageWidth;  */
    OUTPUT_HEIGHT,       /* imageHeight; */
    OUTPUT_STRIDE,       /* imageStride; */
};
```

The `outputFormat` structure specifies the format of the Video Transformer output. The component is capable of writing its output to either SDRAM or PCI. For output to SDRAM, the processed data will be placed in a packet. For output to PCI, the data will be stored in the PCI video card memory and no packet will be used. In either case, the output format must be specified using a `tmVideoFormat_t` structure.

The `dataClass` field must always be set to `avdcVideo`. The `dataType` field may be either `vtfYUV` or `vtfRGB` when the output is to SDRAM. When writing to PCI, the output must be `vtfRGB`.

The `dataSubtype` depends upon the `dataType` field. For YUV data it can be `vdfYUV422Planer`, `vdfYUV420Planer`, `vdfYUV422Sequence`, or `vdfYUV422SequenceAlpha`. For RGB, it can be `vdfRGB8A_233`, `vdfRGB8R_332`, `vdfRGB15Alpha`, `vdfRGB16`, `vdfRGB24`, or `vdfRGB24Alpha`. As the Video Transformer will be writing to the PC screen, the output must be RGB. The subtype will be specified by the user via the command arguments.

The `description` field is set to zero as the component does not use this value on its output.

Finally, the output height, width, and stride are specified.

Packet Defines and Function Prototypes

```

#define          NUMPACKETS          4
#define          NUMBUFFERS          3   /* Y, U, V */

/*
 * ----- function prototypes -----
 */

extern void
get_parameters(Int argc, Char * argv[],
               Int * disp_addr, Int * stride, Int * mode);

extern tmLibappErr_t
tmaIVtransICPProgress(Int instId, UInt32 flags, ptsaProgressArgs_t args);

extern tmLibappErr_t
tmaIVtransICPCompletion(Int instId, UInt32 flags,
                        ptsaCompletionArgs_t args);

```

The example uses four packets (`NUMPACKETS`) to exchange video frames between the digitizer and transformer. Each packet contains three buffers (`NUMBUFFERS`) which store the Y, U, and V data.

The `get_parameters()` function is used to obtain the user-specified command line arguments. This function will not be described.

The `tmaIVtransICPProgress()` and `tmaIVtransICPCompletion()` callback functions will be used by the Video Transformer to report information to the application. These will be described later.

Variables

```

void tmosMain()
{
    tmLibappErr_t    rval;
    Int digitizerInstance;
    ptmolVdigVIInstanceSetup_t digitizerInstSetup;

    Int              vtrans0Instance;
    ptmolVtransICPInstanceSetup_t vtransInstSetup;
    ptsaControlDescriptor_t    vtransCommand;
    tsaControlDescriptorSetup_t csetup;

    ptsaInOutDescriptor_t    iodesc;
    ptsaInOutDescriptorSetup_t ioSetup;

    ptmolVdigVICapabilities_t digitizerCap;
    ptmolVtransICPCapabilities_t vtransCap;

    char    ins[80];
    Int     pciAddress;
    Int     pciStride;
    Int     OutputFormat;
    Int     videoMode;

    tsaControlArgs_t controlArgs;
    Bool    quitDetected = False;
    Bool    antiflickerEnable = False;
    Bool    deinterlaceEnable = False;

```

The `rval` variable is used to store the values returned by calls to the Video Digitizer, Video Transformer, and `tsaDefaults` library. Error values are defined in `tmLibappErr.h`, with a return value of `TMLIBAPP_OK` indicating no error.

The `digitizerInstance` variable is used to store the instance id of the Video Digitizer, while `digitizerInstSetup` is a pointer to the setup structure which will be used to configure the digitizer.

The `vtrans0Instance` variable will be used to store the instance id of the Video Transformer. The component enables up to four instances to be open. The `vtransInstSetup` variable points to the component's setup structure and will be used to configure the instance. The `vtransCommand` variable is a pointer to a control descriptor. The Video Transformer allows the application to send configuration commands to it while it is streaming data. The control descriptor is used to specify the message interface between the application and the instance of the transformer. The `csetup` structure specifies parameters that are used when the control descriptor is created.

The connection between the Video Digitizer and Video Transformer is specified using a `tsaInOutDescriptor`. This describes the connection and the packets that will be used to transfer data.

The capabilities of the two components will be pointed to using `digitizerCap` and `vtransCap`. These will be used by the format manager to ensure that the two components can communicate with each other.

The `ins[80]` char array is used to store character commands entered by the user.

The `pciAddress`, `pciStride`, `OutputFormat`, and `videoMode` are used to store information concerning the PCI video card. These will be initialized via the command arguments.

The `controlArgs` structure is used to pass control information from the application to the component instance. This will be described in more detail in the section “User Input” beginning on page 2-17.

Finally, the `quitDetected`, `antiflickerEnable`, `deinterlaceEnable` are boolean flags. The `quitDetected` flag is used to indicate that the user has typed an exit command. The `antiflickerEnable` and `deinterlaceEnable` are flags that indicate whether the antiflicker filter and deinterlace filter are enabled.

Initialization

```

DPmode(DP_PERSIST);
DPsize(1024*1024);

tmosInit();

printf("TriMedia OS Video Transformer Demo. v1.0\n");
printf("\nThis program uses the video digitizer and video
transformer\n");
printf("to pass video in to the PCI video.\n");
printf("The program is compiled to support NTSC and CVBS.\n");
printf("Recompile to change this.\n\n");

/*
 * get parameters from the command line
 */

get_parameters(__argc, __argv, &pciAddress, &pciStride, &videoMode);

if (videoMode == 1)
    outputFormat.dataSubtype = vdfRGB24Alpha;
else if (videoMode == 2)
    outputFormat.dataSubtype = vdfRGB24;
else if (videoMode == 3)
    outputFormat.dataSubtype = vdfRGB15Alpha;
else if (videoMode == 4)
    outputFormat.dataSubtype = vdfRGB16;

outputFormat.imageStride = pciStride;

```

The `DPmode()` and `DPsize()` functions are used to specify the debug print buffer. This buffer facilitates debugging and stores information that is written to it by either the application or the component instances.

The `tmosInit()` function will initialize the multi-tasking operating system. In this example, the application executes in the default task, while a separate task will be created automatically for the Video Transformer instance. The Video Digitizer is an interrupt-based component and, therefore, does not have a separate task.

The command line parameters are read from arguments passed down to the example program. The user must specify the PCI video address, the PCI stride, and the PCI screen mode. The user will enter the screen mode as a value from one to four and this is re-mapped to the corresponding `tmAvFormat_t` type.

Get Capabilities

```
printf("Getting VdigVI Capabilities\n");
if(rval = tmoLVdigVIGetCapabilities(&digitizerCap)) {
    printf("Error in tmoLVdigVIGetCapabilities: 0x%x\n",rval);
    tmosExit(-1);
}
printf("Getting VtransICP Capabilities\n");
if(rval = tmoLVtransICPGetCapabilities(&vtransCap)) {
    printf("Error in tmoLVtransICPGetCapabilities: 0x%x\n",rval);
    tmosExit(-1);
}
```

The capabilities of the two components must be obtained before the `tsaInOutDescriptor` is created. This information will be used to ensure that they are compatible.

Make the Connection Between the Two Components

```

ioSetup = (tsaInOutDescriptorSetup_t)
    malloc(sizeof(tsaInOutDescriptorSetup_t)
        +(NUMBUFFERS-1)*sizeof(UInt32));
ioSetup->format          = (ptmAvFormat_t)(&digitizerFormat);
ioSetup->flags           = tsaIODescSetupFlagCacheMalloc;
ioSetup->fullQName       = "VDF0";
ioSetup->emptyQName      = "VDE0";
ioSetup->queueFlags      = tmosQueueFlagsStandard;
ioSetup->senderCap       = digitizerCap->defaultCapabilities;
ioSetup->receiverCap     = vtransCap->defaultCapabilities;
ioSetup->senderIndex     = VDIGVI_MAIN_OUTPUT;
ioSetup->receiverIndex   = VTRANSICP_MAIN_INPUT;
ioSetup->packetBase      = 0x100;
ioSetup->numberOfPackets = NUMPACKETS;
ioSetup->numberOfBuffers = NUMBUFFERS;
ioSetup->bufSize[0] = INPUT_HEIGHT * INPUT_STRIDE;      /* Y */
ioSetup->bufSize[1] = INPUT_HEIGHT * INPUT_STRIDE / 2; /* U */
ioSetup->bufSize[2] = INPUT_HEIGHT * INPUT_STRIDE / 2; /* V */

/*
 * Create InOutDescriptor
 */
printf("Creating InOutDescriptor\n");
if(rval = tsaDefaultInOutDescriptorCreate(&iodesc, ioSetup)) {
    printf("Error in tsaDefaultInOutDescriptorCreate: 0x%x\n",rval);
    tmosExit(-1);
}

```

A `tsaInOutDescriptor` setup structure is created and initialized. This is similar to the connection setup described in the section “Make the Connection Between the Two Components” described in Chapter 1. The difference being that the Video Transformer capabilities are passed as the `receiverCap`. Note that the packets that will be placed in the empty queue will have an id beginning with `0x100`; i.e. the four packets will have the following id’s: `0x100`, `0x101`, `0x102`, and `0x103`.

Create the Video Transformer Control Descriptor

```

csetup.commandQName = "vt0C";
csetup.responseQName = "vt0R";
csetup.queueFlags = tmosQueueFlagsStandard;
csetup.flags = 0;
if(rval = tsaDefaultControlDescriptorCreate(&vtransCommand, &csetup)) {
    tmAssert((rval == TMLIBAPP_OK), rval);
}

```

The application may send configuration commands to the Video Transformer using a control descriptor. A setup structure must first be initialized before the descriptor is created. The `commandQName` and `responseQName` fields specify a four letter name which will be associated with the command and response queues; this may be used for debugging purposes. The `queueFlags` specify information used for message queue creation. The `tmolQueueFlagsStandard` flags specify that the queues will be local to the processor, and there is no limit to the number of messages which can be placed on them. The `flags` field is currently unused and should be set to zero.

The `tsaDefaultControlDescriptorCreate()` function will allocate memory for the control descriptor, initialize the relevant values, and create the message queues.

Setup the Video Digitizer

```

/*
 * setup video input digitizer
 */
rval = tmolVdigVIOpen(&digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
rval = tmolVdigVIGetInstanceSetup(digitizerInstance,
                                  &digitizerInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);

digitizerInstSetup->instSetup->outputDescriptors[VDIGVI_MAIN_OUTPUT] =
    iodesc;

rval = tmolVdigVIInstanceSetup(digitizerInstance, digitizerInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer initialized.\n");

```

An instance of the Video Digitizer is first opened. It is important that the application check the return value of this function. A typical error would be

`TMLIBAPP_ERR_MODULE_IN_USE`, which indicates that another task has already opened an instance. The digitizer supports only a single instance.

The `tmolVdigVIGetInstanceSetup()` function should be called to obtain a pointer to the instance setup structure. This will be used to configure the instance. The output descriptor is set to point to the `InOutDescriptor` created previously.

Finally, the `tmolVdigVIInstanceSetup()` function is called to configure the instance.

Setup the Video Transformer

```

rval = tmlVtransICPOpen(&vtrans0Instance);
tmAssert((rval == TMLIBAPP_OK), rval);

rval = tmlVtransICPGetInstanceSetup(vtrans0Instance,
                                     &vtransInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);

/*
 * Queues have to be initialized. We are using only the main input, but
 * no overlay inputs. As we are using the PCI for output we have no
 * output queue/pin. By default, unused pins will be set to Null.
 */
vtransInstSetup->defaultSetup->inputDescriptors[VTRANSICP_MAIN_INPUT] =
    iodesc;

vtransInstSetup->defaultSetup->controlDescriptor = vtransCommand;

vtransInstSetup->defaultSetup->progressFunc = tmlVtransICPProgress;
vtransInstSetup->defaultSetup->completionFunc =
    tmlVtransICPCompletion;

/*
 * setup the PCI output image parameters
 */
vtransInstSetup->outputFormat = outputFormat;

vtransInstSetup->outputDest = tmlVtransICPPPCI;
vtransInstSetup->outputPCIAddr = (UInt8 *) pciAddress;

vtransInstSetup->deinterlaceEnable = False;
vtransInstSetup->antiflickerEnable = False;

rval = tmlVtransICPInstanceSetup(vtrans0Instance, vtransInstSetup);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("transformer instance 0 initialized.\n");

```

An instance of the Video Transformer is opened. Up to four instances may be open at any instant of time. The instance setup structure is obtained by calling `tmlVtransICPGetInstanceSetup()` and this is used to specify the initial configuration. The main image input descriptor is set to the `InoutDescriptor` that was created before. In the example, the overlay input is not used and, therefore, no setup information is specified. The `controlDescriptor` is initialized with the control descriptor.

The `progressFunc` and `completionFunc` callback functions are set to point to functions contained within the example program. The Video Transformer will call the application's progress function when an image transformation request has been placed on the ICP queue. The completion function will be called once the transformation request has been processed. These callback functions are optional.

The output parameters specify the output format and the destination of the video transformation. In this example, the output is to PCI, so it is necessary for the application to specify the `outputDestination` as `tmalVtransICPPCI`, and the `outputPCIAddr` to the address of the PCI video memory. It is also necessary to initialize the output format as this specifies the image output parameters.

If the output was to SDRAM, then an `InOutDescriptor` must be created which connects the output of the Video Transformer to the input of another component. The `outputDestination` should be set to `tmalVtransICPSPDRAM`, with the `outputPCIAddr` and `outputFormat` set to `Null`. In this mode, the instance will obtain the output format from the output descriptor.

The `deinterlaceEnable` and `antiflickerEnable` flags are set to disabled for the initial configuration.

Finally, the `tmolVtransICPInstanceSetup()` function is called to transfer the setup parameters to the instance.

Starting the Component Instances

```
DP("\nStarting Video transformer\n");
rval = tmolVtransICPStart(vtrans0Instance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("transformer started.\n");

DP("\nStarting Video Digitizer\n");
rval = tmolVdigVISTart(digitizerInstance);
tmAssert((rval == TMLIBAPP_OK), rval);
printf("digitizer started.\n");
```

Data streaming is initiated by calling the start function for the two instances. These functions are `tmolVdigVISTart()` and `tmolVtransICPStart()` respectively. The Video Digitizer executes entirely in an interrupt service routine, while the Video Transformer instance executes within its own task.

User Input

```
printf("\nVideo transformer demo started.\n");
printf("Video input is being echoed to video output.\n");
printf("\nThe following commands are available:\n");
printf("\tA - toggle antiflicker filter\n");
printf("\tD - toggle deinterlace filter\n");
printf("\tI - disable both antiflicker and deinterlace filters\n");
printf("\tQ - quit\n");

printf("Press return after entering the required option \n");

while (!quitDetected) {
    gets(ins);
    switch(ins[0]) {
```

The user may enter commands via the console to control the operation of the Video Transformer. The 'A' key will toggle the antiflicker filter, the 'D' key will toggle the deinterlace filter, and the 'I' key will disable the antiflicker and deinterlace filters if they are enabled. The 'Q' key will cause the program to exit.

The input parsing uses a simple switch statement to interpret the commands.

```

case 'a':
case 'A':
    /*
     * disable the deinterlace if it is enabled
     */
    if (deinterlaceEnable) {
        deinterlaceEnable = False;
        controlArgs.command = VTRANS_CONFIG_DEINTERLACE_ENABLE;
        controlArgs.parameter = (Pointer) &deinterlaceEnable;
        controlArgs.timeout = 0;
        rval = tmolVtransICPInstanceConfig(vtrans0Instance,
                                           tsaControlWait,
                                           &controlArgs);

        tmAssert(rval == TMLIBAPP_OK, rval);
        printf("Disabled Deinterlace filter\n");
    }

    antiflickerEnable ^= 1;
    controlArgs.command = VTRANS_CONFIG_ANTIFLICKER_ENABLE;
    controlArgs.parameter = (Pointer) &antiflickerEnable;
    controlArgs.timeout = 0;
    rval = tmolVtransICPInstanceConfig(vtrans0Instance, tsaControlWait,
                                       &controlArgs);
    tmAssert(rval == TMLIBAPP_OK, rval);

    if (antiflickerEnable) {
        printf("Enabled antiflicker filter\n");
    }
    else {
        printf("Disabled antiflicker filter\n");
    }
    break;

```

When the antiflicker key is entered, a check is made to see if the deinterlace filter is enabled. If it is, the application will disable it. Note that this is not a restriction of the Video Transformer component, which is able to do both deinterlacing and antiflicker filtering—it is simply made to be mutually exclusive in the application. Deinterlacing will be disabled by setting up a `controlArgs` structure with the relevant command and command parameter. The command is `VTRANS_CONFIG_DEINTERLACE_ENABLE` in this case, and the parameter will be the `deinterlaceEnable` flag, which was set to false. The timeout field specifies the time the configuration function should wait before returning a time-out error. In this case, the value of zero indicates that the function should wait until it receives a response. It then calls the `tmolVtransICPInstanceConfig()` function to perform the configuration.

The `antiflickerEnable` flag is toggled, and the control arguments structure initialized. The command field is set to `VTRANS_CONFIG_ANTIFLICKER_ENABLE` and a call is made to `tmolVtransICPInstanceConfig()`.

```

case 'd':
case 'D':
    /*
     * Disable antiflicker if it is enabled
     */
    if (antiflickerEnable) {
        antiflickerEnable = False;
        controlArgs.command = VTRANS_CONFIG_ANTIFLICKER_ENABLE;
        controlArgs.parameter = (Pointer) &antiflickerEnable;
        controlArgs.timeout = 0;
        rval = tmolVtransICPInstanceConfig(vtrans0Instance,
                                           tsaControlWait,
                                           &controlArgs);

        tmAssert(rval == TMLIBAPP_OK, rval);
        printf("Disabled antiflicker filter\n");
    }

    deinterlaceEnable ^= 1;
    controlArgs.command = VTRANS_CONFIG_DEINTERLACE_ENABLE;
    controlArgs.parameter = (Pointer) &deinterlaceEnable;
    controlArgs.timeout = 0;
    rval = tmolVtransICPInstanceConfig(vtrans0Instance, tsaControlWait,
                                       &controlArgs);
    tmAssert(rval == TMLIBAPP_OK, rval);

    if (deinterlaceEnable) {
        printf("Enabled Deinterlace filter\n");
    }
    else {
        printf("Disabled Deinterlace filter\n");
    }
    break;

```

The deinterlace toggle operates in a similarly to the antiflicker toggle. If the antiflicker filter is enabled, it is switched off using the `tmolVtransICPInstanceConfig()` function.

The deinterlace enable flag is then toggled and the `VTRANS_CONFIG_DEINTERLACE_ENABLE` command is sent to the transformer instance.

```

case 'i':
case 'I':
    /*
     * Disable antiflicker and deinterlace (ie. display interlaced)
     */
    antiflickerEnable = False;
    controlArgs.command = VTRANS_CONFIG_ANTIFLICKER_ENABLE;
    controlArgs.parameter = (Pointer) &antiflickerEnable;
    controlArgs.timeout = 0;
    rval = tmolVtransICPInstanceConfig(vtrans0Instance, tsaControlWait,
                                       &controlArgs);
    tmAssert(rval == TMLIBAPP_OK, rval);
    printf("Disabled antiflicker filter\n");

    deinterlaceEnable = False;
    controlArgs.command = VTRANS_CONFIG_DEINTERLACE_ENABLE;
    controlArgs.parameter = (Pointer) &deinterlaceEnable;
    controlArgs.timeout = 0;
    rval = tmolVtransICPInstanceConfig(vtrans0Instance, tsaControlWait,
                                       &controlArgs);
    tmAssert(rval == TMLIBAPP_OK, rval);
    printf("Disabled Deinterlace filter\n");
    break;

```

The 'interlace' command simply switches off both the antiflicker and deinterlace filters using two calls to the `tmolVtransICPInstanceConfig()` function. In this mode, the Video Transformer instance will only perform scaling and color conversion on the captured video frames.

```

case 'q':
case 'Q':
    DP(("User requested to quit the example\n"));
    quitDetected = True;
    break;

default:
    break;
}
}

```

Once the user enters the quit command from the console, the `quitDetected` flag will be set, which causes the main while loop to be exited.

Stop and Shutdown

```

printf("\nStopping video transformer instance 0\n");
DP("\nStopping video transformer instance 0\n");
rval = tmolVtransICPStop(vtrans0Instance);
tmAssert(rval == TMLIBAPP_OK, rval);

printf("\nStopping video digitiser\n");
DP("\nStopping video digitiser\n");
rval = tmolVdigVISTop(digitizerInstance);
tmAssert(rval == TMLIBAPP_OK, rval);

tmolVdigVIClose(digitizerInstance);
rval = tmolVtransICPClose(vtrans0Instance);
tmAssert(rval == TMLIBAPP_OK, rval);

/*
 * Check we have the correct number of packets left in the queues
 */
rval = tsaDefaultCheckQueues(iodesc);
printf("tsaDefaultCheckQueues() returned 0x%x\n", rval);

/*
 * Destroy InOutDescriptors and command queues
 */
printf("Destroying InOutDescriptors\n");
if(rval = tsaDefaultInOutDescriptorDestroy(iodesc)) {
    printf("Error in tsaDefaultInOutDescriptorDestroy: 0x%x\n",rval);
    tmosExit(-1);
}

rval = tsaDefaultControlDescriptorDestroy(vtransCommand);
tmAssert(rval == 0, rval);

DP("Demo Complete.\n");
printf("Demo Complete. \n");
tmosExit(0);
}

```

Data streaming is terminated by calling the stop functions of each component. When `tmolVtransICPStop()` is called, the Video Transformer instance will return any packets it may have in its possession, it then calls the completion function, and suspends its task. When `tmolVdigVISTop()` is called, it will stop video capture and return the packet that it had in its possession.

The two instances are then closed by calling `tmolVdigVIClose()` and `tmolVtransICPClose()` respectively. Closing the Video Transformer will destroy the transformer instances task.

We recommend that the application call the `tsaDefaultCheckQueues()` function to ensure that the correct number of packets has been left in the InOutDescriptor.

Calling `tsaDefaultInOutDescriptorDestroy()` will remove all packets from the descriptor queues, free up their data buffers, and free the space allocated for the descriptor.

Finally, the Video Transformer control descriptor should be destroyed by calling the `tsaDefaultControlDescriptorDestory()` function.

Application Progress Function

```
tmLibappErr_t
tmalVtransICPProgress(Int instId, UInt32 flags, ptsaProgressArgs_t args)
{
    DP("tmalVtransICPProgress[%x]: inside callback!\n", instId);
    return (TMLIBAPP_OK);
}
```

The application may supply a progress function to the Video Transformer instance. This function will be called by the Video Transformer once a packet has been placed on the ICP request queue. The example progress function simply prints a message to the DP debug buffer.

Application Completion Function

```
tmLibappErr_t
tmalVtransICPCompletion(Int instId, UInt32 flags,
                        ptsaCompletionArgs_t args)
{
    DP("tmalVtransICPCompletion[%x]: inside callback!\n", instId);
    return (TMLIBAPP_OK);
}
```

The application may supply a completion function to the transformer instance. This function will be called once a frame has been processed by the ICP. It will also be called after the instance has been asked to stop. The example completion function prints a message to the DP debug buffer.

Chapter 3

Programming TriMedia Audio Applications

Topic	Page
Introduction	3-2
TSSA Audio Modules	3-3
Audio Device Library	3-16
Board Support Package	3-27

Note

Whenever you see this icon in the text, you can click it to view an animation that helps explain the accompanying text. These animations require QuickTime to play.



Introduction

This chapter describes how to write an audio application using the range of programming interfaces available on TriMedia. For a detailed description of these APIs, refer to Reference Manuals I and II of the Philips TriMedia SDE, especially the sections on the audio renderer, audio digitizer, and the audio device library.

This chapter begins by describing a high level interface to the audio system. The audio renderer and the audio digitizer modules provide a high level interface to TriMedia audio services. These are fully compatible with other useful libraries, such as the Dolby AC3 and ProLogic decoders, and the audio mixers.

Next the reader is introduced to the audio device libraries that underlie the renderer and digitizer. Finally, the foundation provided by the board support library is briefly discussed.

TSSA Audio Modules

TriMedia software modules are constructed to a specification known as the TriMedia Streaming Software Architecture (TSSA). This software architecture is documented in Reference Manual I, Part 4. While the present chapter is easily intelligible without a background in TSSA, users will find it helpful to read about TSSA before starting serious programming.

The audio system on TriMedia is built in layers. Since the highest layer has the most functionality, this discussion will start at the top and work its way down.

A number of audio modules are available for use on TriMedia. These include the audio renderer and audio digitizer, which are used for audio playback and capture, respectively. A Dolby AC3 decoder and a Dolby ProLogic decoder are available. An example of a simple audio mixer is provided with source code. And the DTV demonstration application includes an audio system that connects all of these together. In addition, MPEG audio decoders and G.723 audio codecs are available as portions of the DVD player and the Video Phone packages, respectively. The DTV demonstration is constructed using TSSA-compatible libraries. The DVD and Video Phone libraries are not yet TSSA-compliant.

The Audio Renderer

This chapter is an overview of the audio renderer. A detailed reference to the API of the audio renderer is provided in Chapter 6, “TriMedia Audio Renderer API,” of Reference Manual II Part 2.

The audio renderer is designed to make it easy to play audio on TriMedia. The audio renderer installs an interrupt service routine and uses it to play buffers of audio. The audio renderer is a high level interface that is uniform across different hardware implementations.

The audio renderer can, in fact, be run in two different modes. These are sometimes known as *push* mode and *pull* mode. In the push mode, no operating system dependencies exist, and a simple function is used to copy audio to the output. This is the push, from application to renderer. While this model is easy to understand, it does not lend itself to expansion. In particular, many details of operation are left to the application. A higher level interface standardizes many of the details of data exchange in order to eliminate the duplication of code. A demonstration of the push model is available in the `exalArendAO` demonstration program.

When the pull model is used to render audio, the producer of audio places buffers full of data into a queue. Empty packets are available in another queue. Since the application is driven by the need for empty packets, we say that it “pulls” packets from the empty queue. Several demonstration programs illustrate the use of the audio renderer in this mode. We will first discuss the one known as `exolArendAO`.

It is easiest to demonstrate the audio renderer by connecting it to a file reader. This lets you play audio files. An illustration of the code used for this task is shown in its entirety below. Directly following, each part of code is examined and discussed.

```

void ARendFilePlay(char *fileName){
    tmLibappErr_t          err;
    Int                    readerInstance, arendInstance;
    Char                   ins[80];
    ptmolArendAOInstanceSetup_t arSetup;
    ptmolFreadInstanceSetup_t  frSetup;
    ptmAudioFormat_t        paf;
    tmAudioFormat_t         audioFormat;
    ptmolFreadCapabilities_t  frCaps;
    ptmolArendAOCapabilities_t arCaps;
    tsaInOutDescriptorSetup_t iodSetup;
    ptsaInOutDescriptor_t    iod;

    /* find out what formats are supported */
    tmolFreadGetCapabilities(&frCaps);
    tmolArendAOGetCapabilities(&arCaps);
    if (!(paf->dataSubtype & apfStereo16)) {
        printf("Stereo audio playback not supported on this board.\n");
        return;
    }

    /* Open the components involved and get their setup structures */
    err = tmolFreadOpen(&readerInstance);
    tmAssert((err == TMLIBAPP_OK), err);
    err = tmolFreadGetInstanceSetup(readerInstance, &frSetup);
    tmAssert((err == TMLIBAPP_OK), err);

    err = tmolArendAOOpen(&arendInstance);
    tmAssert((err == TMLIBAPP_OK), err);
    err = tmolArendAOGetInstanceSetup(arendInstance, &arSetup);
    tmAssert((err == TMLIBAPP_OK), err);

    /* describe the connection between the two components */
    /* assemble audio format */
    audioFormat.size          = sizeof(tmAudioFormat_t);
    audioFormat.hash          = audioFormat.referenceCount = 0;
    audioFormat.dataClass     = avdcAudio;
    audioFormat.dataType      = atfLinearPCM;
    audioFormat.dataSubtype   = apfStereo16;
    audioFormat.description   = 16;
    audioFormat.sampleRate    = sRate;
    /* create an InOutDescriptor */
    iodSetup.format           = (ptmAvFormat_t)&audioFormat;
    iodSetup.flags            = tsaIODescSetupFlagCacheMalloc;
    iodSetup.fullQName        = "full";
    iodSetup.emptyQName       = "mpty";
    iodSetup.queueFlags       = tmosQueueFlagsStandard;
    iodSetup.senderCap        = frCaps->defaultCapabilities;
    iodSetup.receiverCap      = arCaps->defaultCapabilities;
    iodSetup.senderIndex      = 0;

```

```

iodSetup.receiverIndex = 0;
iodSetup.packetBase = 0;
iodSetup.numberOfPackets = NUMBER_OF_PACKETS;
iodSetup.numberofBuffers = 1;
iodSetup.bufSize[0] = 2 * sizeof(Int16) * BUFSIZE;
err = tsaDefaultInOutDescriptorCreate(&iod, &iodSetup);
tmAssert((err == TMLIBAPP_OK), err);

/* setup file reader */
frSetup->defaultSetup->outputDescriptors[0] = iod;
frSetup->defaultSetup->priority = READER_PRIORITY;
frSetup->fileName = fileName;
printf("Opening %s for playback\n", frSetup->fileName);
err = tmolFreadInstanceSetup(readerInstance, frSetup);
tmAssert((err == TMLIBAPP_OK), err);

/* setup audio renderer */
arSetup->defaultSetup->inputDescriptors[0] = iod;
arSetup->defaultSetup->errorFunc = arend_error_func;
arSetup->maxBufferSize = 2 * sizeof(Int16) * BUFSIZE;
err = tmolArendAOInstanceSetup(arendInstance, arSetup);
tmAssert((err == TMLIBAPP_OK), err);

/* now everything is ready: Start the renderer */
err = tmolArendAOStart(arendInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolFreadStart(readerInstance);
tmAssert((err == TMLIBAPP_OK), err);

printf("file %s playing as stereo audio. \n", frSetup->fileName);
printf("Press return to stop\n");
gets(ins);

/* Stop the File everything. */
err = tmolFreadStop(readerInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolArendAOStop(arendInstance);
tmAssert((err == TMLIBAPP_OK), err);
printf("All stopped.\n");
err = tsaDefaultInOutDescriptorDestroy(iod);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolFreadClose(readerInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolArendAOClose(arendInstance);
tmAssert((err == TMLIBAPP_OK), err);

return;
}

```

Check Capabilities:

```

/* find out what formats are supported */
tmolFreadGetCapabilities(&frCaps);
tmolArendAOGetCapabilities(&arCaps);
if (!(paf->dataSubtype & apfStereo16)) {
    printf("Stereo audio playback not supported on this board.\n");
    return;
}

```

The capabilities function allows you to find out what formats are supported by the system. The capabilities of each component are required for setup. Hence these calls are made at the start of the program. Ultimately, the board support package is responsible for setting the capabilities of the audio system. The audio formats returned by the renderer are retrieved through the board support package (see “Board Support Package” starting on page 3-27).

Open the Components:

```

err = tmolFreadOpen(&readerInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolFreadGetInstanceSetup(readerInstance, &frSetup);
tmAssert((err == TMLIBAPP_OK), err);

err = tmolArendAOpen(&arendInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolArendAOGetInstanceSetup(arendInstance, &arSetup);
tmAssert((err == TMLIBAPP_OK), err);

```

Each component that will be used must be opened. This creates an instance of the component for you to use. The GetCapabilities function is called to retrieve a setup structure that has been initialized to default values. Notice the use of tmAssert. Like the ANSI assert(), tmAssert() will halt the program and print the file name and line number on an error condition. In addition, tmAssert() prints the error code, and it prints it all both to STDOUT and to the DP buffer. This assert mechanism is used liberally throughout TM audio code. It is invaluable in the identification of programming errors. And when the program is running, it is easy to turn off the tmAssert checking. Compilation with the flag “-DNO_DEBUG” removes all of the assertion checking. In this way, the assert checking provides strong error checking when appropriate, and it has no run time impact when the code is released.

Make the Connection Between Each Pair of Components:

```

/* assemble audio format */
audioFormat.size           = sizeof(tmAudioFormat_t);
audioFormat.hash           = audioFormat.referenceCount = 0;
audioFormat.dataClass      = avdcAudio;
audioFormat.dataType        = atfLinearPCM;
audioFormat.dataSubtype     = apfStereo16;
audioFormat.description     = 16;
audioFormat.sampleRate     = sRate;
/* create an InOutDescriptor */
iodSetup.format             = (ptmAvFormat_t)&audioFormat;
iodSetup.flags               = tsaIODescSetupFlagCacheMalloc;
iodSetup.fullQName           = "full";
iodSetup.emptyQName         = "mpty";
iodSetup.queueFlags         = tmosQueueFlagsStandard;
iodSetup.senderCap          = frCaps->defaultCapabilities;
iodSetup.receiverCap        = arCaps->defaultCapabilities;
iodSetup.senderIndex        = 0;
iodSetup.receiverIndex      = 0;
iodSetup.packetBase         = 0;
iodSetup.numberOfPackets    = NUMBER_OF_PACKETS;
iodSetup.numberOfBuffers    = 1;
iodSetup.bufSize[0]         = 2 * sizeof(Int16) * BUFSIZE;
err = tsaDefaultInOutDescriptorCreate(&iod, &iodSetup);
tmAssert((err == TMLIBAPP_OK), err);

```

Each pair of TSSA components are connected by a structure called an InOutDescriptor. The function `tsaDefaultInOutDescriptorCreate()` is used to create one of these connections. The parameters that must be specified are illustrated here. In this example, a valid format structure is passed in when the connection is created. It is also possible to pass in Null. The format can be specified later using the `tsaDefaultInstallFormat()` command, or it can even be determined after the receiving component has started. This might be more convenient when using a decoder that finds the format in the data stream only after it has decoded some data. In this case, the format is passed in the data packet that travels through the queue inside of the InOutDescriptor.

The `CreateInOutDescriptor` function can also create the data packets that are used to stream data between the file reader and the audio renderer. These are initially placed in the empty queue. Setting the `numberOfPackets` field to zero will bypass this step, if you have some special reason to create your own packets. This code illustrates a fairly typical approach to the problem.

Setup the File Reader

```
frSetup->defaultSetup->outputDescriptors[0] = iod;
frSetup->defaultSetup->priority = READER_PRIORITY;
frSetup->fileName = fileName;
printf("Opening %s for playback\n", frSetup->fileName);
err = tmoLFreadInstanceSetup(readerInstance, frSetup);
tmAssert((err == TMLIBAPP_OK), err);
```

The file reader is a TSSA component that provides a streaming interface to a file. It takes packets from its empty queue, reads from the disk to fill them, and then places the packets in its full queue. As a default, the file reader loops back to the beginning when it reaches the end of the file. More information about the file reader can be found in Chapter 3, “TriMedia File Reader API,” in Reference Manual II Part 2.

Given the already initialized file reader setup structure that was retrieved after open, the file reader is very simple to setup. A file name is clearly required. The InOutDescriptor is required. And a priority is assigned for the pSOS task that will be created.

The amount of data read in each packet is determined by the bufSize field in the header of each packet. This was initialized when the packets were created and the memory was allocated by tsaDefaultInOutDescriptorCreate().

Setup the Audio Renderer

```
arSetup->defaultSetup->inputDescriptors[0] = iod;
arSetup->defaultSetup->errorFunc = arend_error_func;
arSetup->maxBufferSize = 2 * sizeof(Int16) * BUFSIZE;
err = tmoLArendAOInstanceSetup(arendInstance, arSetup);
tmAssert((err == TMLIBAPP_OK), err);
```

The audio renderer is implemented as an interrupt service routine. It is not a task. Like the reader, a partially initialized setup structure was obtained after the component was opened. The user must specify an InOutDescriptor, and a maximum buffer size. The error reporting function is optional. The format of the audio data stream is specified as part of the InOutDescriptor. After the call to tmoLArendInstanceSetup(), we are ready for start.

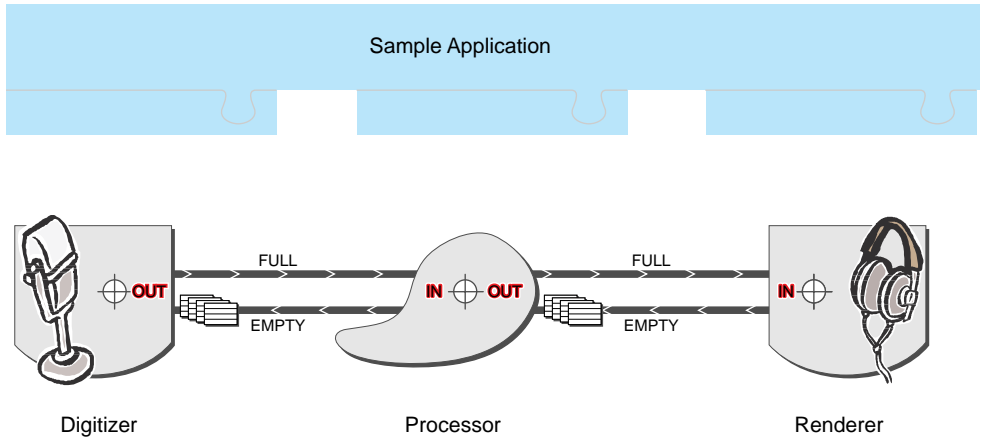
Start

```
err = tmoLFreadStart(readerInstance);
err = tmoLArendAOSTart(arendInstance);

printf("file %s playing as stereo audio.\n", frSetup->fileName);
printf("Press return to stop\n");
gets(ins);
```

The calls to the start functions (tmoLArendAOSTart, and tmoLFreadStart) cause these two independent components to begin exchanging data. The audio renderer runs in an

interrupt service routine. Under current pSOS rules, this uses the stack of the currently running task. The file reader is started as an autonomous task. Since buffers start in the empty queue, the file reader will immediately begin to fill these buffers, and packets will bunch up in the full queue. The audio renderer will be activated after each buffer has played. If these buffers contain 256 samples of stereo audio (1024 bytes), and the sample rate is 44100, the audio renderer will request a new packet every 5.8ms. The renderer requests a new packet from the full queue. In steady state operation, it also places the previous packet in the empty queue. Since the reader task is blocked waiting for an empty buffer, the reader is now ready to run and the cycle can continue.



The `printf` and `gets` provide a simple and convenient development interface. Since this code is in a thread separate from the reader and the renderer, the fact that this thread is blocked has no effect on the other threads.

Stop and Shutdown

```
err = tmolFreadStop(readerInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolArendAOSTop(arendInstance);
tmAssert((err == TMLIBAPP_OK), err);
printf("All stopped.\n");
err = tsaDefaultInOutDescriptorDestroy(iod);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolFreadClose(readerInstance);
tmAssert((err == TMLIBAPP_OK), err);
err = tmolArendAOClose(arendInstance);
tmAssert((err == TMLIBAPP_OK), err);
```

When it is time to stop the process, the stop functions are called. The operation of each stop function is synchronous; that is, the stop function will not return until the component being stopped has actually completed its work. Under TSSA, stop means “return all your

memory and exit your processing loop.” Hence at the end of the stop procedure, all of the packets should be returned to the queues.

Advanced Features

The audio renderer is a reasonably mature interface. It supports the basic features well, and it also provides some advanced features. One of these is the progress callback function. The progress callback function can be called at every interrupt service routine. This can be used to implement synchronization functions like that required to lock the output to a digital audio input.

Another advanced feature of the audio renderer is its handling of time-stamped packets. If the renderer is set up with a clock reference, and if its packets are time-stamped, the renderer will attempt to present these packets at the correct time. If the packet arrives too early, the renderer will hold onto it until its presentation time arrives. If it is too late, the packet will be returned immediately so as to catch up. This mechanism can be used to implement AV (“lip”) sync. It assumes that once sync is achieved, the audio and video will remain in sync. If that is not the case, then the DDS should be used to vary the audio clock so as to achieve long term sync.

Audio Digitizer

The audio digitizer is an interface to audio input. Like all TSSA components, a section of the API reference manual is devoted to it. Some example programs such as ex01AIO are provided as well. The following code illustrates the basic operation of the audio digitizer:

```

/* Get Capabilities */
rval = tmlAdigAIGetCapabilities(&AdigAICap);
rval = tmlArendAOGetCapabilities(&ArendAOCap);

/* Open components */
rval = tmlAdigAIOpen(&digitizerInstance);
rval = tmlArendAOOpen(&arendInstance);

/* Get setup variables */
rval = tmlAdigAIGetInstanceSetup(digitizerInstance, &digitizerSetup);
rval = tmlArendAOGetInstanceSetup(arendInstance, &arendSetup);

/* create the I/O descriptor to connect components */
descriptorSetup.format          = (ptmAvFormat_t)&audioFormat;
descriptorSetup.flags          = tsaIODescSetupFlagCacheMalloc;
descriptorSetup.fullQName      = "AIOQ";
descriptorSetup.emptyQName    = "AOIQ";
descriptorSetup.queueFlags     = 0;
descriptorSetup.senderCap      = AdigAICap->defaultCapabilities;
descriptorSetup.receiverCap    = ArendAOCap->defaultCapabilities;
descriptorSetup.senderIndex    = 0;
descriptorSetup.receiverIndex  = 0;
descriptorSetup.packetBase     = 0x100;
descriptorSetup.numberOfPackets = MAX_PACKETS;
descriptorSetup.numberOfBuffers = 1;
descriptorSetup.bufSize[0]     = bytesPerPacket;
rval = tsaDefaultInOutDescriptorCreate(&iod, &descriptorSetup)

/* setup components */
digitizerSetup->defaultSetup->errorFunc = digitizer_error_func;
digitizerSetup->defaultSetup->outputDescriptors[0] = iod;
rval = tmlAdigAIInstanceSetup(digitizerInstance, digitizerSetup);

arendSetup->defaultSetup->inputDescriptors[0] = iod;
arendSetup->defaultSetup->errorFunc = renderer_error_func;
arendSetup->maxBufferSize = bytesPerPacket;
rval = tmlArendAOInstanceSetup(arendInstance, arendSetup);

/* now everything is ready: Start */
rval = tmlArendAOSTart(arendInstance);
rval = tmlAdigAISTart(digitizerInstance);

printf("Press return to stop\n");
gets(ins);

* Stop everything. */
rval = tmlAdigAISTop(digitizerInstance);
rval = tmlArendAOSTop(arendInstance);

```

```
rval = tmoAdigAIClose(digitizerInstance);
rval = tmoArendAOClose(arendInstance);
rval = tsaDefaultInOutDescriptorDestroy(iod);
```

You can see the similarity to the setup of other TSSA components. The sequence of `Open`, `GetInstanceSetup`, `InOutDescriptorCreate`, `InstanceSetup`, `Start` is very common. Like the audio renderer, the audio digitizer runs in an interrupt service routine.

One interesting feature of the audio digitizer is its second output. The digitizer has two outputs. This allows the output to be simultaneously routed to a file writer and to the audio renderer as a monitor.

CopyAudio Example

The `copyAudio` example program connects the audio digitizer to a simple data copier and through to the audio renderer. This can easily serve as a starting point for the development of new TriMedia audio modules.

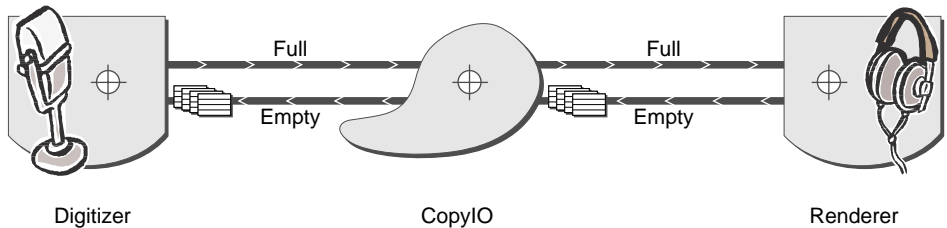


Figure 3-1 Example of the flow of a simple audio copy

As you might guess from looking at the picture, this will consist of code to create the modules, create the queues, and connect the modules.

Create the Components:

```
tmoAdigAIOpen(&digitizerInstance);
tmoCopyIOOpen(&copyInstance);
tmoArendAIOpen(&arendInstance);
```

Of course the return values must be checked.

Create and Populate the Queues

```

/* Get Capabilities */
rval = tmlAdigAIGetCapabilities(&digitizerCap);
rval = tmlCopyIOGetCapabilities(&copyCap);
rval = tmlArendAOGetCapabilities(&arendCap);

/* Setup iosetups */
iosetup1.format          = (ptmAvFormat_t)&aFormat;
iosetup1.flags          = tsaIODescSetupFlagCacheMalloc;
iosetup1.fullQName      = "digF";
iosetup1.emptyQName     = "digE";
iosetup1.queueFlags     = 0;
iosetup1.senderCap      = digitizerCap->defaultCapabilities;
iosetup1.receiverCap    = copyCap->defaultCapabilities;
iosetup1.senderIndex    = 0;
iosetup1.receiverIndex  = 0;
iosetup1.packetBase     = 0x100;
iosetup1.numberOfPackets = MAX_PACKETS;
iosetup1.numberOfBuffers = 1;
iosetup1.bufSize[0]     = BUFSIZE * 2 * sizeof(Int16);

iosetup2.format          = (ptmAvFormat_t)&aFormat;
iosetup2.flags          = tsaIODescSetupFlagCacheMalloc;
iosetup2.fullQName      = "renF";
iosetup2.emptyQName     = "renE";
iosetup2.queueFlags     = 0;
iosetup2.senderCap      = copyCap->defaultCapabilities;
iosetup2.receiverCap    = arendCap->defaultCapabilities;
iosetup2.senderIndex    = 0;
iosetup2.receiverIndex  = 0;
iosetup2.packetBase     = 0x200;
iosetup2.numberOfPackets = MAX_PACKETS;
iosetup2.numberOfBuffers = 1;
iosetup2.bufSize[0]     = BUFSIZE * 2 * sizeof(Int16);

/* Create InOutDescriptors */
rval = tsaDefaultInOutDescriptorCreate(&iodesc1, &iosetup1);
rval = tsaDefaultInOutDescriptorCreate(&iodesc2, &iosetup2);

```

The error checking is removed from this example, but the important points are illustrated. `aFormat` is a statically initialized `tmAudioFormat_t` structure. It determines the buffer size to be used as well as the sample rate. The `CreateInOutDescriptor` function is used to create the queues and the packets. After these calls, `MAX_PACKETS` valid packets have been placed in the empty queues. When the components are started, these packets will begin to flow. The number of packets is up to the application. A queue full of packets performs a buffering function, but it also increases the delay, or latency through the system.

Set Up the Components

```

/* setup audio digitizer */
rval = tmlAdigAIGetInstanceSetup(digitizerInstance, &digitizerSetup);
digitizerSetup->defaultSetup->outputDescriptors[0] = iodesc1;
rval = tmlAdigAIInstanceSetup(digitizerInstance, digitizerSetup);

/* setup copy component */
rval = tmlCopyIOGetInstanceSetup(copyInstance, &copySetup);
copySetup->defaultSetup->inputDescriptors[0] = iodesc1;
copySetup->defaultSetup->outputDescriptors[0] = iodesc2;
rval = tmlCopyIOInstanceSetup(copyInstance, copySetup);

/* initialize audio renderer */
rval = tmlArendAOGetInstanceSetup(arendInstance, &arendSetup);
arendSetup->defaultSetup->inputDescriptors[0] = iodesc2;
arendSetup->defaultSetup->errorFunc = renderer_error_func;
arendSetup->maxBufferSize = bytesPerPacket;
rval = tmlArendAOInstanceSetup(arendInstance, arendSetup);

```

That is all that there is to the component setup. All that remains is to start:

```

tmlAdigAIStart(digitizerInstance);
tmlArendAOStart(arendInstance);
tmlCopyIOStart(copyInstance);

```

Modifying the Copy Component:

The CopyIO component makes a very good starting point for the creation of your own audio processing component. Before you jump in and start to hack up the CopyIO component, we recommend that you read Chapter 9 “TSSA Component Internals” of Reference Manual I Part 4. The TSA architecture chapter will give you the background to understand a component’s framework. The core of the copyIO component is found in the function `tmlCopyIOCopyPacket`. There you’ll find this code:

```

for(i=0; i<outpacket->allocatedBuffers; i++) {
    memcpy(outpacket->buffers[i].data, inpacket->buffers[i].data,
        inpacket->buffers[i].dataSize);
    outpacket->buffers[i].dataSize = inpacket->buffers[i].dataSize;
}

```

You can replace the `memcpy` with your audio processing routine. Other things that you may need to change include the allocation and freeing of memory. That should be done in the `Open` and `Close` functions. Initial parameter setting is done in the `InstanceSetup` function. When you come to this point, you’ll see how the AL layer interfaces with the OL layer in the `InstanceSetup` function.

This approach will get your component up and running quickly. But as your component becomes more mature, you will want to adopt more of the TSA conventions that are illustrated in the audio mixer code below.

Audio Mixer

The example program known as `ex01AmixSimple` demonstrates the use of a simple audio mixer. The simple mixer is supplied complete with source. Out of the box, it accepts three stereo inputs and mixes them into one stereo output. This simple audio mixer demonstrates the concepts involved in the construction of a mixer. It is a simplified version of the mixer used with the TriMedia Digital Television (DTV) system. The source for the library illustrates several important concepts:

- The mixer supports a configuration function with a queued interface.
- The mixer demonstrates how to handle multiple input pins. The principles are similar for multiple output pins.
- The mixer separates the `tmal` and `tmol` layers using a subdirectory. This is done to make it easier to isolate the valuable intellectual property that exists in your code at the AL layer. It is common practice to guard the AL layer source. By making the OL layer source available, it is possible for a client to change the operating system without accessing your private source.

Audio Decoders

A number of audio decoders are available for use with TriMedia. These include decoders for Dolby AC3 and Dolby ProLogic. Each of these are delivered as TSSA-compatible modules. Since a separate licensing fee is required for these decoders, you are advised to contact your TriMedia sales representative for more information. Code is also available to decode MPEG 1 layer 2 audio, and G.723, although it is not packaged as a TSSA module.

Audio Device Library

If for some reason the TSSA audio interface is not appropriate, a lower level of access is available. It is this “device library” interface that is used to construct the audio renderer and the audio digitizer. The TSSA interface solves many problems that have deliberately been left unaddressed at the device library level. But of course, there are other ways to solve the same problems.

Audio Hardware Overview

The TriMedia Audio-In unit connects to an off-chip stereo analog-to-digital (A/D) converter subsystem through a flexible bit-serial bus. It provides all signals needed to interface to high-quality, low-cost oversampling (analog-to-digital) A/D converters, including a precisely programmable oversampling A/D system clock.

The TriMedia Audio-Out unit connects to an off-chip stereo digital-to-analog (D/A) converter subsystem through a flexible bit-serial interface. It provides an interface to high-quality, low-cost oversampling D/A converters and a precisely programmable oversampling D/A system clock.

The Audio-In /Audio-Out unit implements a double-buffering scheme, ensuring that no samples are lost even if the DSPCPU is highly loaded and slow to respond to interrupts.

The Audio-In /Audio-Out unit is reset by writing a 0x80000000 to the AI_CONTROL/AO_CONTROL) register. This disables capture/transmission by setting the CAP_ENABLE / TRANS_ENABLE) flag to 0, and makes `buffer1` the active buffer by setting `BUF1_ACTIVE` flag to 1.

Capture/Transmission by DSPCPU

1. The DSPCPU initiates capture/transmission by providing two empty/full buffers and putting their base addresses and sizes in the `BASEn` and `COUNTn/SIZEn` registers. It does so by writing a base address and size to MMIO control fields.
2. After two valid local memory buffers are assigned, capture/transmission is enabled by setting `CAP_ENABLE/TRANS_ENABLE` to 1.
3. The Audio-In /Audio-Out unit hardware then fills/empties `buffer1` by reading input/transmitting output samples. After `buffer1` fills/empties, `BUF1_FULL/ BUF1_EMPTY` is asserted and capture/transmission continues without interruption in `buffer2`.
4. Before `buffer2` fills up, the DSPCPU must assign a new, empty/full buffer to `BASE1`, `COUNT1/SIZE1`, and perform an `ACK1`. `BUF2_FULL/BUF2_EMPTY` is asserted when `buffer2` fills up/empties, and capture/transmission continues in/from the new `buffer1`, and so forth.

5. Upon receipt of an ACK, the Audio-In /Audio-Out hardware removes the interrupt line assertion at the next DSPCPU clock edge. Refer to the interrupt controller documentation for details about interrupt handler programming.

In normal operation, the DSPCPU and the Audio-In /Audio-Out hardware continuously exchange buffers without losing a sample.

However, timing is important in the Audio-In unit. If, for example, the DSPCPU fails to provide a new buffer in time, the `OVERRUN` error flag is raised, causing a temporary halt to input sampling. Sampling resumes as soon as the DSPCPU makes one or more new buffers available through an `ACK1` or `ACK2` operation.

Timing is important in the Audio-Out unit, as well. If, for example, the DSPCPU fails to provide a new buffer in time, the `UNDERRUN` error flag is raised, and the last valid sample or sample pair is repeated until a new buffer of data is assigned by `ACK1` or `ACK2`.

The TriMedia Audio-In/Out APIs provide the necessary interface for audio applications to access the TriMedia Audio-In/Out unit hardware.

Using the TriMedia Audio-In/Audio-Out API

The functions provided in the TriMedia Audio-In/Audio-Out API enable you to access both the Audio-In and Audio-Out hardware units of TriMedia. The Audio-In/Audio-Out device library provides functions to control audio coders-encoders (codecs) attached to the TM-1000, as well as support for the audio mixer and other audio subsystems.

The interface provided by the Audio-In/Audio-Out device library is simple to use. To access the Audio-In or Audio-Out unit, the application program first opens the unit and sets a few parameters, and then initiates capturing or transmission by removing the pause condition. The audio is then serviced by interrupts. After the audio is running, its volume, sample rate, and input selection are controlled by the APIs provided in the Audio-In/Audio-Out device library.

Guidelines for Use of the Audio-In/Audio-Out APIs

General guidelines for using the TriMedia Audio-In/Audio-Out APIs are as follows:

- Include the `<tm1/tmAI.h>` and `<tm1/tmAO.h>` header files.
- Use the archive version (`libdev.a`), rather than building the library yourself. (The Audio-In/Audio-Out device library is archived in `libdev.a`)
The source for the Audio-In/Audio-Out device library is included in the TriMedia Compilation System (TCS). This makes it easier to incorporate new versions of the library as they become available.
- Pass the specific owner ID when making subsequent calls.

The Audio-In/Audio-Out device library operates as an exclusive device driver, and, as such, can service only one task at a time. This is enforced through the owner field of the control data structure, which is returned by all the initialization functions.

- Check the error values returned by the initialization functions. Most of the Audio-In/Audio-Out device library functions return zero on success, or nonzero error codes. Many functions check and report the use of sizes and alignments that the hardware cannot support.

Restrictions

Because of hardware or software limitations, the Audio-In/Audio-Out device library has the following restrictions:

- The buffers must be 64-byte aligned, and buffer sizes must be a multiple of 64 samples.
- Calculation of the sample rate is based on the TriMedia cycle clock. The software gets its definition of this clock from the `tmman.ini` file residing in the current directory. You must ensure the value of `tmman.ini` matches your hardware.
- When setting sample rates, consider that the value for the DDS control register is computed in 32-bit math. This might lead to inaccuracies because of truncation. The problem will be fixed in future releases.

Demonstration Programs

Included with the Audio-In/Audio-Out device library are six demonstration programs:

- `fplay`
- `fplay6`
- `sine`
- `sthru`
- `avio`
- `patest`

If you want to develop audio applications for TriMedia, you can use these demonstration programs to gain an understanding of how to use Audio-In/Audio-Out device library APIs within your applications.

IMPORTANT

You will achieve a greater level of compatibility with other TriMedia software modules through the use of the TSSA audio interface.

Playing an Audio File

The following example demonstrates the role Audio-Out APIs play in an audio file by using the Audio-Out unit. The code is taken from the `fplay` demonstration program that is provided with the Audio-In/Audio-Out device library.

```
static void
fPlay(char *waveFile, float srate)
{
    aoInstanceSetup_t ao;
    FILE          *fp;
    Int           instance, i;
    char          ins[80];

    samples = (int *) malloc(MAX_SAMPLE_SIZE * 4);
    if (!samples) {
        printf("FATAL ERROR: Error getting sample memory\n");
        exit(1);
    }

    printf("loading sound file %s...\n", waveFile);
    fp = fopen(waveFile, "rb");
    if (!fp) {
        printf("FATAL ERROR: Failed to open sound file.\n");
        exit(2);
    }
    sample_bytes = fread(samples, 1, MAX_SAMPLE_SIZE, fp);
    printf("sample size is %d bytes.\n", sample_bytes);
    fclose(fp);

    pbuf1 = (int *) (((unsigned long) buf1 + 63) & ~63U);
    pbuf2 = (int *) (((unsigned long) buf2 + 63) & ~63U);

    memset(pbuf1, 0, BUF_SIZE * 4);
    memset(pbuf2, 0, BUF_SIZE * 4);

    for (i = 0; i < BUF_SIZE; i += 16) {
        _cache_copyback(pbuf1, BUF_SIZE);
        _cache_copyback(pbuf2, BUF_SIZE);
    }

    ao.isr = fPlayISR;
    ao.interruptPriority = intPRIO_3;
    ao.audioTypeFormat = atfLinearPCM;
    ao.audioSubtypeFormat = apfStereo16;
    ao.srate = srate;
    ao.size = BUF_SIZE;
    ao.base1 = pbuf1;
    ao.base2 = pbuf2;
    ao.underrunEnable = True;
    ao.hbeEnable = True;
    ao.buf1emptyEnable = True;
    ao.buf2emptyEnable = True;
}
```

```

LIBDEV(aoOpen(&instance));
LIBDEV(aoInstanceSetup(instance, &ao));

aoEnableLITTLE_ENDIAN();

LIBDEV(aoStart(instance));

printf("wave file playing: Press return to stop.\n");
gets(ins);

LIBDEV(aoStop(instance));
LIBDEV(aoClose(instance));

exit(0);
}

```

Before initializing the audio output hardware, the `fplay` demonstration program must do the following:

- Align the buffers on a 64-byte boundary.
- Call the `_cache_copyback()` function to ensure cache coherency. This is done because the Audio hardware reads from SDRAM and not from Cache. The `_cache_copyback()` function uses an optimized algorithm to flush the cache.
- Set the audio out parameters. The Interrupt Service Routine (ISR) pointer is set to `fPlayISR` (see the following example).
- The `fplay` demonstration program then initializes the Audio-Out hardware by calling the `aoOpen` function, which assigns the audio control block to the owner for exclusive use. `LIBDEV` checks the return value of the `aoOpen` function before proceeding.

The format and the interrupt parameters are initialized from the values in `AO`. The endianness is set to little endian to conform to the file format.

The procedure halts until a console line is read. Wave file playback is interrupt driven. After the input, the `AO` unit is stopped and closed.

Interrupt Routine `fplayISR`

The following is a description of the interrupt routine `fplayISR`, which is followed by code excerpts that illustrate sequential operations.

The `pragma` tells the compiler to save and restore the interrupt state. The routine first checks for data underrun and highway bandwidth error conditions and acknowledges them.

There are two Audio Out buffers, with empty status bits for each. If the second is empty, it is filled with the data in “sample” (a circular buffer). The data read is copied back to memory and the interrupt is acknowledged. If buffer 1 is empty, it is handled in the same manner as buffer 2.

The pragma at the end of the function forces a decision tree jump. This is to allow sufficient time between the acknowledgment and the return from interrupt.

The code for the interrupt routine `fplayISR` is shown below.

```
static void
fPlayISR(void)
{
#pragma TCS_handler

    int        i;
    UInt       stat = MMIO(AO_STATUS);
```

1. Check for underrun and highway bandwidth errors.

```
    if (aoUNDERRUN(stat))
        aoAckACK_UDR();
    if (aoHBE(stat))
        aoAckACK_HBE();
```

2. Next, it copies data to AO buffer 2, if it is empty, and it resets the pointer if it is at the end of the buffer.

```
    if (aoBUF2_EMPTY(stat)) {
        for (i = 0; i < BUF_SIZE; i++) {
            pbuf2[i] = samples[sample_pos];
            if (sample_pos++ >= (sample_bytes >> 2))
                sample_pos = 0;
        }
    }
```

3. Next, it forces the cache to write data to memory and it acknowledges the interrupt

```
    for (i = 0; i < BUF_SIZE; i += 16)
        _cache_copyback(pbuf2, BUF_SIZE);
    aoAckACK2();
}
```

4. `fplayISR` uses the same code for AO buffer 1 that it used with buffer 2:

```
    if (aoBUF1_EMPTY(stat)) {
        for (i = 0; i < BUF_SIZE; i++) {
            pbuf1[i] = samples[sample_pos];
```

5. Next, it resets the pointer if it is at the end of the circular buffer.

```
        if (sample_pos++ >= (sample_bytes >> 2))
            sample_pos = 0;
        }
        for (i = 0; i < BUF_SIZE; i += 16)
            _cache_copyback(pbuf1, BUF_SIZE);
        aoAckACK1();
    }

#pragma TCS_break_dtree
}
```

Recording an Audio File

The following example demonstrates the use of Audio-In APIs to create an audio file by reading audio data from the Audio-In unit. The code is taken from the sthru demonstration program, which is provided with the Audio-In/Audio-Out device library.

sthru Demonstration Program

The first part of the sthru demonstration program allocates and clears the capture buffer and the data buffers. (This code is not shown).

On receiving the interrupt, the DSPCPU executes the interrupt service routine inISR (see the following example). The interrupt routine first reads the newly captured data from the inactive buffer pointer using aiGetBase and then writes a new pointer to the buffer that is ready for capture data using aiChangeBuffer. Because the captured data is not cache-coherent, stale data is removed from the buffer using invalidate. Finally, the ISR acknowledges the interrupt by clearing the bit in the status register.

```
void sCapture(float srate)
{
    AUDIO_CB      in_a;
    int           i, j;
    FILE          *fp;
    int           retval;
    int           *p1, *p2;

    ptr = rawPtr;
    capCount = 0;

    /* setup control structure */
    in_a.format = AIO_FORMAT_STEREO_16;
    in_a.sRate_hz = srate;
    in_a.size_samples = BUFSIZE;
    in_a.flags = 0;
    in_a.isr = capISR;

    retval = aiOpen(&in_a, &in_owner);
    if (0 != retval)
    {
        printf("aiOpen failed with %d. Aborting...\n", retval);
        return;
    }
    if (0 != aiSetBufferSize(in_owner, in_a.size_samples))
        printf("aiSetBufferSize failed (illegal size?)\n");

    p1 = (int *) (((int) ptr + 63) & 0xFFFFFC0);
    p2 = &p1[BUFSIZE];
    ptr += BUFSIZE;
    if (0 != aiSetBuffer1Base(in_owner, p1))
```



```

        printf("aiSetBuffer1Base failed (illegal alignment?
        0x%x)\n", p1);

        printf("aiSetBuffer2Base failed (illegal alignment?
        0x%x)\n", p2);
    printf("\nCapturing %d seconds of audio input..\n",
    (mallocSize>>2)/ (int)srate);

    aiUnpause(in_owner);

    while (capCount < ((mallocSize>>2)/BUFSIZE -2))
        if ( (capCount % 192) == 0)
            printf("..\n");
    printf("writing data to 'capture.bin'...\n");
    aiPause(in_owner);
    aiClose(in_owner);

    fp = fopen("capture.bin", "wb");
    if (!fp)
    {
        printf("Failed to open capture file.\n");
        return;
    }
    printf("Wrote %d words into capture.bin. \n", fwrite(rawPtr,
    sizeof(int), mallocSize>>2, fp));
    fclose(fp);

    printf("capture Test completed\n");
}

```

Setting Audio Parameters

After the audio is running (capture or transmission), you can change the volume (left and right gain), sample rate, and input source by using the APIs provided in the Audio-In/Audio-Out device library.

The following examples demonstrate the use of these APIs. All of the code is taken from a demonstration program, which is provided with the Audio-In/Audio-Out device library.

The following code uses the `aiSetSampleRate` and `aoSetSampleRate` APIs to set the Audio-In and Audio-Out sample rates.

```

    aoSetSampleRate(out_owner, srate);
    aiSetSampleRate(in_owner, srate);

```

For analog input/output devices (such as the AD1847), both the audio input and audio output are performed by the same chip. Therefore, both the input and output use the same sample rate. In such cases, you can use either function.

Audio-In and Audio-Out each have an instance setup structure. These are initialized with the interrupt parameters and formats.

1. The following code shows the call to the open routines (`aoOpen`, `aiOpen`) to acquire an instance, and the instance setup routines (`aoInstanceSetup`, `aiInstanceSetup`) are then called with the instance value and the appropriate parameters.

```
int main(int argc, char **argv)
{
    aoInstanceSetup_t ao;
    aiInstanceSetup_t ai;
    char            ins[80];
    int             i;
    int             *buf;
    FILE           *fp;
```

The initialization code, argument checking code, and buffer setup is not shown.

2. The following code sets up the audio formats.

```
/* setup control structure */
ai.audioTypeFormat = ao.audioTypeFormat = atfLinearPCM;
if (monoFlag)
    ai.audioSubtypeFormat = ao.audioSubtypeFormat = apfMono16;
else
    ai.audioSubtypeFormat = ao.audioSubtypeFormat = apfStereo16;
```

3. Set up the interrupt service routine and the priority level

```
ao.isr = outISR;
ai.isr = inISR;
ao.interruptPriority = ai.interruptPriority = intPRIO_3;
```

4. Set up the sampling rate, and the size and buffer pointers (for AO).

```
ao.srate = ai.srate = sRate;
ao.size = ai.size = BUFSIZE;
ao.base1 = b[0];
ao.base2 = b[1];
```

5. Set up the interrupt enable flags for AO.

```
ao.underrunEnable = True;
ao.hbeEnable = True;
ao.buf1emptyEnable = True;
ao.buf2emptyEnable = True;
```

6. Setup the buffer pointers and the interrupt enable flags for AI.

```
ai.base1 = b[2];
ai.base2 = b[3];
ai.overflowEnable = True;
ai.hbeEnable = True;
ai.buf1fullEnable = True;
ai.buf2fullEnable = True;
```

7. Open AO and AI and configure the device.

```
ERROR_REPORT(aoOpen(&ao_instance));
ERROR_REPORT(aoInstanceSetup(ao_instance, &ao));
ERROR_REPORT(aiOpen(&ai_instance));
ERROR_REPORT(aiInstanceSetup(ai_instance, &ai));
```

8. The input and output volumes (left and right channels) are set in hundredths of DB. A negative value corresponds to attenuation and a positive value to gain. Note that the input volume must be positive or zero and the output volume must be negative or zero. The input is set to the line input of the microphone.

```
aoSetVolume(ao_instance, outputVolume * 100, outputVolume * 100);
aiSetVolume(ai_instance, inputVolume * 100, inputVolume * 100);
```

9. Set the input to the line or to the mike.

```
if (mode == MODE_MIC) {
    ERROR_REPORT(aiSetInput(ai_instance, aaaMicInput));
} else {
    ERROR_REPORT(aiSetInput(ai_instance, aaaLineInput));
}
ERROR_REPORT(aiStart(ai_instance));
ERROR_REPORT(aoStart(ao_instance));
```

10. Pause until the user types the following:

```
<CR>
printf("Audio Pass Thru is running. Press return to exit :\n");
```

11. Stop and close AI and AO.

```
ERROR_REPORT(aoStop(ao_instance));
ERROR_REPORT(aiStop(ai_instance));
ERROR_REPORT(aoClose(ai_instance));
ERROR_REPORT(aiClose(ai_instance));
```

12. Write the data to capture.bin (binary mode).

```
if (captureFlag) {
    printf("Writing %d samples of captured data to "
           "capture.bin...\n", CAPSIZE);
    fp = fopen("capture.bin", "wb");
    if (!fp) {
        printf("FATAL ERROR: capture.bin fopen failed\n");
        exit(2);
    }
    fwrite(capBuffer, 1, CAPSIZE * 4, fp);
    fclose(fp);
    printf("Done!\n");
}
exit(0);
}
```

13. After receiving the interrupt, the CPU executes the inISR ISR. (See the following example).

The ISR first determines which buffer is inactive by using the `aoBuf1Active` macro (defined in `tmAO.h`). It then fills the inactive buffer with new audio data, flushes the cache, and finally acknowledges the interrupt by clearing the bit in the status register.

```
static void
inISR(void)
{
#pragma TCS_handler

    int      *foo;
    int      i;
    UInt     stat;
```

14. Increment the input buffer pointer modulo 4.

```
inBuf++;
inBuf &= 0x3;
```

15. Read the AI status. Check for exceptional conditions.

```
stat = MMIO(AI_STATUS);
if (aiOVERRUN(stat))
aiAckACK_OVR();
if (aiHBE(stat))
aiAckACK_HBE();
```

16. If buffer 2 is full, set `foo` to the pointer. Switch to the next available buffer.

```
if (aiBUF2_FULL(stat)) {
foo = (int *) aiGetBASE2();
aiChangeBuffer2(ai_instance, b[inBuf]);
aiAckACK2();
}
```

17. If buffer one is full, set `foo` to the pointer. Switch to the next available buffer. The two buffers should never be full simultaneously.

18. Invalidate any stale data in the cache.

```
if (aiBUF1_FULL(stat)) {
    foo = (int *) aiGetBASE1();
    aiChangeBuffer1(ai_instance, b[inBuf]);
    aiAckACK1();
}
for (i = 0; i < BUFSIZE; i += 16)
    INVALIDATE((char *) &foo[i], 1);
```

19. Copy the data into the capture buffer.

```
for (i = 0; i < BUFSIZE; i++) {
    if (capPtr >= CAPSIZE)
        break;
    capBuffer[capPtr++] = foo[i];
}
```

There are decision tree breaks previously, so this one is actually unnecessary.

Board Support Package

The board support package is an integral part of the TriMedia audio system. It is the lowest functional level of the interface. It is at this level that the actual capabilities of the system are determined.

The board support package delivered with the TriMedia developers kit includes support for a number of boards. These include the standard “IREF” board, as well as Philips reference boards for DTV. The board support package detects which board is in use and selects the appropriate function tables to drive that board. This mechanism is explained in some depth in Chapter 10 “TriMedia TMBoard API” of Reference Manual II, Part 1.

Some examples of the types of capabilities that can be supported through the board support package are:

- The IREF hardware cannot support simultaneous stereo input and six channel output. This is coded into the board support package.
- The DTV board supports 8 channels of 20-bit audio output. This is done using an external FPGA with the audio clock running at double speed. All of the setup for this configuration is in the board support package.
- The AD1847 on the IREF board supports volume control. This is accessible because it is supported in the board support package.
- The DTV board supports digital audio input. The code to control this resides in the board support package.

Philips TriMedia SDE Cookbook

Part 3:

Bootstrapping TriMedia



Table of Contents

Chapter 1	Bootstrapping TriMedia in Autonomous Mode	
	Introduction.....	1-2
	Overview of Stand-Alone Boot.....	1-2
	Creating an EEPROM image.....	1-2
	EEPROM Header	1-2
	L1 Boot Program.....	1-3
	Sample Programs	1-5
	makefile.unix.....	1-6
	makefile.win.....	1-8
	l1main.c.....	1-10
	l1rom.c.....	1-13
	l1start.trees.....	1-19
Chapter 2	Bootstrapping TriMedia in Host-Assisted Mode	
	TriMedia Initialization in Host Assisted Mode.....	2-2
	Overview	2-3
	Plug and Play BIOS.....	2-4
	BIU and Interrupt Initialization	2-6
	Putting the processor in reset.....	2-8
	Taking the processor out of reset	2-9
	tmmprun - multiprocessor download program	2-10
	Makefile	2-10
	Header files	2-11
	tmmprun main program	2-12
	Using the downloader library.....	2-16

tmcrd.c.....	2-22
Shutting down the RPC server	2-25
Implementation of POSIX system functions	2-26

Chapter 1

Bootstrapping TriMedia in Autonomous Mode

Topic	Page
Introduction	1-2
Overview of Stand-Alone Boot	1-2
Sample Programs	1-5

Introduction

Bringing up TM1000 in stand-alone mode involves a number of steps. This chapter outlines the essential steps common to different stand-alone configurations. It also includes sample programs that you can modify to suit your needs.

In order to fully understand this chapter, you must be familiar with the TM1000 architecture and you will need to have read Chapter 12 of the TM1000 Preliminary Data Book (April 1997): “System Boot,” which is the official document on both stand-alone and host-assisted boot procedures.

All the examples in this chapter refer to TCS software tools released in August 1997, or later.

Overview of Stand-Alone Boot

During power-on reset, TM1000 boot block reads some configuration information from the EEPROM through I2C. The contents of the EEPROM determine, among other things, whether TM1000 continues to boot from the EEPROM or expects another processor (such as a PC or a Mac) to complete the TM1000 boot sequence. In a host-assisted boot, the EEPROM contains just 10 bytes that set a few parameters such as `TRI_CLKIN`, `PCI Sub-system Id`, `Vendor Id`, `MM_CONFIGs`, and `PLL_RATIOs`. The task of downloading an application to SDRAM and taking TM1000 out of reset is left to a host-based program (such as **tmmon** on the PC or Mac).

In a stand-alone boot, the EEPROM contains, in addition, the initial boot program whose size is restricted to 2K bytes. This initial boot program, called *L1 boot program*, is transferred by the TM1000 boot block from EEPROM to SDRAM and then executed. It is the responsibility of the L1 boot program to load other programs (we will call them L2 programs) from any attached device, such as on-board UVEPROMs (or flash) or networks and to execute them.

Creating an EEPROM image

The L1 boot EEPROM consists of a 47-byte header followed by the L1 boot program.

EEPROM Header

Contents of the EEPROM header are documented in Chapter 12 of the TM1000 Preliminary Data Book (April 1997 edition): “System Boot.” The memory system parameters are documented in Chapter 11 of the TM1000 Preliminary Data Book: “SDRAM Memory System.”

This chapter includes a sample program *l1rom.c*, which creates an EEPROM image file (binary) of the L1 boot program. The *l1rom.c* program adds a 47-byte header to the given L1 boot program and swaps the bytes of the L1 boot program when creating the EEPROM image. *l1rom.c* uses fixed values for `TRI_CLKIN`, `PLL` clock ratios, and so on. Stand-alone system developers need to examine and change the first 8 bytes of the EEPROM header in *l1rom.c*, if necessary, to suit their system. The EEPROM header bytes are documented in Chapters 11 and 12 of the TM1000 Preliminary Data Book (April 1997 edition).

L1 Boot Program

L1 boot code needs to do some initialization of TM1000, such as setting the PCSW, `BIU_CTRL`, setting up stack and frame pointers, initializing PCI devices (if any) and copying the L2 code to SDRAM. It then jumps to the beginning of L2 code.

The sample L1 program consists of two files:

- `l1start.trees`

This file defines a function `__start()` which initializes PCSW and `BIU_CTRL`; sets up SP (stack pointer), FP (frame pointer), and RP (return pointer); and calls `L1main()`. On return from `L1main`, it jumps to the L2 load address returned by `L1main()`.

- `l1main.c`

The function `L1main()` simply copies L2 code from a PCI-slave UVEPROM to SDRAM. After copying L2 code to SDRAM, the data cache is flushed and then invalidated. After that, the instruction cache is cleared. `L1main()` returns the L2 load address to the caller, `__start()`.

Note

If you are using TM1000 chips earlier than revision 1s1.1, I2C might be in some stuck state after autoboot. The `l1main.c` file contains a simple workaround.

On the TM1000 debug board, the UVEPROM is located at (PCI) address `0xFFC0000`. The sample L1 boot code loads the sample L2 code from (PCI) address `0xFFC0000` to (SDRAM) address `0x840` (the first cache aligned address after 2 K, because L1 code can be at most, 2K bytes).

Steps in creating an EEPROM image.

1. Compile `l1start.trees` and `l1main.c` as follows.

```
cp l1start.trees l1start.t
tmcc -x -v -c -eb -DL2_LOAD_ADDR=0x840 \
      -DL2_CODE_SIZE=200000 \
      -DL2_ROM_DEV_ADDR=0xFFC00000 \
      l1start.t l1main.c
```

The L1 boot program needs to know the size of L2 code. The `tmcc` option `-DL2_CODE_SIZE=150000` defines `L2_CODE_SIZE`. The sample L2 code fits within 200000 bytes. L1 boot code sets up SP and FP starting at `MEMORY_SIZE` (defined to be 8 MB, because IREF boards have 8 MB memory). For stand-alone systems, `MMIO_BASE` is defined to be `0xEFE0000`. This value must agree with that used in `l1rom.c` as part of the 47-byte EEPROM header.

2. Link `l1start.o` and `l1main.o` and verify the executable size.

```
tmld -eb -o l1.out l1start.o l1main.o
tmsize l1.out
```

You cannot use the `tmcc` compiler driver to link the L1 boot code, because `tmcc` adds a number of options and libraries by default to the linker command line. This step just verifies that the sum of text, data, data1, and bss section sizes is less than 2K bytes.

IMPORTANT

It is important that `l1start.o` appears first in the link command before all other files that are linked. ▲

3. Relocate the executable and produce a memory image.

The executable `l1.out` produced in Step 2 has text, data, data1, and bss sections. In addition, it contains information about the executable itself. To generate a memory image, you must specify the load start address and the memory size and pass the `-mi` option to `tmld`. This concatenates the text, data, data1, and bss sections and produces a memory image. You must also define `__clock_freq_init`, `__MMIO_base_init`, and `__begin_stack_init` as download parameters (`-bdownload __clock_freq_init` etc.) and then define their values (`-tm_freq 100000000` defines the TM1000 clock frequency as 100 MHz). If you use a TM1 IREF board with an 80 MHz TM1.1 chip, change this option to `-tm_freq 80000000`. Ensure that `l1start.o` is the first file in the list of files linked. This is because TM1000 starts execution at SDRAM BASE) and you want the startup code `__start` to be located at that address.

```
tmld -eb -o "l1.mi" -bdownload __clock_freq_init -mi \
-bdownload __MMIO_base_init \
-bdownload __begin_stack_init \
-exec -start=__l1start -tm_freq 100000000 \
-mmio_base 0xEFE00000 \
-load=0,0x800000 l1start.o l1main.o
```

In the above example, memory starts at 0 and the size is 8 MB.

Note

`__clock_freq_init` is required because the TM1000 device libraries rely on this definition of the clock frequency to determine things like the number of ticks in a microsecond or the proper control value to set the video clock to 27MHz.

4. Add a 47-byte header to the memory image, swap the bytes in the L1 boot program, and produce the L1 EEPROM image. Swapping bytes of the L1 boot program is always needed because of the way the boot block transfers bytes from EEPROM to SDRAM. The `l1rom.c` sample program has hard-coded values for the 47-byte header. You might want to modify `l1rom.c` and change the first 8 bytes to suit your system. The command `l1rom l1.mi` produces the `l1.eeprom` EEPROM image file, which is a binary file that you can use to program an EEPROM part such as ATML646 24c16, using an EEPROM programmer such as BP 1200.

Sample Programs

This chapter includes the following sample programs:

Sample Programs	Description
<code>makefile.unix</code>	Makefile for SunOS and HP-UX. It is used to create L1 boot code, L2 code, EEPROM image, etc. The TCS and CC macros need to be customized for the particular compilation host platform.
<code>makefile.win</code>	Makefile for MKS Make on Windows 95/NT. It is used to create L1 boot code, L2 code, EEPROM image, etc. The TCS and CC macros need to be customized for the particular compilation host platform.
<code>l1start.trees</code>	These 2 files form the L1 boot code.
<code>l1main.c</code>	
<code>l1rom.c</code>	This program is built as a host shell command. It is used to create the L1 EEPROM image.
<code>vivot.c</code>	This file forms the L2 code (plus standard device libraries).
<code>seeval.c</code>	If you are using the SEEVAL EEPROM programmer, you need this. If not, ignore this file.

makefile.unix

```

# -----
# L2 program must be compiled to have a load address of
# L2_LOAD_ADDR, since L2_LOAD_ADDR is used in llmain.c
# -----

CP          = /bin/cp
MV          = /bin/mv
RM          = /bin/rm
CC          = /t/lang/acc

TCS        = /t/qasoft/build/tcs1.1z/1054/SunOS
TMCC       = $(TCS)/bin/tmcc
TMLD       = $(TCS)/bin/tmld
TMSIZE     = $(TCS)/bin/tmsize

L1ROM      = llrom

MMIO_BASE  = 0xefe00000
SDRAM_BASE = 0x0
SDRAM_LIMIT = 0x800000
TM_FREQ    = 100000000

# -----
# L1 boot program can be 2048 bytes long atmost.
# L2_LOAD_ADDR is the next cache aligned address, i.e 2112
# -----

L2_LOAD_ADDR = 2112
L2_CODE_SIZE = 150000
L2_ROM_DEV_ADDR = 0xffc00000

ENDIAN       = -el

L1_CFLAGS   = -v $(ENDIAN) -host nohost \
              -DL2_LOAD_ADDR=$(L2_LOAD_ADDR) \
              -DL2_CODE_SIZE=$(L2_CODE_SIZE) \
              -DL2_ROM_DEV_ADDR=$(L2_ROM_DEV_ADDR)

L1_LDFLAGS  = $(ENDIAN) -btype boot \
              -bdownload __clock_freq_init \
              -bdownload __MMIO_base_init \
              -bdownload __begin_stack_init \
              -exec -start=__start

L1_MIFLAGS  = $(ENDIAN) \
              -bdownload __clock_freq_init \
              -bdownload __MMIO_base_init \
              -bdownload __begin_stack_init \
              -mi -exec -start=__start \
              -tm_freq $(TM_FREQ) \
              -mmio_base $(MMIO_BASE) \
              -load=$(SDRAM_BASE),$(SDRAM_LIMIT)

L2_CFLAGS   = -v $(ENDIAN) -I$(TCS)/include/Win95 \
              -host nohost \

```



```

-DMMIO_BASE_ADDR=$(MMIO_BASE)

L2_MIFLAGS      = $(ENDIAN)                \
                -bdownload __clock_freq_init \
                -mi -exec -start=__start    \
                -tm_freq $(TM_FREQ)       \
                -mmio_base $(MMIO_BASE)    \
                -load=$(L2_LOAD_ADDR),$(SDRAM_LIMIT)

# -----

ll.out: llstart.trees llmain.c
@echo ""
@echo making $@
$(RM) -f llstart.t
$(CP) llstart.trees llstart.t
$(TMCC) -x $(L1_CFLAGS) -c llstart.t llmain.c
$(TMLD) $(L1_LDFLAGS) -o $@ llstart.o llmain.o
$(TMSIZE) $@

ll.mi: llstart.trees llmain.c
@echo ""
@echo making $@
$(RM) -f llstart.t
$(CP) llstart.trees llstart.t
$(TMCC) -x $(L1_CFLAGS) -c llstart.t llmain.c
$(TMLD) -o $@ $(L1_MIFLAGS) llstart.o llmain.o

ll.eeprom: ll.mi $(L1ROM)
@echo ""
@echo "Adding 47 bytes autoboot protocol header and swapping bytes"
$(L1ROM) ll.mi

$(L1ROM): llrom.c
@echo ""
@echo making $@
$(CC) -o $@ -DSDRAM_BASE=$(SDRAM_BASE) -DSDRAM_LIMIT=$(SDRAM_LIMIT)
llrom.c

# -----

vivot.out: vivot.c
$(TMCC) $(L2_CFLAGS) -o $@ vivot.c

vivot.mi: vivot.c
$(TMCC) $(L2_CFLAGS) -o $@ -tmld $(L2_MIFLAGS) -- vivot.c

# -----

clean:
$(RM) -f $(L1ROM) *.o *.t *.i *.s *.eeprom *.out *.mi *.dump

```

makefile.win

```

# -----
# L2 program must be compiled to have a load address of
# L2_LOAD_ADDR, since L2_LOAD_ADDR is used in llmain.c
# -----

CP           = cp
MV           = mv
RM           = rm
CC           = cc

TCS          = C:/TriMedia
TMCC         = $(TCS)/bin/tmcc
TMLD         = $(TCS)/bin/tmld
TMSIZE       = $(TCS)/bin/tmsize
L1ROM        = llrom.exe

MMIO_BASE    = 0xfe00000
SDRAM_BASE   = 0x0
SDRAM_LIMIT  = 0x800000
TM_FREQ      = 100000000

# -----
# L1 boot program can be 2048 bytes long atmost.
# L2_LOAD_ADDR is the next cache aligned address, i.e 2112
# -----

L2_LOAD_ADDR = 2112
L2_CODE_SIZE = 150000
L2_ROM_DEV_ADDR = 0xffc00000

ENDIAN        = -el

L1_CFLAGS     = -v $(ENDIAN) -host nohost \
                -DL2_LOAD_ADDR=$(L2_LOAD_ADDR) \
                -DL2_CODE_SIZE=$(L2_CODE_SIZE) \
                -DL2_ROM_DEV_ADDR=$(L2_ROM_DEV_ADDR)

L1_LDFLAGS    = $(ENDIAN) -btype boot \
                -bdownload __clock_freq_init \
                -bdownload __MMIO_base_init \
                -bdownload __begin_stack_init \
                -exec -start=_start

L1_MIFLAGS    = $(ENDIAN) \
                -bdownload __clock_freq_init \
                -bdownload __MMIO_base_init \
                -bdownload __begin_stack_init \
                -mi -exec -start=_start \
                -tm_freq $(TM_FREQ) \
                -mmio_base $(MMIO_BASE) \
                -load=$(SDRAM_BASE),$(SDRAM_LIMIT)

L2_CFLAGS     = -v $(ENDIAN) -I$(TCS)/include/Win95 \
                -host nohost \
                -DMMIO_BASE_ADDR=$(MMIO_BASE)

```

```

L2_MIFLAGS      = $(ENDIAN)                \
                -bdownload __clock_freq_init \
                -mi -exec -start=_start     \
                -tm_freq $(TM_FREQ)        \
                -mmio_base $(MMIO_BASE)     \
                -load=$(L2_LOAD_ADDR),$(SDRAM_LIMIT)

# -----

l1.out: llstart.trees llmain.c
    @echo ""
    @echo making $@
    $(RM) -f llstart.t
    $(CP) llstart.trees llstart.t
    $(TMCC) -x $(L1_CFLAGS) -c llstart.t llmain.c
    $(TMLD) $(L1_LDFLAGS) -o $@ llstart.o llmain.o
    $(TMSIZE) $@

l1.mi: llstart.trees llmain.c
    @echo ""
    @echo making $@
    $(RM) -f llstart.t
    $(CP) llstart.trees llstart.t
    $(TMCC) -x $(L1_CFLAGS) -c llstart.t llmain.c
    $(TMLD) -o $@ $(L1_MIFLAGS) llstart.o llmain.o

l1.eeprom: l1.mi $(L1ROM)
    @echo ""
    @echo "Adding 47 bytes autoboot protocol header and swapping bytes"
    $(L1ROM) l1.mi

$(L1ROM): llrom.c
    @echo ""
    @echo making $@
    $(CC) -o $@ -DSDRAM_BASE=$(SDRAM_BASE) -DSDRAM_LIMIT=$(SDRAM_LIMIT)
llrom.c

# -----

vivot.out: vivot.c
    $(TMCC) $(L2_CFLAGS) -o $@ vivot.c

vivot.mi: vivot.c
    $(TMCC) $(L2_CFLAGS) -o $@ -tmld $(L2_MIFLAGS) -- vivot.c

# -----

clean:
    $(RM) -f $(L1ROM) *.obj *.o *.t *.i *.s *.eeprom *.out *.mi *.dump

```

l1main.c

```

/*
 * +-----+
 * | Copyright (c) 1995,1996,1997 by Philips Semiconductors.
 * |
 * | This software is furnished under a license and may only be used
 * | and copied in accordance with the terms and conditions of such a
 * | license and with the inclusion of this copyright notice. This
 * | software or any other copies of this software may not be provided
 * | or otherwise made available to any other person. The ownership
 * | and title of this software is not transferred.
 * |
 * | The information in this software is subject to change without
 * | any prior notice and should not be construed as a commitment by
 * | Philips Semiconductors.
 * |
 * | This code and information is provided "as is" without any
 * | warranty of any kind, either expressed or implied, including but
 * | not limited to the implied warranties of merchantability and/or
 * | fitness for any particular purpose.
 * |
 * +-----+
 *
 * Module name           : l1main.c
 *
 * Module type          : IMPLEMENTATION
 *
 * Title                : L1 boot code
 *
 * Last update         : 15 July 1997
 *
 * Description          :
 *
 *                       L1 boot code.
 *                       Copies L2 code from a PCI-slave UVEPROM
 */

#include <tml/mmio.h>

/* downloader symbols */
/* Patched when creating a memory image file using tmltd */

extern      long      _clock_freq_init[];
extern unsigned int  _begin_stack_init[];
extern unsigned int  _MMIO_base_init[];

/* MACROS */

#define CACHE_BL_SIZE      64
#define VO_FREQUENCY      27000000.0 /* 27 MHz */

/* globals */

unsigned long  _clock_freq = (unsigned long)  _clock_freq_init;
volatile UInt32 * _MMIO_base = (volatile UInt32 *) _MMIO_base_init;

custom_op void dcb      (unsigned, int);

```

```

custom_op void dinvalid (unsigned, int);
custom_op void iclr      (void);

/* local variables */

volatile static unsigned int  dummy;

/*
 * copyback_dcache (unsigned addr, int nbytes)
 * 1. addr must be cache aligned.
 * This function flushes nbytes starting at addr to memory.
 *
 * L1 boot code copies L2 code from some device
 * This needs to be flushed to memory before jumping to the
 * L2 load address
 *
 */

static void
copyback_dcache(unsigned addr, int n)
{
    int i;

    for (i = 0; i < n; i = i + CACHE_BL_SIZE)
        dcb(0, addr + (unsigned) i);
}

/*
 * iclr is in a separate function to ensure that it is in a
 * dtree by itself
 */

static void
clear_icache(void)
{
    iclr();
}

/* Copies L2 code via JTAG to SDRAM */

unsigned int Llmain ()
{
    int i;
    unsigned char  byte;
    unsigned int  *base_addr = (unsigned int *) L2_ROM_DEV_ADDR;
    unsigned char *load_addr = (unsigned char *) L2_LOAD_ADDR;

#if 0

    /* Not needed for TM1s 1.1 chip.
     * In previous versions, autoboot leaves IIC in stuck state.
     * Steps 1, 2, and 3 will reset IIC.
     */

```

```

/* Step 1: Set up VO clock */
MMIO(VO_CLOCK) = (unsigned int) (0.5 + (1431655765.0 *
                                     VO_FREQUENCY / _clock_freq));

MMIO(VO_CTL) = 0x02700000;

/* and wait for vo clock to stabilize */
for (i = 0; i < 1000 * 1000; i++)
    dummy++;

/* Step 2. Toggle I2C control */
MMIO(IIC_CTL) = 0;
MMIO(IIC_CTL) = 0x03c00001;

/* Step 3. Single I2C read and throw away */
MMIO(IIC_AR) = 0x71000100;
dummy = MMIO(IIC_DR);

#endif

/* Load L2 code from an attached PCI device */

/* start copying of L2 code to sdram */
/* Assumes TM1 debug board schematics.
 * Assumes L2 program is in a single UVEPROM plugged into
 * byte 3 slot. The other 3 slots (which supply bytes 0, 1, and 2
 * of a word loaded from PCI) are empty.
 */

for (i=0; i < L2_CODE_SIZE; i++) {
#ifdef __BIG_ENDIAN__
    byte = base_addr[i] & 0xFF;
#else
    byte = (base_addr[i] >> 24) & 0xFF;
#endif
    load_addr[i] = byte;
}

/* flush data cache */
copyback_dcache(L2_LOAD_ADDR, L2_CODE_SIZE);

/* clear any interrupts */
MMIO(ICLEAR) = 0xffffffff;

clear_icache();
/*
 * Return from Llmain() causes L2 code to be executed.
 */
return L2_LOAD_ADDR;
}

```

l1rom.c

```

/*
 * copyright (c) 1995,1996,1997 by Philips Semiconductors
 *
 * +-----+
 * | This software is furnished under a license and may only be used |
 * | and copied in accordance with the terms and conditions of such |
 * | a license and with the inclusion of the this copy right notice. |
 * | this software or any other copies of this software may not be |
 * | provided or otherwise made available to any other person. The |
 * | ownership and title of this software is not transferred. |
 * +-----+
 *
 * Module name:
 *     l1rom.c
 *
 * Author:
 *     Renga Sundararajan
 *     renga.sundararajan@sv.sc.philips.com
 *
 * Description:
 *     Generates an EEPROM image (binary file)
 *
 * Input:
 *     f.mi - generated using -mi option of tmlld
 *
 * Output:
 *     f.eeprom
 *     f.eeprom contains 47 header bytes as required by
 *     TMI autoboot protocol, followed by the program bytes
 *     (bytes are swapped as required by boot)
 *
 * Assumption:
 *     1. f.mi contains less than 2001 bytes, divisible by four
 *        (as required by the boot protocol).
 *     2. short is 2 bytes.
 *
 * Update History:
 *     May 23, 1998   Modify to accept SDRAM_BASE and SDRAM_LIMIT
 *                   macros values from the Makefile.
 *                   rudy.wang@sv.sc.philips.com
 */

#if defined(__sun)
#include <unistd.h>
#endif
#include <sys/stat.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

#if !defined(SDRAM_BASE) || !defined(SDRAM_LIMIT)

```

```

#error "Macros SDRAM_BASE and SDRAM_LIMIT have to be defined for this file to
build"
#endif

#define MAX_FILE_SIZE          2000
#define MAX_EPROM_SIZE        (1024 * 2)
#define BUF_SIZE              MAX_EPROM_SIZE
#define NUM_HEADER_BYTES      47

#define MSB_1ST(n)             (unsigned char)(((n) >> 24) & 0xff)
#define MSB_2ND(n)             (unsigned char)(((n) >> 16) & 0xff)
#define MSB_3RD(n)             (unsigned char)(((n) >> 8) & 0xff)
#define MSB_4TH(n)             (unsigned char)((n) & 0xff)

static void
basename(char *fname, char *bname)
{
    char *ptr, *ptr2;
    int i;

    if ((ptr = strrchr(fname, '.')) == NULL) {
        strcpy(bname, fname);
    }
    else {
        for (ptr2 = fname, i = 0; ptr2 != ptr; ptr2++, i++) {
            bname[i] = *ptr2;
        }
        bname[i] = '\0';
    }
}

int
read_file(unsigned char *buffer, char *filename)
{
    FILE          *Llfp;
    int           Llfd, n, nbytes;
    struct stat   file_stat;

    if ((Llfd = open(filename, O_RDONLY)) == -1) {
        fprintf(stderr, "Unable to open file: %s\n", filename);
        fprintf(stderr, "File doesn't exist or not readable\n");
        exit(1);
    }

    if (fstat(Llfd, &file_stat) {
        fprintf(stderr, "Unable to fstat file: %s\n", filename);
        exit(1);
    }
    nbytes = (unsigned long)file_stat.st_size;
    close(Llfd);

    if ((Llfp = fopen(filename, "rb")) == NULL) {
        fprintf(stderr, "Unable to open file: %s\n", filename);
        fprintf(stderr, "File doesn't exist or not readable\n");
        exit(1);
    }
}

```



```

if (nbytes > MAX_FILE_SIZE) {
    fprintf(stderr, "File has %5d bytes. must be less than %5d bytes\n",
        nbytes, MAX_FILE_SIZE);
    exit(1);
}

n = fread(buffer, 1, nbytes, L1fp);
if (n != nbytes) {
    fprintf(stderr, "Unable to read %5d bytes, error no: %5d\n",
        nbytes, errno);
    exit(1);
}

fprintf(stderr, "        Program Size: %5d bytes\n", nbytes);
return nbytes;
}

/*
 * Header bytes are hard-coded. Read the TM-1 boot block paper
 * to see what needs to go in here for AUTO boot.
 */
int
output_eeprom_header(int nbytes, unsigned char obuffer[])
{
    int i = 0;

    /* Output eeprom header bytes 0 thru 46, as per
     * Chapter 12 of TM 1000 Data Book (April 1997 edition).
     * These go into output array index 0 onwards
     */

    /* 0xc8 for 50 and 40 MHz TRI_CLKIN. 0xcc for 33 MHz */

    obuffer[i++] = 0xc8;    /* 0 */

    /* Sub-system Id */

    obuffer[i++] = 0x00;    /* 1 */
    obuffer[i++] = 0x03;    /* 2 */

    /* Sub-system Vendor Id */

    obuffer[i++] = 0x11;    /* 3 */
    obuffer[i++] = 0x31;    /* 4 */

    /* Bytes 5 6 7: MM Config register */

    /*
     * Byte 6 and 4 bits of byte 7 determine refresh rate.
     * The refresh rate is 4c4 for 80MHz sdram clock, 384 for 60 Mhz.
     * Use Table 11-10 Refresh Intervals of TM 1000 Preliminary Data
     * for other SDRAM clock speeds and interpolate for speeds not
     * mentioned in that table
     */

    obuffer[i++] = 0x00;    /* 5 */
    obuffer[i++] = 0x4c;    /* 6 */

```

```

obuffer[i++] = 0x44;      /* 7 */

/* Byte 8: PLL Ratios */

obuffer[i++] = 0x00;      /* 8 */

/*
 * Byte 9: Most significant bit is 1 for stand-alone boot
 * Least 3 bits of byte 9 and 8 bits of byte 10 determine
 * L1 boot program code size. 11 bits == 2K bytes at most
 */

obuffer[i++] = (0x80 | ((nbytes >> 8) & 0x7));
obuffer[i++] = (nbytes & 0xfc);

/* MMIO base register address, MSB first */
obuffer[i++] = 0xef;      /* 11 */
obuffer[i++] = 0xf0;      /* 12 */
obuffer[i++] = 0x04;      /* 13 */
obuffer[i++] = 0x00;      /* 14 */
/* MMIO base register value, MSB first */
obuffer[i++] = 0xef;      /* 15 */
obuffer[i++] = 0xe0;      /* 16 */
obuffer[i++] = 0x00;      /* 17 */
obuffer[i++] = 0x00;      /* 18 */

/* DRAM base register address, MSB first */
obuffer[i++] = 0xef;      /* 19 */
obuffer[i++] = 0xf0;      /* 20 */
obuffer[i++] = 0x00;      /* 21 */
obuffer[i++] = 0x00;      /* 22 */
/* DRAM base register value, MSB first */
obuffer[i++] = MSB_1ST(SDRAM_BASE); /* 23 */
obuffer[i++] = MSB_2ND(SDRAM_BASE); /* 24 */
obuffer[i++] = MSB_3RD(SDRAM_BASE); /* 25 */
obuffer[i++] = MSB_4TH(SDRAM_BASE); /* 26 */

/* DRAM limit register address, MSB first */
obuffer[i++] = 0xef;      /* 27 */
obuffer[i++] = 0xf0;      /* 28 */
obuffer[i++] = 0x00;      /* 29 */
obuffer[i++] = 0x04;      /* 30 */
/* DRAM limit register value, MSB first */
obuffer[i++] = MSB_1ST(SDRAM_LIMIT); /* 31 */
obuffer[i++] = MSB_2ND(SDRAM_LIMIT); /* 32 */
obuffer[i++] = MSB_3RD(SDRAM_LIMIT); /* 33 */
obuffer[i++] = MSB_4TH(SDRAM_LIMIT); /* 34 */

/* DRAM cacheable limit reg address, MSB first */
obuffer[i++] = 0xef;      /* 35 */
obuffer[i++] = 0xf0;      /* 36 */
obuffer[i++] = 0x00;      /* 37 */
obuffer[i++] = 0x08;      /* 38 */
/* DRAM cacheable limit reg value, MSB first */
obuffer[i++] = MSB_1ST(SDRAM_LIMIT); /* 39 */ /* assumes to be the same as
SDRAM_LIMIT */
obuffer[i++] = MSB_2ND(SDRAM_LIMIT); /* 40 */

```

```

obuffer[i++] = MSB_3RD(SDRAM_LIMIT); /* 41 */
obuffer[i++] = MSB_4TH(SDRAM_LIMIT); /* 42 */

/* DRAM base reg value, MSB first */
obuffer[i++] = MSB_1ST(SDRAM_BASE); /* 43 */
obuffer[i++] = MSB_2ND(SDRAM_BASE); /* 44 */
obuffer[i++] = MSB_3RD(SDRAM_BASE); /* 45 */
obuffer[i++] = MSB_4TH(SDRAM_BASE); /* 46 */

if (i != NUM_HEADER_BYTES) {
    fprintf(stderr, "Error: header bytes count = %5d, shd be %5d\n",
            i, NUM_HEADER_BYTES );
    exit(1);
}
fprintf(stderr, "EEPROM Header Size: %5d bytes\n", NUM_HEADER_BYTES);

return i;
}

int
main(int argc, char **argv)
{
    int            i, j, file_size;
    int            header_bytes;
    FILE           *fp;
    char           *o_file_name, *cp;
    unsigned char  ibuffer[BUF_SIZE] = {0};
    unsigned char  obuffer[BUF_SIZE] = {0};

    if (argc < 2) {
        fprintf(stderr, "Usage: llprom file.mi \n");
        exit(1);
    }

    /* find output file name */

    i = strlen(argv[1]);

    /* .eeprom extension needs 7+1 chars */
    o_file_name = (char *)malloc(i+8);
    if (o_file_name == NULL) {
        fprintf(stderr, "unable to malloc\n");
        exit(1);
    }

    /* skip all directory names */

    if ((cp = strrchr(argv[1], '/')) == NULL) {
        cp = argv[1];
    }
    basename(cp, o_file_name);
    i = strlen(o_file_name);
    o_file_name[i++] = '.';
    o_file_name[i++] = 'e';
    o_file_name[i++] = 'e';
    o_file_name[i++] = 'p';
    o_file_name[i++] = 'r';

```

```

o_file_name[i++] = 'o';
o_file_name[i++] = 'm';
o_file_name[i++] = '\0';

if ((fp = fopen(o_file_name, "wb")) == NULL) {
    fprintf(stderr, "Could not open (binary) file %s for write\n",
            o_file_name);
    exit(1);
}

file_size = read_file(ibuffer, argv[1]);

header_bytes = output_eeprom_header(file_size, obuffer);

/*
 * Output 4 bytes at a time.
 * Swap the byte ordering since boot block expects
 * words in eeprom to have MSB first and LSB last.
 */

for (i = header_bytes; i < file_size + header_bytes; i += 4) {
    obuffer[i] = ibuffer[i+3-header_bytes];
    obuffer[i+1] = ibuffer[i+2-header_bytes];
    obuffer[i+2] = ibuffer[i+1-header_bytes];
    obuffer[i+3] = ibuffer[i-header_bytes];
}

j = fwrite(obuffer, sizeof(char), file_size + header_bytes, fp);
if (j != file_size + header_bytes) {
    fprintf(stderr, "Unable to write %5d bytes. Wrote %5d \n",
            file_size + header_bytes);
    exit(1);
}

close(fp);
return 0;
}

```

l1start.trees

```

(*)
* +-----+
* | Copyright (c) 1995,1996,1997 by Philips Semiconductors. |
* | |
* | This software is furnished under a license and may only be used |
* | and copied in accordance with the terms and conditions of such a |
* | license and with the inclusion of this copyright notice. This |
* | software or any other copies of this software may not be provided |
* | or otherwise made available to any other person. The ownership |
* | and title of this software is not transferred. |
* | |
* | The information in this software is subject to change without |
* | any prior notice and should not be construed as a commitment by |
* | Philips Semiconductors. |
* | |
* | This code and information is provided "as is" without any |
* | warranty of any kind, either expressed or implied, including but |
* | not limited to the implied warranties of merchantability and/or |
* | fitness for any particular purpose. |
* +-----+
*)

(*)
* Module name           : l1.trees
*
* Title                 : L1 startup code
*
* Last update           : Tue Jul 15 09:53:10 PDT 1997
*
*)

(*-----*)
(* Copy this file to l1start.t and then compile as tmcc -x l1start.t *)

(* Compile this file with tmcc -x ....
* The -x flag tells tmcc to run cpp on this file before assembly.
* The -el or -eb option causes tmcc to define cpp flag
*   __LITTLE_ENDIAN__ or __BIG_ENDIAN__
* and the right INITIAL_PCSW_VALUE and INITIAL_BIU_CTL_VALUE get used.
*
* Running cpp on this file (via tmcc) also causes
* symbolic constants such as BIU_CTL etc to be resolved
* (these are defined in TCS_INSTAL_DIR/tml/mmio.h).
*)

#define __TMAS__
#include <tml/mmio.h>

```

```

(*-----*)
#ifdef __BIG_ENDIAN__
#define INITIAL_PCSW_VALUE    0x0800    (* S *)
#define INITIAL_BIU_CTL_VALUE 0x0200    (* Host nable *)
#else
#define INITIAL_PCSW_VALUE    0x0A00    (* CS + Byte Sex *)
#define INITIAL_BIU_CTL_VALUE 0x0201    (* Host Enable + Byte Swap Enable *)
#endif
(*-----*)

        .text
        .global __start
        .global _Llmain                (* defined in llmain.c *)

__start:
__start_DT_0:
entree(0)
.treeinfo regmask "0x0000000000000000fffffffffffffff";

        (* iclr just to be sure *)

        10 iclr;

        20 uimm (INITIAL_PCSW_VALUE);
        21 uimm (-1);
        22 writepcsw 20 21;
        (* set up stack: FP and SP *)

        30 uimm (__begin_stack_init);
        33 wrreg (3) 30;
        34 wrreg (4) 30;

        (* set up return pointer *)

        40 uimm(__start_DT_1);
        41 wrreg (2) 40;

        (* configure BIU CTL *)

        50 uimm (BIU_CTL);
        51 uimm (__MMIO_base_init);
        52 iadd 50 51;
        53 uimm (INITIAL_BIU_CTL_VALUE);
        54 st32 52 53;

        gotree {_Llmain}

enttree

```

```
(* Control returns to __start_DT_1 when Llmain() is done with loading
 * L2 code into SDRAM. Jump to L2_LOAD_ADDR
 * returned in register 5
 *)

__start_DT_1:
entree(0)
.treeinfo regmask "0x0000000000000000fffffffffffffff";

        12 rdreg (5); (* L2 Load Address *)
        cgoto 12
enttree
```


Chapter 2

Bootstrapping TriMedia in Host-Assisted Mode

Topic	Page
TriMedia Initialization in Host Assisted Mode	2-2
Overview	2-3

TriMedia Initialization in Host Assisted Mode

The purpose of this document is to explain how the IREF board gets initialized in host assisted mode. In this mode the program is downloaded using the PCI bus. Host processor control over the program is ensured by writing to the PCI bus using the MMIO registers. Host assisted mode corresponds to a **tmcc** command line with the `-host Win95`, `-host WinNT`, or `-host MacOS` options.

The TriMedia processor begins in reset and is initialized as a result of actions by the host. The initial state of the processor is defined by the first 10 bytes of the EEPROM.

The processor state is initialized as the result of actions in several places. These include: the plug and play BIOS, the OS configuration manager, a kernel driver, and the user program. On Windows 95, the kernel driver is `vtmman.vxd` and the user program is `tmgmon.exe`.

The information in this document is useful for anyone that needs to understand the TriMedia processor at a systems level.

For more information about the TM-1000 implementation of PCI, refer to chapter 10 of the databook. Figure 10-2 explains the PCI configuration registers. Chapter 12 describes the boot process. You may want to refer to the sections on the host assisted boot and on the EEPROM format.

You may also want to refer to the document PCI design Issues for Windows 95, and Windows NT, Microsoft Corporation, 1/25/95 (rev 1.0) for more information on PCI configuration in a PC environment.

For more information about the PCI local bus, refer to the PCI Local Bus Specification, version 2.1, available from the PCI consortium, tel: (503) 797 4207, fax (503) 234 6762.

For information about Microsoft Visual C++ (MSVC++) command line options, type:

```
cl /help
```

For information about Microsoft LINK command line options, type:

```
link /help
```

For information about the downloader library, see `<tmlib/TMDownloader.h>`.

For information about the TMMAN API read the TriMedia Software Reference. You can also refer to the `<Win95/tmman32.h>` and `<Win95/tmwincom.h>` header files in the release.

For information about the object file format and section types, refer to the **tmld** man page.

Overview

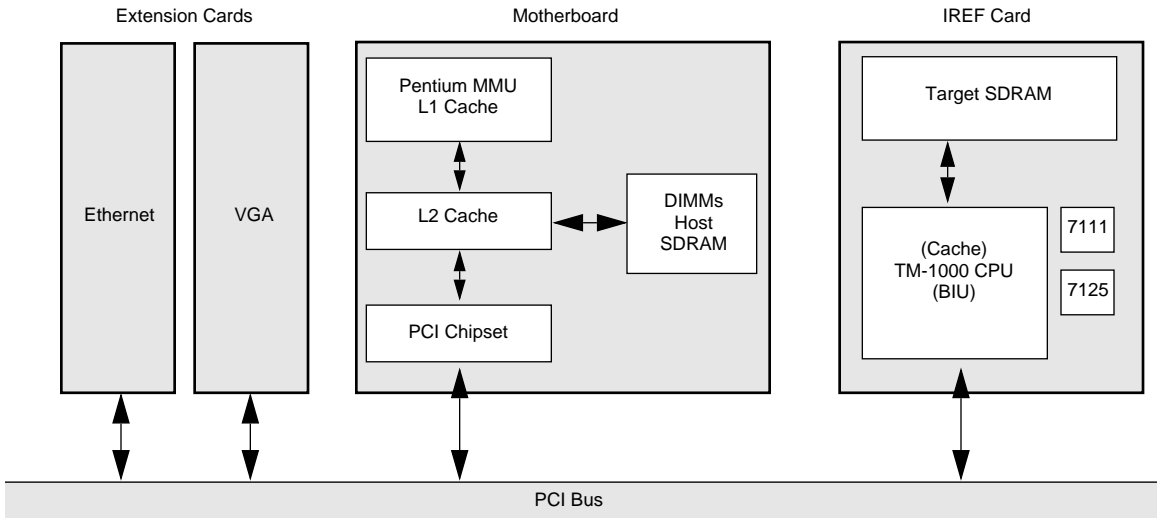


Figure 2-1

Figure 2-1 gives an overview of a typical host assisted system. The BIU (Bus Interface Unit) and the PCI chipset on the IREF board are equivalents.

In a host assisted system, the TM-1000 is initialized over the PCI bus. A Pentium is being used in the example above.

Essentially, booting TriMedia in host assisted mode requires nothing more than loading a boot image into memory and taking the processor out of reset. This can be done by clearing BIU set reset and setting clear reset. However, several things do complicate matters.

First of all, the base address of the DRAM on the board and the MMIO registers is not fixed but determined at system startup. This is because of the plug and play nature of the PCI bus and it is done in the BIOS and the OS. Finding out the actual addresses assigned requires querying the OS configuration manager. Under Windows this is done by **tmman**.

Secondly, the DRAM and the MMIO on the board needs to be mapped in virtual memory for the Pentium processor to access it. Under Windows, this requires a kernel mode driver.

Thirdly, the TriMedia downloader library must be used to construct the boot image. There are three reasons for this.

- The linker output is relocatable and needs to be made absolute.

- Symbols in the boot image needs to be patched for it to work. For example, for the processor to access its own registers MMIO_base needs to be patched.
- Special treatment is needed for cache locked and uncacheable memory and for shared sections for multiprocessors.

The downloader library depends on the object format library to read the executable. The same downloader is used to load boot images, applications and TriMedia dynamic linked libraries (dlls). For multiprocessors, processors are loaded individually and a shared section table is used for global addresses.

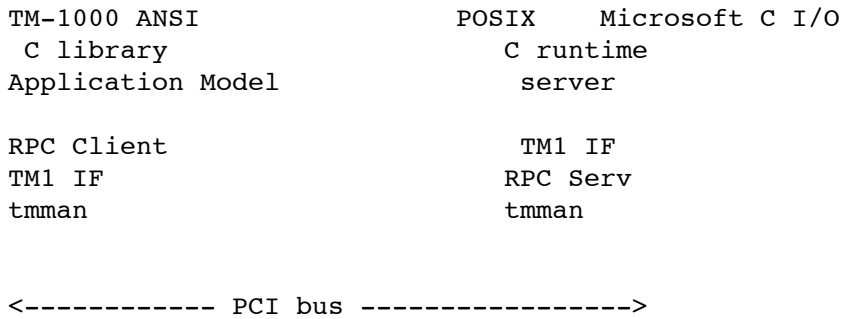


Figure 2-2

Fourthly, more functionality is required for system services (files, I/O) Figure 2 shows how system calls on the host are implemented.

I/O calls in the ANSI C library are mapped to system calls. They are transmitted as remote procedure calls (RPCs) using tmman to the host. On the host, the call is executed using the C run time server. Microsoft I/O is used for access to console windows and for files that can be redirected. Implementing RPC requires the ability to install an interrupt handler on the host.

The IPENDING, IMASK, and ICLEAR MMIO registers can be programmed on the host to generate a host -> TriMedia interrupt dynamically. The interrupt vectors can be reprogrammed also. Interrupt pin A is used for TriMedia -> host interrupts. For more information, see chapters 3 and 10 of the databook.

The way downloading works means that a boot image that has been constructed in memory can be written to disk and executed simply by restoring and clearing reset.

Plug and Play BIOS

The PC BIOS allocates base addresses and interrupts for all cards using

a technique called "plug and play". The interrupt vector is allocated by the plug and play BIOS also.

The following elements of the PCI configuration are significant. For retrieving these parameters, please refer to the PCI specification. The command `DOSPCI` in the `bin` directory of the Win95 release can be used to read the PCI registers. Here is an example.

```
[C:/Trimedia/bin] dospci

PCI Configuration Tool - Copyright (c) Philips Semiconductors 1996
PCI VendorID [1131] : DeviceID [5400] : Bus#[00] : Dev#[0e] : Func#[0]
PCI Reg#[00] : Offset[00] : Value [54001131]
PCI Reg#[01] : Offset[04] : Value [02000116]
PCI Reg#[02] : Offset[08] : Value [04800091]
PCI Reg#[03] : ...

[C:/Trimedia/bin]

PCI Register 0, bits 0 .. 16 : vendor ID
                      bits 16 .. 31 : Device ID
```

The PCI specification identifies peripherals using a device ID, vendor ID. The device ID identifies the silicon. The device for TM-1000 is 5400 and the vendor ID is 1131 (Philips).

```
Register 2, bits 0 .. 7 : Revision ID register
```

The 8 lower bits identify the CPU version (TM1, TM1S). Bits 7-6 indicate the fab. ST (Crolles) is 00, MOS4 is 01, TSMC, is 10, and 11 is unused. Bits 5-4 indicates the all layer revision. CTC/TM1 is 00, TM1S is 01, TM1C is 10, and 11 is unused. The four last bits indicates the metal layer revision. 0000 is revision 0.

```
Register 4, bits 0 .. 31 : SDRAM Base Physical Address
Register 5, bits 0 .. 31 : MMIO Base Physical Address
```

During startup, the card is accessed using the slot number of the PCI board in PCI configuration space. This is because the address is not allocated yet. PCI cards can have up to six base addresses. The TM-1000 IREF card has two (`MMIO_BASE`) and `DRAM_BASE`, corresponding to registers 4 and 5, above).

The necessary address range is determined as follows. The BIOS writes all 1's to these registers. The values that are read back tell the BIOS how much memory needs to be allocated, and the alignment to use. For example, writing `FFFFFFFF` and reading back `FF000000` means that 16 megabytes need to be allocated. Natural alignment is enforced (e.g. 16 megabytes need to be allocated on a 16 megabyte boundary).

Win16 and DOS apps can query the PCI configuration space registers using the call `int 1A`. For more information, refer to the PCI BIOS specification. Kernel mode applications

can query the Win95 configuration manager. **tmman** provides an API to query the address ranges (`tmDSPGetCaps`).

```
Register B, bits 0 .. 15 SubSystem Vendor ID
           bits 16 .. 31 Subsystem ID
```

These two fields identify the manufacturer and subsystem ID (board ID). They correspond to bytes 1-4 of the EEPROM. They can be used by software to distinguish between different boards. Board manufacturers should request a SubSystem Vendor ID from Philips.

To find out how to obtain a board ID, contact TriMedia Customer Support. Once an ID has been maintained, management of the subsystem space is the board manufacturers responsibility.

```
Register f, bits 0 .. 7 : Interrupt line register.
```

This determines the value to use for host interrupts. Note that TMMAN does not support sharing of interrupts.

The value in registers 4 and 5, and F are allocated by the PCI BIOS as part of the setup. The values allocated by the BIOS are used by the Win95 configuration manager. The exact way this is done is documented in "PCI Design Issues for Windows 95 and Windows NT" (Microsoft Corporation, 1/25/95) (document is available from Microsoft).

Depending on the BIOS, and exact PC configuration, plug and play may not always work. In this case, the configuration needs to be changed so that these are not allocated automatically. This is done using the Windows 95 Device Manager (Start -> Settings -> Control Panel -> System -> Device Manager). To disable automatic selection of the base address, the Resources menu needs to be selected and changed. Both the base address and the interrupt number may need to be allocated manually.

BIU and Interrupt Initialization

The TM-1 processor comes up in big endian mode. Depending on the endianness of the host processor, the BIU control register needs to be reconfigured. After this write, all further accesses should be done in big endian format.

On a little endian processor, this write has to be done in big endian format. The `BYTESWAP` macro converts the ordering.

```
#define BYTESWAP(x)
    ((x) << 24 | ((x) & 0xFF00) << 16 | ((x) & 0xFF0000) >> 8 | \
     ((x) & 0xFF000000) >> 24 )

VOID    halRegisterInit ( PVOID pvObject, DWORD dwSDRAMPhys,
                          DWORD dwSDRAMCacheLimit, DWORD dwMMIOPhys )
{
    MMIO.pVIC= dwMMIOBase + 0x100800;
    MMIO.pTimers= dwMMIOBase + 0x100c00;
    MMIO.pDebug= dwMMIOBase + 0x101000;
    MMIO.pBIU= dwMMIOBase + 0x103004;
    MMIO.pAudioIn= dwMMIOBase + 0x101c00;
    MMIO.pAudioOut = dwMMIOBase + 0x102000;
    MMIO.pCache= dwMMIOBase + 0x100000);
```

This initializes pointers to the MMIO registers of the different peripherals.

```
FirstTimeReset = !(MMIO.pBIU->dwBIUControl & (BIU_SE|BIU_HE));
```

The Windows 95 driver checks whether the BIU control register HE and SE bits are set. If these bits are not set, it assumes that we are just doing a reboot and that none of the registers needs to be initialized.

```
if ( FirstTimeReset)
    MMIO.pBIU->dwBIUControl = BYTESWAP (BIU_SE | BIU_HE | BIU_SR );
```

This turns on the BIU byte swap enable bit, host enable, and set reset bits.

```
MMIO.pCache->dwDRAMCacheableLimit = dwSDRAMPhys + dwSDRAMSize;
MMIO.pCache->dwDRAMLimit = dwSDRAMPhys + dwSDRAMSize;
```

The DRAM Limit and DRAM Cacheable Limit registers are set to the end of memory.

```
MMIO.pVIC->dwIMask = 0;
MMIO.pVIC->dwIClear = 0xFFFFFFFF;
```

Writing zeroes to the `IMASK` registers ensures that all interrupts are off. Note that if interrupts need to be generated from the host to the TM processor, then the relevant bits in `IMASK` need to be set. Writing all ones to the `ICLEAR` register ensures that all the pending interrupts are cleared. For more information, see Figure 3-7 of the databook.

Putting the processor in reset

In what follows, figure references are to the databook.

The following code from <tmhal.c> puts the processor in a reset state.

```
VOID halDSPStop ( PVOID pvObject ) {
    MMIO.pVIC->dwIMask = (0x0);
    MMIO.pVIC->dwIClear = 0xffffffff;
    MMIO.pBIU->dwBIUControl &= (~BIU_CR);
    MMIO.pBIU->dwBIUControl |= BIU_SR;
```

Interrupts are masked and pending interrupts are cleared (Figure 3-7 of databook).

Turning off CR (clear reset) and turning on set reset clears the reset.

```
*((PDWORD)(MMIO.pSpace + AO_CTL )) = 0x80000000;
*((PDWORD)(MMIO.pSpace + AO_FREQ )) = 0;
```

This resets audio out. PDWORD is a Windows type for a pointer to a double word (32 bits). See Figure 9-6.

```
/* audio in AI_CTL */
*((PDWORD)(MMIO.pSpace + AI_CTL )) = 0x80000000;
*((PDWORD)(MMIO.pSpace + AI_FREQ )) = 0;
```

This resets audio in. Generally speaking, the most significant bit in the control register for a peripheral is reset. See Figure 8-5.

```
*((PDWORD)(MMIO.pSpace + VI_CTL )) = 0x00080000;
*((PDWORD)(MMIO.pSpace + VI_CLOCK )) = 0;
*((PDWORD)(MMIO.pSpace + VO_CTL )) = 0x80000000;
*((PDWORD)(MMIO.pSpace + VO_CLOCK )) = 0;
```

This resets video in and video out. See Figures 6-11 and 7-26.

```
*((PDWORD)(MMIO.pSpace + SSI_CTL )) = 0xc0000000;
*((PDWORD)(MMIO.pSpace + SSI_CTL )) = (1 << 18);
```

The first instruction resets the Synchronous Serial Interface (SSI). The two upper most bits correspond to Transmitter reset and receiver reset. The second leaves the phone on hook after reset. See figure 16-1.

The following code resets the ICP. The loop is executed 10 times, just to make sure.

```
for ( Idx= 0 ; Idx < 10 ; Idx ++ ) {
    if ( *((PDWORD)(MMIO.pSpace + ICP_SR )) & 0x01 )
        break;
    ( *((PDWORD)(MMIO.pSpace + ICP_SR )) ) = 0x80;
}
```


The least significant bit (LSB) corresponds to ICP busy. If the ICP is busy executing microcode there is no reset. The assignment resets the ICP internal registers on reset. See Figure 13-17.

```
* ((PDWORD)(MMIO.pSpace + IIC_CTL )) = 0;
```

The IIC bus is disabled (figure 15-2 of the databook).

```
* ((PDWORD)(MMIO.pSpace + VLD_COMMAND )) = 0x00000401;
```

The writes a reset command with a count of 1 to the VLD. See Chapter 14.

```
* ((PDWORD)(MMIO.pSpace + TIMER1_TCTL)) &= ~0x1;
* ((PDWORD)(MMIO.pSpace + TIMER2_TCTL)) &= ~0x1;
* ((PDWORD)(MMIO.pSpace + TIMER3_TCTL)) &= ~0x1;
* ((PDWORD)(MMIO.pSpace + SYSTIMER_TCTL)) &= ~0x1;
```

The RUN bit is turned off in the four timer control registers.

```
* ((PDWORD)(MMIO.pSpace + BICTL)) = 0;
* ((PDWORD)(MMIO.pSpace + BDCTL)) = 0;
```

The instruction and data breakpoints are turned off See Figures 3-10 and 3-13. of the databook).

```
* ((PDWORD)(MMIO.pSpace + JTAG_DATA_IN))= 0x0;
* ((PDWORD)(MMIO.pSpace + JTAG_DATA_OUT))= 0x0;
* ((PDWORD)(MMIO.pSpace + JTAG_CTL))= 0x04;
```

The JTAG data registers and full bits are cleared. The JTAG interface is put in sleepless mode. See figure 17-3 of the databook.

Taking the processor out of reset

Once the program has been loaded, the following code from tmhal.c will begin initialization.

```
VOID halDSPstart ( PVOID pvObject )
{
    MMIO.pVIC->dwIMask = 0;
    MMIO.pVIC->dwIClear = 0xFFFFFFFF;
```

This disables interrupts and clear all pending interrupts See Figure 3-7.

```
MMIO.pBIU->dwBIUControl &= ~BIU_SR;
MMIO.pBIU->dwBIUControl |= BIU_CR;
```

Turning off SR (set reset) and turning on CR (clear reset) in the Bus Interface Unit (BIU) takes the processor out of reset See Figure 10.6.4.

The following code determines whether the TriMedia processor is running or not.

```

BOOL    halIsTMRunning (void) {
        return ( ( MMIO.pBIU->dwBIUControl & BIU_SR ) == 0 );
    }

```

tmmprun - multiprocessor download program

tmmprun is intended to run as an independent executable to be started from a PC command line. Its first argument is the name of a TM-1 executable to be downloaded, started, and which is passed all additional command line arguments. Between starting the executable and receiving its termination message, the module behaves as a server for the HostCall interface.

tmmprun is the driver for the host part of the PC version of the Level 2 Remote Procedure Call Server (RPCserv). An implementation of this is needed for each particular host of a TM-1 board which uses TCS's generic ANSI C library.

The **tmmprun** source consists of three C source files, `tmmprun.c`, `tmcrt.c`, and `unixlib.c`. This is a simplified version of the **tmmprun** source in the examples directory.

Makefile

The Makefile for `tmmprun.exe` uses the NMAKE utility from Microsoft. Source code for the makefile is shown below. The TCS variable needs to be set to point to the 1.1Y release. This Makefile was designed for Microsoft Visual C++ 4.0 TriMedia Developers Kit.

```

SDK      = c:\msdev
SDKBIN   = $(SDK)\bin
CL       = $(SDKBIN)\cl
LINK     = $(SDKBIN)\link

```

The Makefile assumes that Microsoft C++ is installed in C:\msdev (the default location).

```

CFLAGS = -W3 -Gs -Zi -Zp4 -c -Od -Ze -nologo

LIBS   = kernel32.lib $(TCS)\lib\Win95\tmman32.lib \
        $(TCS)\lib\Win95\host_comm.lib libcmt.lib \
        $(TCS)\lib\Win95\libload.lib

OBSJS  = tmmprun.obj tmcert.obj unixlib.obj

tmmprun.exe : $(OBSJS)
    $(LINK) @<<tmmprun.lnk
        -nodefaultlib
        -nologo
        -machine:i386
        -debug
        -debugtype:both
        -out:tmmprun.exe
        -map:tmmprun.map
        -pdb:none
        -subsystem:console
        -libpath:$(SDK)\lib
        $(OBSJS)
        $(LIBS)
<<

```

Header files

The file <tmlib/tmtypes.h> defines type naming conventions for TriMedia. The files <tmlib/HostCall.h> <tmlib/RPCServ.h>, <tmcert.h>, <tmif.h>, and <TM1IF.h> are used to implement access via the C run time library to files on the host. The file <Win95/tmman32.h> and <Win95/tmwincom.h> are for access to TMMAN functions. Compiling the tmmprun application requires header files from both Microsoft C and the 1.1Y release. The files <windows.h>, <stdio.h>, <io.h>, <time.h>, <sys\stat.h>, <windows.h>, and <fcntl.h> are from Microsoft C.

tmmprun main program

The main program for **tmmprun** is shown below.

```
int
main(int argc, char *argv[])
{
    STATUS      Status;
    COORD       ConsoleSize;
    DWORD       IdxNode;
    CRunTimeParameterBlock CRTParam;
    TMSTD_VERSION_INFO Version;
    int i, j;

    if (argc<3 || !_stricmp(argv[1], "-exec"))
        fatal ("bad usage\n");
```

The syntax is as follows:

```
tmmprun -exec first.out arg1 arg2 arg3 -exec second.out arg4 arg5
```

With this, **first.out** will be run on board 0 and **second.out** on board 1, with arguments "arg1 arg2 arg3".

```
Version.dwMajor = TMMAN_DEFAULT_VERSION_MAJ;
Version.dwMinor = TMMAN_DEFAULT_VERSION_MIN;
if ((Status = tmNegotiateVersion(TMMAN_DEFAULT, &Version))
    != TMOK)
    fatal ("**Error: tmNegotiateVersion failed
           (0x%x).\n", Status);
```

Before starting, the version of the TMMAN library (tmmman.dll) must be checked for compatibility. It needs to be present in the search path.

```
SetConsoleCtrlHandler(tmrunControlHandler, TRUE);
```

A control C handler is installed to free the resources allocated.

The following code counts the number of DSPs to download to. This corresponds to the number of "-execs" in the command line.

```
j = 0;

for ( i = 1 ; i < argc; i++ ) {
    if (!_stricmp(argv[i], "-exec" )) {
        if (tmDSPOpen(j, &DSPHandle[j]) != TMOK)
            fatal("tmDSPOpen failed, status %x\n", Status);
```

The `tmDSPOPEN` call returns a handle for accessing the *j*th processor. The function can be called more than once. All future accesses use the handle returned.

```
tmDSPGetCaps ( DSPHandle[ DSPCount ] , &DSPCaps );
```

The `tmDSPGetCaps` call returns the hardware capabilities of the board.

The following structure is defined in `<Win95/tmman32.h>`.

```
typedef struct TMMAN_DSP_CAPS
{
    CHAR                szPCIName[ TMSTD_NAME_LENGTH ];
    DWORD               dwHWVersion;
    DWORD               dwCPUVersion;
```

`szPCIName` is initialized with the ASCII name of the PCI device. `dwCPUVersion` is initialized with the revision ID register (see above). Possible values are as follows (hex).

	Fab	Crolles	MOS4	TSMC
TM1	00	40	80	
TM1.1	01	41	81	
TM1S	10	50	90	
TM1.1S	11	51	91	
TM1C	20	70	A0	
TM1.1C	21	71	A1	

```
DWORD               dwROMVersion;
TMSTD_MEMORY_BLOCK SDRAM;
TMSTD_MEMORY_BLOCK MMIO;
```

The SDRAM and MMIO fields define where the Trimedia memory and I/O registers are mapped. The `TMSTD_MEMORY_BLOCK` structure is defined in `<Win95/tmwincom.h>`. There are three values for each (TM-1000 address, Windows address, size).

```
TMSTD_MEMORY_BLOCK User;
DWORD               DSPNumber;
} TMMAN_DSP_CAPS, *PTMMAN_DSP_CAPS;
```

TMMAN allocates a 4K user page for host target communication. `DSPNumber` corresponds to the parameter passed to `tmDSPOPEN`.

```
MMIOPhysicalAddressArray[ DSPCount ] =
    DSPCaps.MMIO.dwPhysical;
```

The program to be downloaded needs to be patched with the MMIO addresses of all the other processors. `MMIO.dwPhysical` equals `MMIO_BASE` and PCI configuration register 5. `SDRAM.dwPhysical` equals `DRAM_BASE` and PCI configuration register 4.

SDRAM.dwLinear and MMIO.dwLinear correspond to the address in Windows virtual memory where these are mapped. SDRAM.dwsizes is just DRAM_LIMIT-DRAM_BASE and MMIO.dwSize is 2 megabytes.

```

        j ++;
    }
}

DSPCOUNT = j;
TMDwnLdr_create_shared_section_table(&SharedSections);

```

A shared section table needs to be allocated for downloading to work. The way shared variables are allocated is to the first processor in the command line order.

```

cruntimeInit();
IdxNode = 0;

```

The following loop loads a boot image in SDRAM for all processors.

```

for ( i = 2 ; i < argc; ) {

```

The CreateEvent parameters correspond to not running on workstation 4.0, auto reset event, initial state is not signalled.

```

EventArray[IdxNode] = CreateEvent(NULL, FALSE, FALSE,
                                  NULL);
assert (EventArray[IdxNode] != INVALID_HANDLE_VALUE) ;

```

Standard input, output, and error are treated specially.

```

CRTParam.OptionBitmap = 0;
CRTParam.StdHandle[0] =
    (DWORD)GetStdHandle(STD_INPUT_HANDLE);
CRTParam.StdHandle[1] =
    (DWORD)GetStdHandle(STD_OUTPUT_HANDLE);
CRTParam.StdHandle[2] =
    (DWORD)GetStdHandle(STD_ERROR_HANDLE);

```

The next argument should be the filename. The program is loaded into memory.

```

    for (j = i; j<argc && _stricmp (argv[j], "-exec"); j++);
    argv[j] = 0;
    Status= tmDSPExecutableLoadEx (
        DSPHandle[IdxNode],
        argv[i],
        DSPCount,
        SharedSections,
        MMIOPhysicalAddressArray );
    if (Status != TMOK)
        fatal ("**Error:
            Can't load %s. (0x%x) \n", argv[i], Status);

    CRTParam.OptionBitmap |=
        constCRunTimeFlagsUseSynchObject;
    CRTParam.SynchronizationObject =
        (DWORD) EventArray[IdxNode];
    CRTParam.VirtualNodeNumber = IdxNode;

```

A RPC server for the node is set up.

```

    if (!cruntimeCreate
        (IdxNode, j-i, &argv[i],
         &CRTParam, &CRTHandle[IdxNode] ))
        fatal("\r\nTMRUN: ERROR :
            Cannot Initialize C Run Time Server");

    IdxNode++;
    i = j + 1;

}

```

At this point code has been copied to the memory of the IREF boards and an RPC client has been created.

The following loop begins execution on all boards.

```

    for ( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ) {
        TMMAN_TMCONS_PARAMS TMConsControl;

        TMConsControl.fRedirectedStdin = FALSE;
        TMConsControl.fRedirectedStdout = FALSE;
        TMConsControl.fRedirectedStderr = FALSE;
        TMConsControl.fUseWindowSize = FALSE;
        TMConsControl.fIgnoreTMCons = TRUE;
        TMConsControl.fUseTMMonWindow = FALSE;
    }

```

The `tmDSPExecutableRun` routine just corresponds to a call to the `halDSPStart` routine explained above.

```

        Status = tmDSPExecutableRun
                (DSPHandle[IdxNode], TMMAN_DEFAULT,
                &TMConsControl);

        if (Status != TMOK)
            fatal("Cannot Start Target
                Executable (0x%x) \n", Status);
    }

```

The `waitForMultipleObjects` function waits for all the processors to terminate. `shutDown` frees the resources that have been allocated.

```

    WaitForMultipleObjects (DSPCount,
                            EventArray, TRUE, INFINITE );

    shutDown(0);
    return 0;
}

```

Using the downloader library

The `tmDSPExecutableLoadEx` routine loads the executable.

```

STATUS tmDSPExecutableLoadEx (
    DWORD DSPHandle,
    PCHAR pszImagePath,
    DWORD NumberOfDSPs,
    TMDwnLdr_SharedSectionTab_Handle SharedSections,
    PDWORD MMIOPhysicalAddressArray )
{
    PTMSTD_MEMORY_BLOCKpSDRAM;
    TMMAN_DSP_CAPS DSPCaps;
    TMMAN_DSP_INFO DSPInfo;
    TMDwnLdr_Status LoaderStatus;

    STATUS Status = TMOK;
    DWORD ImageSize;
    DWORD Alignment;
    DWORD AlignedDownloadAddress;
    TMDwnLdr_Object_HandleObjectHandle;
    Endian endian;
    CHAR szDeviceName[0x10];
}

```

The default clock speed is 100 Mhz. The default cache option is to leave caching to the downloader.

```

    DWORD ClockSpeed = 0x5f5e100; /* 100,000,000 */
    DWORD CacheOption = TMDwnLdr_LeaveCachingToDownloader;

```


The `tmDSPGetMiscInfo` function gets TMMAN software parameters. The variable points to the SDRAM address.

```
tmDSPGetCaps ( DSPHandle , &DSPCaps );
tmDSPGetMiscInfo ( DSPHandle , &DSPInfo );
pSDRAM = &DSPCaps.SDRAM;
```

The executable object is loaded into the host processors from the object file defined by the first parameter. This loads the object from the file. The first parameter is the filename. The object handle is initialized with a pointer to the shared section created in the main program. If the function is successful, the return value is zero.

```
LoaderStatus = TMDwnLdr_load_object_from_file
                ( pszImagePath, SharedSections,
                  &ObjectHandle )
if ( LoaderStatus )
    goto done;
```

The endianness is encoded in the object file (big or little endian). TriMedia supports both. The Windows host is always little endian.

```
LoaderStatus = TMDwnLdr_get_endian ( ObjectHandle, &endian);
if LoaderStatus )
    goto unload;
```

The Windows implementation requires that the program to be executed be little endian.

```
if ( endian != LittleEndian ) {
    TMDwnLdr_unload_object ( ObjectHandle );
    return TM_STATUS
                            ( TMMAN32_ERR_IMAGENOTLITTLEENDIAN );
}
```

Communication between the host and the target is done using shared variables. The function call patches the `TMMANSharedPatch` variable in the TriMedia executable to point to a communications area in host memory. This is used for system calls.

```
LoaderStatus = TMDwnLdr_resolve_symbol
                ( ObjectHandle,
                  "_TMMANSharedPatch",
                  DSPInfo.TMMANSharedPhys )
```

The software status is updated to indicate whether the symbol was found.

```
DSPInfo.Flags &= ~TMIF_DSPMISCINFO_SYMBOLNOTPATCHED;
if ( LoaderStatus )
    DSPInfo.Flags |=
        TMIF_DSPMISCINFO_SYMBOLNOTPATCHED;
tmDSPSetMiscInfo ( DSPHandle, &DSPInfo );
```

The image size and alignment are extracted from the executable.

```
LoaderStatus = TMDwnLdr_get_image_size
               ( ObjectHandle, &ImageSize, &Alignment );
if ( LoaderStatus )
    goto unload;
```

The base address of SDRAM is rounded up to the required alignment. The program will normally be read in at SDRAM base as reset starts there.

```
AlignedDownloadAddress =
    ( (DSPCaps.SDRAM.dwPhysical + Alignment - 1 ) &
      ~( Alignment - 1 ) );
```

The clock speed is read in from tmman.ini. DSPNumber corresponds to the number of the processor.

```
sprintf ( szDeviceName, "Device.%x", DSPCaps.DSPNumber );
ClockSpeed = GetPrivateProfileInt
              (szDeviceName, "ClockSpeed",
              ClockSpeed, "tmman.ini" );
```

The caching options are read in.

```
CacheOption = GetPrivateProfileInt
              (szDeviceName, "CacheOption",
              CacheOption, "tmman.ini" );
```

The following function relocates the executable and patches the downloader symbols. The downloader symbols begin with an underscore ('_') and end with (_init). The symbols are patched with the values below.

<code>_begin_stack_init</code>	base address of stack
<code>_begin_heap_init</code>	base address of heap
<code>_MMIO_base_init</code>	MMIO base for the current processor
<code>_MMIO_base<n>_init</code>	MMIO base for processor <n>
<code>_host_type_init</code>	host type (Windows 32 host, standalone, tmsim)
<code>_clock_freq_init</code>	clock frequency in Mhz
<code>_segment_list_init</code>	segment list
<code>_node_number_init</code>	number from 0 to total # of processors - 1
<code>_number_of_nodes_init</code>	total # of processors
<code>_do_section_lock</code>	locked sections are activated
<code>_locked_data_addr</code>	base address of locked sections (data)
<code>_locked_data_size</code>	size of locked section (data)
<code>_locked_text_addr</code>	base address of locked sections (text)
<code>_locked_text_size</code>	size of locked section (text)
<code>_cacheable_limit</code>	starting address of non cached memory
<code>_begin_mem</code>	this corresponds the beginning load address (DRAM BASE)
<code>_end_mem</code>	this corresponds the end of memory (DRAM limit)

```
LoaderStatus = TMDwnLdr_multiproc_relocate (
                                                ObjectHandle,
                                                tmWin32Host,
```

tmWin32Host corresponds to the `host_type` symbol above.

```
(Address*)MMIOPhysicalAddressArray,
```

The array of MMIO base addresses is used to patch the `MMIO_base` and `MMIO_base<n>` symbols (see above)

```
DSPCaps.DSPNumber,
NumberOfDSPs,
```

These correspond to `node_number` and `number_of_nodes` symbols above.

```
ClockSpeed,
(Address)AlignedDownloadAddress,
DSPCaps.SDRAM.dwSize,
```

These correspond to `clock_freq`, `begin_mem` and `end_mem` symbols above.

```
CacheOption );
```

The last parameter tells the processor how to relocate. The cache options are the same as the values defined in `<TMDwnLoader.h>`. This corresponds to the enumerated type `<TMDwnLdr_CacheingSupport>`. 0 means cachelocked regions and cacheable limit are entirely under control of the downloader, which will let the downloaded program run with 'cache off'. 1 means the cacheable limit and cachelocked regions are entirely under control of the user and the downloader/boot code won't touch it. 2 means cachelocked regions and cacheable limit are entirely under control of the downloader, which will use this control to intelligently map the different cached/uncached/cachelocked sections within the specified SDRAM, partitioned in different caching property regions, and let the downloaded program set cacheable limit and cachelocked.

```
if (LoaderStatus)
    goto unload;
```

The memory image for the processor is constructed in host memory. The `get_memory_image` function copies the data into the SDRAM of the target.

```
LoaderStatus = TMDwnLdr_get_memory_image
    (ObjectHandle,
    (Address)tmPhysToLin
    (AlignedDownloadAddress,
    &DSPCaps.SDRAM);
```

Finally, all resources allocated for the executable are freed. Extracted section group images and extracted symbol tables are unaffected. The error status is returned, if any.

Unload:

```
TMDwnLdr_unload_object ( ObjectHandle );
```

Done:

```
return TM_STATUS
    ( TMERR(TM_STATUS_HCOMP_TMLD, LoaderStatus) );
}
```

The Control C handler code is shown below. It signals the event so that the main program can terminate. Cleaning up:

```

BOOL WINAPI
tmrunControlHandler(DWORD dwCtrlType)
{
    int    IdxNode;

    for ( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ )
        SetEvent(EventArray[IdxNode]);
    return TRUE;
}

```

shutDown is called on termination or when an error occurs.

```

void shutDown(status)
{
    DWORD IdxNode;

    for ( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ) {

```

The following call corresponds to the halDSPStop routine explained previously. The loop is executed for the number of processors.

```

tmDSPExecutableStop(DSPHandle[IdxNode] );

```

The RPC server is shut down for the node.

```

    cruntimeDestroy(CRTHandle[IdxNode]);
    if (EventArray[IdxNode])
        CloseHandle(EventArray[IdxNode]);
    if (MMIOPhysicalAddressArray[IdxNode])
        tmDSPClose ( DSPHandle[IdxNode] );
}

```

The call to TM1IF_term shuts down RPC entirely.

```

TM1IF_term( );

```

The shared section table is freed if required.

```

    if (SharedSections)
        TMDwnLdr_unload_shared_section_table(SharedSections);

    SetConsoleCtrlHandler(tmrunControlHandler, FALSE);
    fprintf(stdout, "\nTMRUN:Press a key to close server >>");
    getchar();
    exit(status);
    return;
}

```

tmcrct.c

The code to initialize the C runtime library is shown below. The code declares a fixed number of entry points which will be called for the corresponding system call.

```

Bool   cruntimeInit ( void )
{
    BOOL ok;
    memset(GlobalContext, 0, sizeof(GlobalContext));
    if ( TM1IF_init(
        (RPCServ_OpenFunc) OpenFunc,
        (RPCServ_OpenDllFunc) OpenDLLFunc,
        (RPCServ_CloseFunc) CloseFunc,
        (RPCServ_ReadFunc) ReadFunc,
        (RPCServ_WriteFunc) WriteFunc,
        (RPCServ_SeekFunc) LseekFunc,
        (RPCServ_IsattyFunc) IsattyFunc,
        (RPCServ_FstatFunc) FstatFunc,
        (RPCServ_FcntlFunc) FcntlFunc,
        (RPCServ_StatFunc) StatFunc,
        (RPCServ_ExitFunc) ExitCodeFunc,
        True /* no big endian support yet */) != True )
        return False;
}

```

The following instruction starts the server.

```

    ok = TM1IF_start_serving() != TM1IF_Serving_Failed
    if (!ok)
        TM1IF_term( );
    return ok;
}

```

The following routine creates a C runtime instance.

```

UInt32 cruntimeCreate( unsigned nproc,
                       unsigned argc,
                       char **argv,
                       CRunTimeParameterBlock* Parameters,
                       unsigned int *CRTHandlePointer )
{
    DWORD           Written;
    UInt32          PathIdx;
    static TMCRT_CONTEXT CRT;
    PTMCRT_CONTEXT pCRT = &crt;
    DWORD           ServerStatus;
    TMMAN_DSP_CAPS  Caps;
    CHAR            Key[0x10];
    CHAR            Path[TMSTD_PATH_LENGTH];
    TMCRT_STD_HANDLE StdHandle[3];
}

```

Standard input, output and error are common to all processors. The command lines are processor specific.

```
GlobalContext[Parameters->VirtualNodeNumber] = pCRT;
pCRT->OptionBitmap = Parameters->OptionBitmap;
pCRT->DSPNumber = nproc;
pCRT->StdInHandle = (HANDLE)Parameters->StdInHandle;
pCRT->StdOutHandle = (HANDLE)Parameters->StdOutHandle;
pCRT->StdErrHandle = (HANDLE)Parameters->StdErrHandle;
pCRT->VirtualNodeNumber = Parameters->VirtualNodeNumber;
pCRT->ArgumentCount = argc;
pCRT->ArgumentVector = argv;
```

A synchronization object is used to await for all processors to terminate.

```
pCRT->SynchObject =
    (HANDLE)Parameters->SynchronizationObject;
pCRT->fServerLoaded = FALSE;
pCRT->fTargetExited = FALSE;
pCRT->ExitCode = ~0;
```

tmDSPOpen can be called more than once. Here it is called to get the handle. Calling tmDSPOpen increments a reference count which tmDSPClose decrements.

```
tmDSPOpen ( pCRT->DSPNumber , &pCRT->DSPHandle );
tmDSPGetCaps ( pCRT->DSPHandle, &Caps );
tmDSPClose ( pCRT->DSPHandle );
```

The following code creates a console Window, if required.

```
if ( pCRT->OptionBitmap & constCRunTimeFlagsAllocConsole ) {
    AllocConsole();

    SetConsoleTitle("TriMedia Console");
    if ( pCRT->OptionBitmap &
        constCRunTimeFlagsUseWindowSize ) {
        COORD Coord;
        Coord.Y = (WORD)Parameters->WindowSize;
        Coord.X = 80;
        SetConsoleScreenBufferSize (
            GetStdHandle(STD_OUTPUT_HANDLE), Coord );
    }
}
```

Standard input, output and error are redirected to the console window.

```
pCRT->StdHandle[0] = GetStdHandle(STD_INPUT_HANDLE);
pCRT->StdHandle[1] = GetStdHandle(STD_OUTPUT_HANDLE);
pCRT->StdHandle[2] = GetStdHandle(STD_ERROR_HANDLE);

}
```

An event is created to signal exit.

```

    if ( !(pCRT->ExitObject =
          CreateEvent ( NULL, TRUE, FALSE, NULL)) ) {
        if (pCRT->OptionBitmap & constCRuntimeFlagsAllocConsole)
            FreeConsole();
        return FALSE;
    }

```

The handle for a file descriptor is an integer. Files that are named explicitly via `open` are identified using a small integer. Standard input, output, and error are special because each processor can have its own console window and because of redirection.

For these, the handle is a bit field structure encoding the file descriptor (3 bits), the DSP number (5 bits), and a 24 bit magic value. The following loop initializes the values.

```

    for (i = 0; i<=2; i++) {
        StdHandle[i].Magic      = 0x005a5a5a;
        StdHandle[i].StdType   = i;
        StdHandle[i].DSPNumber = pCRT->VirtualNodeNumber;
    }

```

The following call initializes the RPC server. The return value should be non zero.

```

    assert(TM1IF_add_node_info(nproc, argc, argv,
        &StdHandle[0], &StdHandle[1], &StdHandle[2],
        Caps.SDRAM.dwLinear - Caps.SDRAM.dwPhysical,
        Caps.SDRAM.dwPhysical,
        Caps.SDRAM.dwPhysical + Caps.SDRAM.dwSize,
        (void*)pCRT->DSPNumber ) != 0);

```

The RPC server is initialized with the standard I/O file descriptors.

`TM1IF_add_node_info` creates TMMAN message queue for communication to the host.

The parameters are copied into a table entry for the node. The Hostcall protocol is used. At this point the RPC server is initialized.

During execution, DLLs can be loaded using the downloader also. The DLL search path is initialized from `tmman.ini`.

```

pCRT->fServerLoaded = TRUE;
for ( PathIdx = 0; PathIdx < MAX_PATH_INDEX ; PathIdx++ ) {
    sprintf ( Key, "%u", PathIdx );
    GetPrivateProfileString (
        "DLLPath", Key, "DEFAULT", Path,
        TMSTD_PATH_LENGTH, "tmman.ini" );
    if (!_stricmp ( Path, "DEFAULT" ))
        break;
    OpenDll_add_dll_path ( Path );
}
*CRTHandlePointer = (UInt32)pCRT;
return True;
}

```

Shutting down the RPC server

```

UInt32 cruntimeDestroy (
    UInt32 CRTHandle )
{
    PTMCRT_CONTEXT pCRT= (PTMCRT_CONTEXT)CRTHandle;
    UInt32 ExitCode;
}

```

RPCServ should have created a thread by now so we block here on the global event to be signalled.

```

if ExitProcess = TRUE;

SetEvent ( pCRT->ExitObject );
CloseHandle ( pCRT->ExitObject );
ExitCode = pCRT->ExitCode;

```

The console window is closed, if necessary.

```

if ( pCRT->OptionBitmap & constCRuntimeFlagsAllocConsole )
    FreeConsole();

```

The RPC server is shut down for this node.

```

TMlIF_remove_node_info( pCRT->VirtualNodeNumber );
GlobalContext[pCRT->VirtualNodeNumber] = NULL;
return ExitCode;
}

```

Implementation of POSIX system functions

POSIX I/O calls are implemented in unixlib.c.

Code for the write system call is show below.

```
DWORD WriteFunc ( DWORD Handle, PVOID pBuffer, DWORD Count )
{
    PTMCRT_STD_HANDLE EncodedHandle = (PTMCRT_STD_HANDLE)&Handle;
    DWORD BytesWritten;
```

Handles for standard input, standard output and standard error, are encoded using the TMCRT_STDE_HANDLE type. The 24 lower bits encode a special value.

If the descriptor is stdin, stdout, or stderr, the Windows WriteFile call is used. The first statement extracts the C runtime context for the processor.

```
    if ( EncodedHandle->Magic == 0x005a5a5a ) {
        PTMCRT_CONTEXT pCRT = (PTMCRT_CONTEXT)
            GlobalContext[EncodedHandle->DSPNumber];
        handle = pCRT->StdHandle[EncodedHandle->StdType]
        if ( !WriteFile (handle, pBuffer,
                        Count, &BytesWritten, NULL ) )
            return 0;
        else
            return BytesWritten;
    }
```

Otherwise the POSIX `_write` call is used.

```
    return _write ( Handle, pBuffer, Count );
}
```

Code for the read system call is shown below.

```
DWORD ReadFunc ( DWORD Handle, PVOID pBuffer, DWORD Count ) {
    PTMCRT_STD_HANDLE EncodedHandle = (PTMCRT_STD_HANDLE)&Handle;
    DWORD BytesRead;
    int handle;
    HANDLE Objects[2];
    DWORD ObjectSignalled;
```

If the process has terminated via exit, we return immediately.

```
    if ( fExitProcess == TRUE )
        return 0;
```

If the descriptor is standard in, standard out, or standard error Microsoft event based I/O is used.

```
if ( EncodedHandle->Magic == 0x005a5a5a ) {
    PTMCRT_CONTEXT pCRT = (PTMCRT_CONTEXT)
        GlobalContext[EncodedHandle->DSPNumber];
    BOOL Status;
```

We wait for the read to complete or the process to terminate via exit.

```
handle = pCRT->StdHandle[EncodedHandle->StdType]
Objects[0] = handle;
Objects[1] = pCRT->ExitObject;
ObjectSignalled = WaitForMultipleObjects ( 2, Objects,
        FALSE, INFINITE );
```

If the process terminated via exit return, zero is returned.

```
if ( (ObjectSignalled - WAIT_OBJECT_0 ) == 1 ) {
    return 0;
```

At this point the data is available. The Microsoft ReadFile API is used to read it.

```
Status = ReadFile ( pCRT->StdInHandle,
        pBuffer, Count, &BytesRead, NULL );
```

TriMedia and UNIX use LF (linefeed) for end of line and Windows uses CR/LF. The StripCR routine removes unnecessary carriage returns from text files (not shown).

```
if ( GetFileType (pCRT->StdInHandle) == FILE_TYPE_CHAR )
    StripCR ( pBuffer, &BytesRead );
return Status == TRUE ? BytesRead : 0;
}
```

Otherwise, a POSIX API is used.

```
return _read ( Handle, pBuffer, Count );
}
```

Code for the exit system call is shown below.

```
DWORD ExitCodeFunc ( DWORD DSPNumber, DWORD ExitCode ) {
    PTMCRT_CONTEXT pCRT = (PTMCRT_CONTEXT)GlobalContext[ DSPNumber ];
    DWORD BytesWritten;
    CHAR szTemp[0x80];
```

SynchObject is used to dedicate that all the processors have terminated. ExitObject is used to shut down this instance of the C runtime server.

```
if ( pCRT->OptionBitmap & constCRunTimeFlagsUseSynchObject)
    SetEvent ( pCRT->SynchObject );
pCRT->fTargetExited = TRUE;
pCRT->ExitCode = ExitCode;
SetEvent ( pCRT->ExitObject );
return 0;
}
```

Code for the other calls is straightforward.

Philips TriMedia SDE Cookbook

Part 4:

Optimizing TriMedia Applications



Table of Contents

Chapter 1 Porting and Optimizing Programs

Introduction.....	1-2
Porting Considerations.....	1-2
Library and System-Calls Support	1-2
Floating-Point Computations	1-3
File I/O.....	1-3
Performance Tuning	1-4
Profile-Driven Compilation.....	1-6
Grafting Based on Profile Information	1-7
Graft-Tuning Parameters	1-11
Loop Optimization	1-12
Remove If Statements and Conditional Expressions.....	1-13
Collapse Mutually Exclusive if Statements.....	1-17
Use MUX and FMUX Pseudo Operations	1-18
Parallel Reduction Loops.....	1-19
Use MUX on Variable Length Loops.....	1-20
Apply Strength Reduction	1-22
Move Externals and Reference Parameters to Locals.....	1-26
Remove Function Calls.....	1-28
Pay Attention to Compile Time.....	1-30
Use #pragma TCS_break_dtree	1-32
Use Goto for Loops with a Trailing if Statement.....	1-34
Loop Fusion	1-35
Replace by 	1-36
Replace && by & or IZERO.....	1-36
Using Software Pipelining.....	1-37
Use Trimedia Style Booleans in Critical Parts of the Code	1-38

Loop Unrolling	1-38
Loop Unrolling Versus Grafting.....	1-40
Using Restricted Pointers	1-42
Using Custom Operators.....	1-46
Using the Global Optimizer.....	1-49
Using Profiling and Grafting.....	1-57
Using Unsafe Alias Analysis.....	1-59
Using a Dirty Float.....	1-63
Using Cache Optimization.....	1-64
Vary the Right-Most Array Index in the Inner Loop.....	1-64
Pack Data as Tightly as Possible.....	1-66
Trade CPU Cycles for Cache Cycles	1-67
Watch for Cache Set Hotspots.....	1-69
Blocking	1-70
Two-Level Blocking	1-72
Watch for Data Cache Bank Conflicts.....	1-73
Try -noloadspec When Thrashing	1-74
Summary	1-76

Chapter 2 System Programming Support

Programming Support	2-2
Interrupt Service Routines and Exception Handlers	2-2
User View	2-2
Saving/Restoring Behavior.....	2-5
Declaring Interrupt Service Routines	2-5
Usage Notes.....	2-6
Interrupt-Latency Support	2-7
Supporting the Machine Level Simulator: tmsim -il	2-7
Breaking Decision Trees: #pragma TCS_break_dtree.....	2-8
Supporting Cache Control.....	2-9
Using MMIO Locations.....	2-11

Chapter 3	Case Studies	
	Introduction.....	3-2
	Special-Purpose Block Filter.....	3-2
	Fixed-Point Arithmetic	3-4
	IFIR16 Custom Operations	3-5
	Dual-Phase Loop	3-6
	Critical Path.....	3-8
	Algebraic Transformation	3-9
	Balancing the Critical Path.....	3-10
	More Unrolling	3-11
	Matrix Transpose	3-12
	Divide and Conquer	3-14
	Using Custom Operations	3-15
	Inlining and Shrink-Wrapping	3-16
	Cache Alignment	3-19
Chapter 4	Performance Analysis on the Hardware	
	Overview	4-2
	Terminology	4-3
	Reasons for Long Interrupt Latencies.....	4-5
	Clearing the IEN.....	4-6
	Changing the Global Interrupt Priority.....	4-7
	Individual Disabling	4-7
	Preventing Task Preemption.....	4-7

Interrupt Latency Sampling	4-8
Using the Sampler	4-9
Detection of Latency Violators	4-9
Latency Sampler Code	4-10

Chapter A

Shell Scripts

tmprof.select	A-1
select	A-1

Chapter 1

Porting and Optimizing Programs

Topic	Page
Introduction	1-2
Porting Considerations	1-2
Performance Tuning	1-4
Summary	1-76

Introduction

This chapter provides guidelines for porting and optimizing performance tuning. It describes various optimization methods supported by the TriMedia Compilation System as well as techniques for exploiting the fine-grain parallelism of the TriMedia architecture.

Porting Considerations

You should use ANSI Standard C when developing applications for TriMedia processors. The implementation of the TriMedia C compiler is based on the following standards:

- *American National Standard for Programming Languages - C*, ANS X3.159-1989 and ISO/IEC 9899:1990
- *Amendment 1 (1994) to ISO/IEC 9899:1990*
- *Technical Corrigendum 1 (1994) to ISO/IEC 9899:1990*

Additionally, the compiler supports the concept of *restricted pointers*, as proposed by the *Numerical C Extensions Group* in X3J11/95-049, WG 14/N448

This document is available from <ftp://ftp.dmk.com.DMK/sc22wg14/c9x/aliasing>. Chapter 1 of “Programming Languages and File Formats” discusses compatibility issues, C language extensions, and implementation-dependent features.

Library and System-Calls Support

The language implementation supports the standard C library, as defined in the ANSI/ISO C Standard. No other libraries are supported. For example, programs using X11 libraries or Sun-specific libraries do not compile with the TriMedia Compilation System.

The following library and system calls are implemented as traps by simulator **tmsim**; that is, **tmsim** uses the corresponding library and system call routine on the host processor to simulate the routine.

```
close, _fstat, _isatty, _link, _lseek, _mktemp,
open, _read, _unlink, _write, getenv, time
```

The system call names all begin with “_” because of ANSI C Standard name space requirements. Because many traditional C programs use system call names without a leading “_” (for example, `read()` rather than `_read()`), the C library includes stubs that perform the desired renaming (for example, defining `read()`, which simply executes `_read()`). You should always include the appropriate header file (`<fcntl.h>` for `open()`, `<sys/stat.h>` for `fstat()`, and `<unistd.h>` for the remaining system calls) when compiling a program that uses system calls directly.

Floating-Point Computations

All floating-point data types (`float`, `double`, and `long double`) are single precision and have the same range of values in the current compiler and TM-1000 processor. Therefore, you should use `float` instead of `double` or `long double`.

You should be aware that the results of floating-point computations performed on a Sparc or other workstation can differ from the results of computations performed on a TriMedia processor or simulator. Many compilers automatically convert `float` to `double` during expression evaluations and function calls, especially when the compiler cannot find the function prototype.

File I/O

Applications should employ batch processing and use only file-based input. Output can be sent to the standard output stream, to the standard error stream, or to files.

Interactive programs are not supported currently. To avoid distorting profile information, avoid gathering and printing unnecessary output.

Performance Tuning

Use the following techniques and tools to improve program execution times:

- Profile-driven compilation
- Decision-tree grafting
- Loop optimization
- Loop unrolling
- Restricted pointers
- Custom operators
- Fine tuning of grafting
- Global optimizer
- Unsafe alias analysis
- Dirty float option
- Cache optimization
- Cache instructions

The simple function in Figure 1-1 computes the convolution of integer arrays *a* and *b* of lengths 400 and 8, respectively. Although faster algorithms for computing the convolution exist, the code demonstrates the utility of profiling, grafting, loop unrolling, and custom operators. A number of different transformations of this convolution function using the listed techniques for improving the performance are presented throughout this chapter. The full program, including the different versions of the convolution function, is included in the software release directory *examples*.

We start by making optimal use of the processor's computing resources. In particular, we increase the level of parallelism by enlarging the number of operations in decision trees and by removing irrelevant dependencies between these operations. The required techniques are grafting, loop unrolling, and improving the compiler's alias analysis with restricted pointers.

When you use these techniques, you might reach the stage at which the processor is saturated. The processor's computing resources—the number and configuration of the available functional units—limit application performance.

This performance limit applies to the application only as it is formulated by you and compiled by the compiler. To further improve performance, you must either find another implementation (change the formulation of the algorithm) or invoke the global optimizer (change the way the application is compiled).

```

/*                                fir1.c -- (part)
*
* convolution of two 8 bit integer arrays a and b, of length 400 and 8, respectively.
* Rough pictorial description of the process.
*
*          |-----|-----|-----|-----| a[400]
*
*          |-----|                                     b[8]
*
*          |-----|-----|-----|-----| a
*
* |-----|                                     time reversed b
*
*          |-----|-----|-----|-----| a
*
*          |-----|                                     time reversed
*                                     and sliding b
*
* Increase the length of array a so that vector of length 8 could be
* prepended and appended. With this additional zeros, separate handling
* of beginning and end of data is avoided.
*
* |00000000|-----|-----|-----|-----|00000000|
*
* |<-- Original length 400 array a --->|
*
* These arrays hold the result of convolutions. Actual required
* output array length = 400 + 8 - 1 = 407, but our modified algorithm
* calculates one unnecessary element. To handle this, output
* array length has been increased by 1
*/
#define NROF_SAMPLES400

void
direct_convolution( char *a, char *b, int *c )
{
    int      k, j;

    for(k = 0; k < NROF_SAMPLES; k++) {

        c[k] = 0;

        for(j = 0; j < 8; j++)
            c[k] += b[j] * a[k - j];/* a is shifted 8 in the call*/
    }
}

```

Figure 1-1 Convolution Example (Part of Example fir1.c)

Profile-Driven Compilation

The TriMedia Compilation and Simulation system facilitates the *compile-profile-recompile* cycle of performance tuning. First, you compile the program using the compiler driver **tmcc** with the `-p` option (write profiling information to file `dtprof.out`). Next, you simulate program execution using the machine-level simulator **tmsim**. Then, you recompile the program with **tmcc**, using either the `-r` option (profile-driven compilation) or the `-G` option (profile-driven compilation with grafting).

The first optimization step is to obtain profile information about the program and identify the critical sections. After the critical sections are identified, you can perform grafting and loop unrolling and can use restricted pointers to remove spurious dependencies. (Function inlining is not currently supported.)

The following procedure illustrates both how to perform profiling, and how to use **tmprof** to summarize execution statistics:

1. Compile the source modules using **tmcc** with the `-p` option. Do not use the `-G` and `-r` options at this stage. (With the `-p` option, the compiler instruments the user program with code to determine decision-tree execution counts and branch probabilities. Use **tmsim** to simulate the instrumented program, which generates the profile information in file `dtprof.out`. Use the option `-nomm` to switch off the simulation of the memory model and save execution time.)
2. Recompile the source modules with the `-r` option and without the `-p` option. This causes the generated decision trees to be free of profiling code. Assuming the input was representative, recompilation based on profiling adds branch probabilities in the new decision-trees file. It is important not to change the source code because the profiling information is based on the control-flowgraph of the program. When this changes during the generate-profile and read-profile compilations, the two do not match because the profile is ignored.)
3. Run **tmsim** with the `-statfile` option to save the execution statistics and since `-nomm` is not used, memory mode is simulated.
4. Run **tmprof** with the `-func` option to generate a report for each function in the program. The `-scale 1` option tells **tmprof** to report the cycle count without scaling.

The following commands generate a summary report for the program `fir1.c`:

```
tmcc -p fir1.c -o fir1          /* Generate program with profiling turned on.*/
tmsim -nomm fir1              /* Simulate intermediate code and produce
                             dtprof.out. */

tmcc -r fir1.c -o fir1        /* Recompile using profile information.*/
tmsim -statfile fir1.stat fir1 /* Simulate and collect accurate cycle
                             information.*/

tmprof -scale 1 -func fir1.stat /* Output is sent to stdout.*/
```


The report produced by this sequence of commands is as follows. Note that the values printed differ depending on the version on the TCS:

Funcname	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
direct_convolution	1	34170 89.34%	85	875
initialize	1	2867 7.50%	261	84
exit	1	193 0.50%	145	22
main	1	153 0.40%	136	3
_clear_all_regs_1	1	129 0.34%	116	0
_clear_all_regs	1	129 0.34%	116	0
_pre_start	1	120 0.31%	65	45
_profile_write	1	114 0.30%	92	11
_start_1	1	89 0.23%	67	15
_start_second	1	64 0.17%	58	0
_return_custom_begin	1	62 0.16%	58	0
_default_exit_2	1	43 0.11%	0	38
_default_exit_1	1	36 0.09%	29	3
_exit	1	33 0.09%	29	0
_custom_begin	1	33 0.09%	29	0
_default_exit	1	6 0.02%	0	0
_return_custom_end	1	4 0.01%	0	0
_custom_end	1	4 0.01%	0	0
total/average		38249 100.00%	1286	1096

The report shows all functions that executed, including the startup and library functions, the total number of cycles executed for each function, and the stall cycle contribution of both the instruction-cache and data-cache.

Because the function `direct_convolution` in the source module `fir1.c` takes about 90% of the total cycles, it is the one to be optimized. The next sections show how to get a further performance gain.

Grafting Based on Profile Information

Grafting increases parallelism within decision trees. As a result, the program size increases. This technique replaces any jump with a copy of the destination tree and thus “grows” larger decision trees.

The core compiler **tmccom** generates an intermediate representation of a program known as a decision-tree representation¹. Decision trees are derived from basic blocks. A basic block is a sequence of instructions with no jumps into it, except to the first instruction and

1. You can generate an example decision tree by compiling a program using **tmcc** with the `-t` option. `tmcc -t foo.c` produces a file `foo.t` with machine-like operations, see Chapter 2 of “Programing Languages and File Formats.”

no jumps out except at the last instruction. Basic blocks are connected to one another by conditional or unconditional jumps. It is well known that not much parallelism within basic blocks exists to be exploited. Furthermore, frequent branching behavior would result in underutilization of processor resources.

A decision tree is similar to a basic block in that the decision tree can be entered only at the beginning. However, a decision tree can have multiple exits. (Chapter 2 of “Programming Languages and File Formats” defines the syntax and semantics of decision trees.) Decision trees are larger than basic blocks and potentially have more fine-grain parallelism that can be exploited during optimization

Figure 1-2 shows a decision tree ending in a branch. The actual operations in the tree are not important for this example. The decision tree `__ip_DT_1` has two exits, one leading back to itself (`gotree {__ip_DT_1}`) and the other leading to another decision tree (`gotree {__ip_DT_2}`).

```
{__ip_DT_1:}
tree (50)
    2 rdreg (12);
    1 ld32 2;
    4 rdreg (11);
    6 rdreg (10);
    7 ld32x 6 4;
    9 rdreg (9);
    10 ld32x 9 4;
    11 imul 7 10;
    12 iaddi(1) 11
    13 st32 2 12
        after 10 7 1;
    14 iaddi (1) 4;
    15 wrreg (11) 14
        after 4;
    16 ilesi (50) 14;
    if 16 (0.980000) then
        gotree {__ip_DT_1}
    else (16)
        gotree {__ip_DT_2}
    end (16)
endtree (*__ip_DT_1*)
```

Figure 1-2 Example of a Decision Tree Ending in a Branch

Notice the back edge from `__ip_DT_1` to itself has a probability of 0.98. This statistic is derived from a profiling run. The compiler can do a better job of grafting if it has information about decision-tree execution counts and branch probabilities. In this case, the decision tree `__ip_DT_1` has an execution count of 50 (the first number after the label). If

grafting is enabled, the compiler replaces the instruction “`gotree {__ip_DT_1}`” with a copy of the tree `__ip_DT_1`, doubling the size of the decision tree `__ip_DT_1`.

Figure 1-3 shows a schematic of the same tree after it is grafted. The scheduler can decide where to place code and when to use guarded execution if it has information about branch probabilities. You can also guide the compiler in its grafting decisions, discussed later in the section.

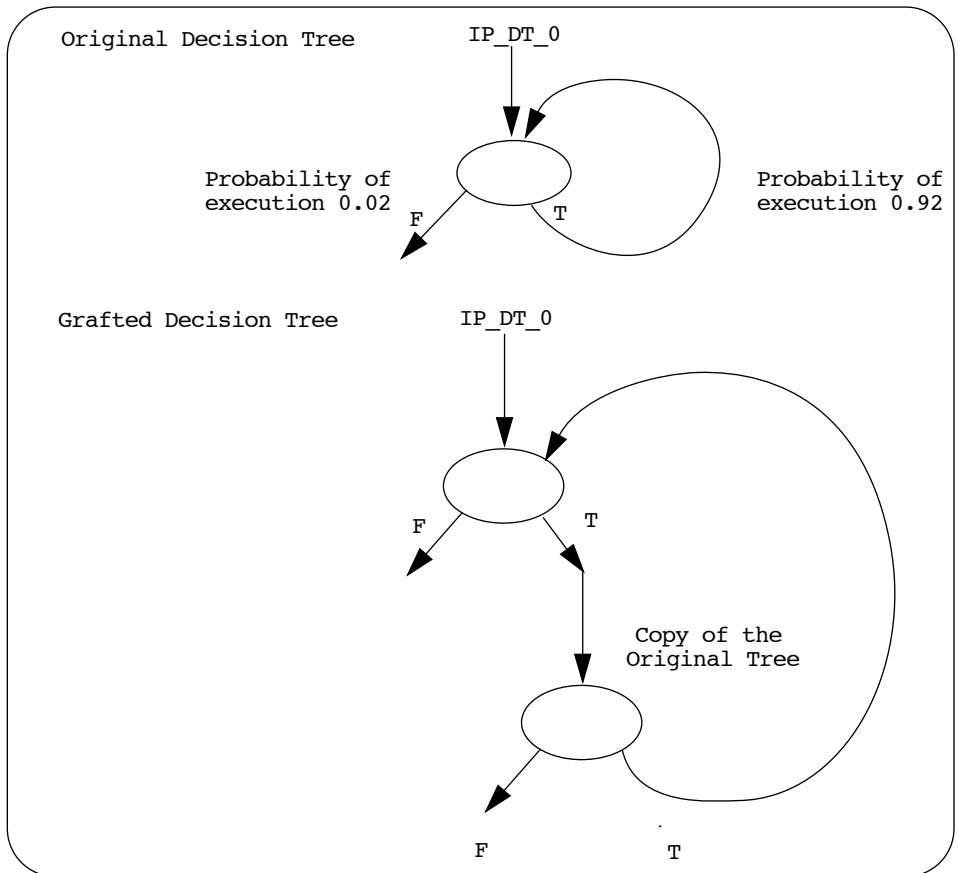


Figure 1-3 Decision Tree After it is Grafted

It is important to note that grafting is a code-replication technique that eliminates branches but increases the code size. It is a technique similar to loop unrolling, but does not reduce the overhead of the loop as manual loop unrolling can. This is shown later on.

You can improve performance of program `fir1.c` by grafting after profiling. The following procedure performs the compile-profile-recompile cycle with grafting enabled after profiling.

```

tmcc -p fir1.c -o fir1          /* Generate program with profiling turned on.*/
tmsim -nomm fir1               /* Simulate intermediate code and produce
                               dtprof.out.*/

tmcc -G fir1.c -o fir1        /* Recompile using profile information and perform
                               grafting.*/

tmsim -statfile fir1.stat fir1 /* Simulate and collect cycle accurate
                               information.*/

tmprof -scale 1 -func fir1.stat /* Output is sent to stdout.*/

```

The report produced is as follows:

Funcname	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
direct_convolution	1	29559 91.95%	302	847
initialize	1	1376 4.28%	493	84
exit	1	193 0.60%	145	22
main	1	153 0.48%	136	3
_clear_all_regs_1	1	129 0.40%	116	0
_clear_all_regs	1	129 0.40%	116	0
_pre_start	1	120 0.37%	65	45
_profile_write	1	114 0.35%	92	11
_start_1	1	89 0.28%	67	15
_start_second	1	64 0.20%	58	0
_return_custom_begin	1	62 0.19%	58	0
_default_exit_2	1	43 0.13%	0	38
_default_exit_1	1	36 0.11%	29	3
_exit	1	33 0.10%	29	0
_custom_begin	1	33 0.10%	29	0
_default_exit	1	6 0.02%	0	0
_return_custom_end	1	4 0.01%	0	0
_custom_end	1	4 0.01%	0	0
total/average		32147 100.00%	1735	1068

We discover that the execution time of the grafted version of `fir1` improved by 18% over the ungrafted version. However, the number of stall cycles in the instruction cache increased, which is due to the increase in code size. We discover this when we list the size information of `fir1.o` with `tmsize` the text size increased from 5952 bytes to 6912 bytes. It appears that grafting is a valuable tool for easy performance increments, but that a trade-off has to be made between performance gain and code size increase. “Graft-Tuning Parameters” on page 1-11 describes how grafting can be customized with a grafting parameter file.

Graft-Tuning Parameters

You can guide the grafting decision of the compiler through a grafting parameters file. The graft-tuning file allows specifying a number of conditions on decision tree grafting on a function by function basis. You can specify graft parameters for some functions and use a default applicable to all functions that were not explicitly listed in the graft parameters file. The parameters that govern grafting decisions are as follows. See the man page for **tmccom** for the current default values for these parameters.

- *Graft Enable*—This is a boolean flag enabling or disabling grafting for a particular function.
- *Maximum Code Replication Factor*—This limits the factor by which the code size for a function can be expanded due to grafting. Note that although grafting might initially increase code size, many optimizations are performed after grafting and these reduce code size.
- *Maximum Graft Depth*—This limits how many times grafting is performed along a particular execution path in the current decision tree. This restricts how much grafting is allowed on a tree.
- *Minimum Probability Threshold*—This specifies the minimum probability of execution of a branch to allow grafting for that branch.
- *Minimum Execution Count Threshold*—A decision tree is not a candidate for grafting if its execution count is below this threshold.

A graft-tuning file can contain different parameters for different functions and at most one default set of parameters for all functions that are not listed explicitly in the file. The default values for the graft tuning parameters are as follows:

#function name	enabled	codesize	depth	probability- threshold	execution-count-threshold
<default>	1	20.0	20	0.4	10.0

The **tmprof** output shows that the stall cycles from the cache misses increased during the optimization stages. First loop unrolling was done, followed by grafting. These are similar techniques, and possibly the code expansion of the two was too much for the default grafting parameters. When limiting the grafting for the already unrolled function, a better cache characteristic can be obtained. The `graftfile` with the following parameters is used on the example:

#function	enabled	codesize	depth	prob. threshold	exec. count threshold
<default>	1	4.0	2	0.4	10.0

This `graftfile` is included in the following sequence of commands to compile `fir4.c`.

```

tmcc -p fir4.c -o fir4
tmsim -nomm fir4
tmcc -G -tmccom -graft_tuning_file graftfile -- fir4.c -o fir4
/* Recompile using profile information,
perform grafting*/
tmsim -statfile fir4.stat fir4 /* Simulate & collect cycle accurate
information.*/
tmprof -scale 1 -func fir4.stat /* Output is sent to stdout.*/

```

The **tmprof** report is as follows:

Funcname	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
initialize	1	1423 36.95%	290	79
custom_ops_direct_convolu	1	1178 30.59%	331	107
exit	1	193 5.01%	145	22
main	1	153 3.97%	136	3
_clear_all_regs_1	1	129 3.35%	116	0
_clear_all_regs	1	129 3.35%	116	0
_pre_start	1	120 3.12%	65	45
_profile_write	1	114 2.96%	92	11
_start_1	1	89 2.31%	67	15
_start_second	1	64 1.66%	58	0
_return_custom_begin	1	62 1.61%	58	0
_default_exit_2	1	43 1.12%	0	38
_start	1	38 0.99%	29	5
_default_exit_1	1	36 0.93%	29	3
_custom_begin	1	33 0.86%	29	0
_exit	1	33 0.86%	29	0
_default_exit	1	6 0.16%	0	0
_return_custom_end	1	4 0.10%	0	0
_custom_end	1	4 0.10%	0	0
total/average		3851 100.00%	1590	328

The performance improvement by tuning the graft file is mainly due to the reduction in the instruction cache stall cycles.

Loop Optimization

You perform loop optimization by moving critical code off the control flow path so that the inner loops of the program can be reduced to a single decision tree. This section describes several techniques, such as loop nesting, using `gotos`, and `dtree` breaks, to achieve loop optimization. Most of the techniques described here are automatically

performed by the compiler. We provided the information to give you control over the number of decision trees when required.

Remove If Statements and Conditional Expressions

Figure 1-4 shows a C program that finds the maximum element of a 100-element vector. 1494 instruction cycles are necessary for the call to `vecmax` with global optimization.

```

1           2           3
12345678901234567890123456789012345

1 float vecmax(float *a, int size) {
2     float max = a[0];
3     int i;
4     for (i=1; i<size; i++) {
5         if (a[i] > max)
6             max = a[i];
7     }
8     return max;
9 }
10
11 main() {
12     float a[100];
13     int i;
14     for (i=0; i<100; i++)
15         a[i] = rand();
16     (void)vecmax(a, 100);

```

Figure 1-4 The `vexmax.c` Program

You can determine the decision-tree flow in `vecmax` using `tmdtprof`.

```

$ tmcc -t -p vecmax.c
$ tmtsim vecmax.t
$ tmdtprof dtprof.out
  (... )
  dt(0)1/1 ops(11) exits(2)
    0 -> dt(1) exec count(1)
    1 -> dt(1) exec count(0)
  dt(1)2/18 ops(7) exits(2)
    0 -> dt(2) exec count(1)
    1 -> dt(4) exec count(0)
  dt(2)5/11 ops(6) exits(2)
    0 -> dt(3) exec count(3)
    1 -> dt(3) exec count(96)
  dt(3)4/22 ops(5) exits(2)
    0 -> dt(2) exec count(98)
    1 -> dt(4) exec count(1)
  dt(4)8/10 ops(3) exits(1)
0 -> dt(-1) exec count(1)

```

The numbers underlined correspond to the source line and column number. The line numbers are indicated in the margins of Figure 1-4. The `vecmax2` corresponds to the `if` statement of line 5. The `vecmax3` corresponds to the incrementation and test of the `for` (line 4).

The numbers in boldface correspond to the flow of control from one decision tree to another. There are two paths from `vecmax3` and from `vecmax2`. One is taken 96 times and one is executed three times. There is more than one path because of the `if`. There is a control flow join at `vecmax3`. This causes a decision tree break. The number of decision trees and the control flow overhead is doubled.

The two paths out of `vecmax2` correspond to the vector element either replacing the maximum or being smaller. It is much more likely that the vector element is smaller assuming the elements are randomly ordered. This is because the accumulated maximum only gets bigger. The element is smaller than the maximum 96 out of 99 times in the example. Figure 1-5 shows how to remove the decision tree break. An additional level of nesting is added. The `if` is moved out of the critical inner loop.


```

float vecmax(float *a, int size)
{
    float max = a[0];
    int i;
    for (i=1; i<size; i++) {
        for (; i<size & a[i] <= max ; i++)
            ;
        if (i < size)
            max = a[i];
    }
    return max;
}

```

Figure 1-5 Program to Find Maximum

The programs of Figure 1-4 and Figure 1-5 are equivalent, but the program of Figure 1-5 executes faster. In Figure 1-5 the 1153 instructions are necessary as opposed to 1494 previously. This corresponds to an overhead per element of 11.53 instructions per vector. The comparative performances are summarized in Table 1-1.

Table 1-1 Time to Calculate 100 Element Vector Maximum (Floating Point)

	Total Cycles	Per Element
if Statement in Inner Loop	1494	14.94
if Statement in Outer Loop	1159	11.59

Frequently, you can transform code to eliminate `if` statements. For example

```

if (p->data < v)
    cnt = cnt + 1;

```

can be replaced by

```

icnt = cnt + (p->data < v).

```

There is a conditional expression in the following preprocessor macro:

```

#define abs(v) ((v) < 0 ? -(v) : (v))

```

You can eliminate it by using a TriMedia custom operation as follows:

```
#include <ops/custom_defs.h>
...
#define abs(v) IABS(v)
```

The following preprocessor macro clips a floating point value between 10^{-38} and 10^{38} :

```
#define THRESHLO 1e-38
#define THRESHHI 1e-38
#define MINFLOAT(x, y) ((x) < (y) ? (x) : (y))
#define MAXFLOAT(x, y) ((x) > (y) ? (x) : (y))
#define CLIP(x) MAXFLOAT(MINFLOAT(x, THRESHHI), THRESHLO)
```

Assuming the values are positive, you can eliminate the conditional expressions as follows:

```
#include <ops/custom_defs.h>
...
#define MINFLOAT(x, y) FMIN(x, y)
#define MAXFLOAT(x, y) FMAX(x, y)
#define CLIP(x) MAXFLOAT(MINFLOAT(x, THRESHHI), THRESHLO)
```

Table 1-2 provides a list of such transformations. Some transformations are applied automatically by the compiler and more will be applied in the future.

Table 1-2 Code Transformation

Original Code	Transformed code	Notes
if (e ₁) v += e ₂	v += INONZERO(e ₁ , e ₂)	1,2,4
if (e ₁ < e ₂) v += 1	v += e ₁ < e ₂	1, 2,5,6
if (e ₁) v = 0;	v = INONZERO(e ₁ , v)	1,2
(e ₁ ? e ₂ : 0)	INONZERO(e ₁ , e ₂)	
if (e ₁) v = e ₂	v = INONZERO(e ₁ , v-(e ₂)) + (e ₂)	1,2
(e ₁ ? e ₂ : e ₃)	(e ₂ + INONZERO(e ₁ , (e ₃)-(e ₂)))	1,2
if (e ₁) v = -v;	v = IFLIP(e ₁ , v)	
(e ₁ != 0 ? -e ₂ : e ₂)	IFLIP(e ₁ , e ₂)	
if (v < 0) v = -v	v = IABS(v) v = FABS(v)	
(e ₁ < 0 ? -e ₁ : e ₁)	IABS(e ₁) FABS(e ₁)	

Table 1-2 Code Transformation (continued)

Original Code	Transformed code	Notes
$(e_1 < e_2 ? e_1 : e_2)$	IMIN(e_1, e_2) FMIN(e_1, e_2)	for values must be positive
$(e_1 > e_2 ? e_1 : e_2)$	IMAX(e_1, e_2) FMAX(e_1, e_2) UMAX(e_1, e_2)	for FMAX values must be positive 7
$\max(\min(e_1, e_2), \sim e_2)$	ICLIPI(e_1, e_2)	8

NOTE

- (1) e_2 should not contain side effects.
- (2) v must contain no side effects. If it is an indirection, the address must be valid.
- (3) float e_1, e_2 . Values must be non-negative.
- (4) idem $-=, *=, \&=, |=, <<=, >>=, \dots$
- (5) idem $-=, *=, <<=, >>=, ++, --$.
- (6) idem $>, >=, <=, ==, !=, \&\&, ||$.
- (7) unsigned e_1, e_2 .
- (8) Clips to $\sim e_2 .. e_2$.

Collapse Mutually Exclusive `if` Statements

Figure 1-6 shows two ways to calculate the minimum and maximum of an array. In the program on the left, there is a control flow join after the first `if`, which adds a decision tree. The 2088 instruction cycles are necessary to calculate the minimum and maximum of a 100-element, sorted vector with global optimization.

```
void minmax(float *a, int size, float *res)
{
    int i;
    float min, max;
    min = max = a[0];
    for (i=1; i<size; i++) {
        if (a[i] > max)
            max = a[i];
        if (a[i] < min)
            min = a[i];
    }
    res[0] = min;
    res[1] = max;
}
```

```
void minmax(float *a, int size, float
*res)
{
    int i;
    float min, max;
    min = max = a[0];
    for (i=1; i<size; i++) {
        if (a[i] > max)
            max = a[i];
        else if (a[i] < min)
            min = a[i];
    }
    res[0] = min;
    res[1] = max;
}
```

Figure 1-6 Vector Minimum and Maximum

The two if statements are never true simultaneously. In the program on the right, they have been folded into a single decision tree. 1494 cycles are necessary to calculate the minimum and maximum of the vector after transformation. Table 1-3 summarizes the performances of the two programs with and without grafting.

Table 1-3 Time to Calculate the Minimum and Maximum of 100 Elements (Floating Point, Cycles)

Elements	Without Grafting	With Grafting
Two Independent if statements	2088	602
if .. else if	1494	382

Use MUX and FMUX Pseudo Operations

The TriMedia C compiler eliminates some decision trees corresponding to simple if-statements and conditional (?:) expressions. This is done using MUX and FMUX pseudo operations. The transformation is not possible in some cases, for example, if there is a store through a pointer. The transformation also depends on the number of operations and on the size of the decision tree. The **tmccom** option `-max_if_size` can be used to control the size. You can eliminate the decision trees corresponding to the if statements in Figure 1-3 and Figure 1-4 by compiling with the options `-O3 -tmccom -max_if_size 10 -dirty_float --`. For more information, see the manual pages for **tmcc** and **tmccom** in Chapter 7 of “Programming and Development Tools.”

You can use MUX and FMUX directly. MUX and FMUX select between the second and third arguments depending on the value of the first argument. Use MUX for values of integer and FMUX for values of floating point type. Figure 1-7 shows how to recode the program using FMUX. 603 instruction cycles are necessary as compared to 1459 for using an if statement. Table 1-4 compares performance of the program with FMUX and the unoptimized program. The header file `<ops/custom_defs.h>` must be included to use such operations.

Table 1-4 Time to Calculate Maximum of 100 Element Vector

Loops	Total Cycles	Per Element
Loop with if	1494	11.59
Loop with FMUX	603	6.03

```

#include <stdio.h>
#include <ops/custom_defs.h>
float vecmax(float *a, int size) {
    float max = a[0];
    int i;
    for (i=1; i<size; i++) {
        max = fmux(a[i] > max, a[i], max);
    }
    return max;
}

```

Figure 1-7 Maximum with FMUX

Parallel Reduction Loops

The loop of Figure 1-7 is called a *reduction* because it reduces the dimension of a vector. Many loops in multimedia and DSP applications are reductions. Operations such as computing a vector sum or product are reductions. Scalar product computations are reductions also.

Unrolling is of limited effectiveness on reduction loops because of the loop-carried dependence on the scalar variable (for example, max). Unrolling the loop of Figure 1-7 four times produces less improvement in performance than using grafting (384 versus 338 cycles). You can optimize reductions by using the mathematical laws of commutativity and associativity to reorganize the order of computation.

In the program of Figure 1-8, four copies have been introduced for the reduction variable. Four independent maximums are computed on four slices of the vector. You can reduce the four results to a single operation using three FMUX operations. Table 1-5 compares the performance of the two loops.

```

#include <ops/custom_defs.h>
float vecmax(float *a, int size) {
    float max0 = a[0], max1 = a[1], max2 = a[2], max3 = a[3];
    int i;
    for (i=4; i<size; i+=4) {
        max0 = fmux(a[i] > max0, a[i], max0);
        max1 = fmux(a[i+1] > max1, a[i+1], max1);
        max2 = fmux(a[i+2] > max2, a[i+2], max2);
        max3 = fmux(a[i+3] > max3, a[i+3], max3);
    }
    max0 = fmux(max0 > max1, max0, max1);
    max2 = fmux(max2 > max3, max2, max3);
    return fmux(max0 > max2, max0, max2);
}

```

Figure 1-8 Reduction Variable Copying

If you know the vector values to be non-negative use `FMIN`, `FMAX` and `IMIN`, `IMAX` instead of `FMUX` and `MUX`; this saves instructions. Calculate integer minimum and maximum using `IMIN` and `IMAX`.

Floating point addition is not associative. You should not optimize reductions using floating point addition if the result of the transformation must be bit exact.

Table 1-5 Time to Calculate Maximum of 100 Element Vector (-O3)

	Total Cycles	Per Element
Reduction Using one <code>FMUX</code>	603	6.03
Reduction Using four <code>FMUX</code>	228	2.28

Use `MUX` on Variable Length Loops

The program of Figure 1-8 was unrolled four times. If the vector length n is not a multiple of the loop step m , the program does not work. There are a number of variable elements equal to the remainder, $n \bmod m$.

You can deal with the variable elements by exploiting the mathematical properties of a group. The identity i of a group is such that $x \text{ op } i = x$ for all elements. The intent is to round up the number of elements to the loop step by appending values equal to the identity element. For example, if a vector sum is being computed we round up by appending trailing zeroes ($x + 0 = x$). If a vector product is being computed, we round up by appending trailing ones ($x * 1 = x$). To round up the vector, $n - (n \bmod m)$ elements need to be added.

```

#include <ops/custom_defs.h>
#include <float.h>
#define STEP 4
float vecmax(float *a, int size) {
    int i, adj;
    float max0, max1, max2, max3;
    adj = size & (STEP-1);
    size &= ~ (STEP-1);
    max0 = fmux(adj>0, a[size], -FLT_MAX);
    max1 = fmux(adj>1, a[size+1], -FLT_MAX);
    max2 = fmux(adj>2, a[size+2], -FLT_MAX);
    max3 = -FLT_MAX;
    for (i=0; i<size; i+=STEP) {
        max0 = fmux(a[i] > max0, a[i], max0);
        max1 = fmux(a[i+1] > max1, a[i+1], max1);
        max2 = fmux(a[i+2] > max2, a[i+2], max2);
        max3 = fmux(a[i+3] > max3, a[i+3], max3);
    }
    max0 = fmux(max0 > max1, max0, max1);
    max2 = fmux(max2 > max3, max2, max3);
    return fmux(max0 > max2, max0, max2);
}

```

Figure 1-9 Unrolling Variable-Length Loops

For a loop step of four, this corresponds to one to four elements. Figure 1-9 shows code for calculating the maximum of a vector of arbitrary length. For each of the reduction variables (`max0`, `max1`, `max2`, `max3`), you need to make a selection between an element at the end of the vector and the identity element. You can do this with an `FMUX`. The initialization code before the `for` does this. The identity for the operation being calculated (the maximum) is negative infinity ($\max(x - \infty) = x$). The identity for the minimum is $(+\infty)$.

In Figure 1-9, the loop step is a power of two. This allows a bitwise and (`&`) to replace a modulus operation. The `&` has a single cycle latency. The number of iterations needs to be rounded down to a multiple of the loop step. You can do this with an `&` also. Integer division and modulus are 50 times slower.

Many DSP kernels are sum reductions for which the identity is zero. In such cases, you should use the Trimedia custom operation `IZERO` to initialize the reduction variables (`max0`, `max1`, `max2`, `max3`) instead of `MUX` and `FMUX`. It selects between zero and a value in one instruction. Use `FZERO` for floating point types.

Table 1-6 compares performances of a 100-element vector maximum for a four-way unrolled loop. Twelve cycles of overhead are necessary to deal with the remaining elements.

Table 1-6 Time to Calculate Maximum of 100 Element Vector (-O3, four-way unrolled loop)

	Total Cycles	Per Element
Length a Multiple of Four	228	2.28
Arbitrary Length	240	2.40

Apply Strength Reduction

Figure 1-10 shows two procedures to normalize a vector. In the program to the left, the individual elements are divided by the sum of values. For 100-vector elements, 2713 instruction cycles are necessary for the program with global optimization. Floating point division requires 17 cycles on TriMedia. The divisions correspond to 1700 of the cycles.

```
norm(float *a, int size) {
    int i;
    float sum = 0.0;
    for (i=0; i<size; i++)
        sum = sum + a[i];
    for (i=0; i<size; i++)
        a[i] = a[i] / sum;
}
```

```
norm(float *a, int size) {
    int i;
    float sum = 0.0, invsum;
    for (i=0; i<size; i++)
        sum = sum + a[i];
    invsum = 1.0 / sum
    for (i=0; i<size; i++)
        a[i] = a[i] * invsum;
}
```

Figure 1-10 Vector Normalization Procedures

Floating point multiplication requires only three cycles. In the program to the right, the division is replaced by a multiplication by the reciprocal. Doing so saves 14 cycles ($14=17-3$) per division. You can calculate the reciprocal using a single division outside the loop. 99 of the 100 divisions can be replaced by a multiplication. This corresponds to a reduction in the total execution time of 1386 cycles ($99*14$). An optimization such as this, which replaces a costly operator by a less expensive one, is called *strength reduction*. The result of the reciprocal followed by the multiplication can vary from the division in the low order bit.

Table 1-7 compares the performance of vector normalization with division, with multiplication, and with and without grafting. Grafting produces only a limited performance improvement in the presence of division. This is because there is one divide unit and a division can be issued only once every 17 cycles. Grafting more than doubles

the performance with multiplication. There are two multiply units and an instruction can issue each cycle.

Table 1-7 Time to Normalize 100 Elements (Floating Point)

	Number of Instruction Cycles	
	Without Grafting	With Grafting
Normalization Using Division	2713	2090
Normalization by Multiplication by the Reciprocal	1327	559

Figure 1-11 shows two procedures that sum a 12 x 12 matrix by column. The element (i, j) of a n x m matrix is at offset (n*i+j) elements from the base. In the program on the right, strength reduction has been applied to replace the multiplication by an addition. The program on the left requires 141 cycles at optimization level -O3. The program on the right requires 101 cycles at optimization level -O2.

<pre>int matrix[12][12]; int colsum(int col) { int i, sum = 0; for (i=0; i<12; i++) sum += matrix[i][col]; return sum; }</pre>	<pre>int matrix[12][12]; int colsum(int col) { int i, sum = 0, *pcol; pcol = &matrix[0][col]; for (i=0; i<12; i++) { sum += *pcol; pcol += 12; } return sum; }</pre>
---	---

Figure 1-11 Two procedures to sum a 12 x 12 matrix by column

Treename	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
-----	-----	-----	-----	-----	-----
__rt_imod	10	490	67.59%	0	0
__gcd_DT_3	10	50	6.90%	0	0
__gcd_DT_2	10	40	5.52%	0	0
(...)					
exact total machine cycles is 725 cycles.					

Figure 1-12 shows two programs to calculate the greatest common divisor (g.c.d.). The program on the left calculates the g.c.d. using integer arithmetic.

The **tmprof** output is generated without the `-func` option. The **tmprof** output from running it is shown below:

```
gcd(int u, int v) {
    unsigned t;
    if (u > v) { t = u; u = v; v = t; }
    while (u > 0) {
        t = u;
        u = v % u;
        v = t;
    }
    return v;
}

main() {
    (void) gcd(12381203, 41231207);
}
```

```
gcd(int u, int v) {
    int t;
    if (u > v) { t = u; u = v; v = t; }
    while (u > 0) {
        t = u;
        u = v - (int)((float)v/u) * u;
        v = t;
    }
    return v;
}

main() {
    (void) gcd(12381203, 41231207);
}
```

Figure 1-12 Two programs to calculate greatest common divisor

There are 580 cycles necessary to calculate the g.c.d. Most of the time is spent in the subroutine `_rt_imod`. This subroutine calculates the remainder for signed integers. 49 cycles are necessary for each execution of `rt_imod` (490 cycles for ten calls), not including call overhead. The `rt_umod` subroutine calculates the remainder for unsigned integers.

The program to the right uses floating point arithmetic. Calculating the remainder in floating point requires 24 cycles. Seventeen cycles are required for the division, three are required for the floating-point-to integer conversion, three are required for the multiplication, and one is required for the subtraction. You need 334 cycles to calculate the g.c.d. There is a saving of 254 ($254=580-334$) cycles for ten remainders (25.4 cycles per remainder) using floating point.

The g.c.d. of 12,381,203 and 41,231,207 is one. Both algorithms give the correct value. The g.c.d. of 268,435,454 and 268,435,582 is two. The algorithm to the right of Figure 4-8 calculates a g.c.d. of 128. The value is incorrect because the values are outside the range $[-2^{24}, 2^{24}]$ (floating-point numbers are represented in 24 bits).

Figure 1-14 shows two ways to subsample a vector with a 2:3 ratio. In the program to the left, the array index is calculated uses an integer division and multiplication. The **tmprof** output for subsampling a 100-element vector is as shown below.

Funcname	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
_rt_idiv	100	4500	79.70%	0	0
subsample	1	1011	17.91%	0	0
(...)					

```

subsample(char *a, char *b, int n) {
    int i;
    for (i=0; i<n; i++ )
        a[i] = b[i*2/3];
}

```

```

subsample(char *a, char *b, int n) {
    int i;
    for (i=0; i<n; i++ )
        a[i] = b[(int)(i*(2.0/3))];
}

```

Figure 1-13 Two ways to subsample a vector with a 2:3 ratio

Most of the time is spent in the subroutine `rt_idiv`. This subroutine implements division for signed integers. 45 cycles are needed per call, not including call overhead. `rt_udiv` implements division for unsigned integers.

Using the algorithm to the right, 1310 cycles are necessary to subsample a 100-element vector. The two algorithms compute the same value for $n < 2^{24}$.

The execution time is reduced by only 400 cycles (7%) when grafting is applied to the program to the left. The subroutine call for the division limits the effectiveness of grafting while it reduces the time for the program to the right from 1310 to 511 cycles. This corresponds to a saving of 61 percent in the execution time (799 cycles). Table 1-8 compares performance, with and without grafting.

Table 1-8 Time to Subsample 100-Element Vector

	Number of Instruction Cycles		Reduction (%)
	Without Grafting	With Grafting	
Subsampling with Integer Divide and Multiply	5611	5211	7.2
Subsampling Using Floating Point	1310	511	60.9

For a variable of signed type, replacing a division by 2^n by a shift (`>>`) eliminates three operations from the program and saves three cycles. The result differs by one from that of the division operator (`/`) if it is negative and there is a remainder. Replacing `x%2n` by `(x & (2n - 1))` eliminates three instructions and saves three cycles. The result is positive or zero. Integer remainder produces a negative or zero result if the result of the division is negative. If the variable is known to be non-negative, changing the type to unsigned obtains the same effect automatically. The results of `>>` and `&` correspond to the mathematical definitions of division and remainder.

Move Externals and Reference Parameters to Locals

You cannot allocate external variables to registers and require memory references. Accesses have a latency of three cycles. Copying an external variable to a local variable can improve performance substantially in time-critical parts of the code. Figure 1-14 shows two ways of writing a program to push an array onto a stack. The stack pointer is contained in the external variable `stackp`. 1911 instruction cycles are necessary to push 100 words. 1644 cycles are necessary using `-A2` (relaxed alias analysis). Only 1256 cycles are necessary if `stackp` is copied to a local variable.

The function of Figure 1-15 takes binary data as input, with one bit per byte. The result is packed into a 32-bit word. A pointer to the current pointer in the input is passed by reference. For the program on the left, 202 cycles are necessary to pack 32 bytes of data with global optimization. Most of this time is spent in the decision tree corresponding to the “for: packbits₁”. There are 145 cycles necessary after grafting. The `-statfile` output for `packbits` with grafting is given below:

tree name	execs	instc	isopers	exopers
__packbits_DT_1	8	136	368	368

After grafting, the loop is executed eight times as compared to 32 times. This corresponds to a grafting factor of four. For each iteration, 17 cycles ($=136/8$) are necessary to execute 46 ($368/8$) operations. This corresponds to an ILP of 2.70.

```

int *pusha(int nargs, int *p)
{
    int *oldstkp;
    extern int *stackp, stack[NSTACK];

    /* save the old stack pointer */
    oldstkp = stackp;

    /* save each node pointer */
    while ( nargs-- ) {
        if (stackp <= stack)
            abort("evaluation stack
overflow");
        *--stackp = *p++;
    }

    /* return the old stack pointer */
    return oldstkp;
}

```

```

int *pusha(int nargs, int *p)
{
    int *newstkp, *oldstkp;
    extern int *stackp, stack[NSTACK];

    /* save the old stack pointer */
    oldstkp = stackp;

    newstkp = stackp;
    /* save each node pointer */
    while (nargs--){
        if (newstkp <= stack)
            abort("evaluation stack
overflow");
        *--newstkp = *p++;
    }

    stackp = newstkp;

    /* return the old stack pointer */
    return oldstkp;
}

```

Figure 1-14 Pushing an Array Onto a Stack

In the program on the right, the reference parameter `bytepp` has been copied to a local. This allows a store through a pointer to be removed from the loop. 145 cycles are necessary with global optimization and without grafting. 92 cycles are necessary with grafting. The **tmprof** output for `packbits` with grafting follows:

tree name	execs	instc	isopers	exopers
<code>__packbits_DT_1</code>	8	83	336	336

For each iteration, about ten ($83/8$) cycles are necessary to execute 42 ($=336/8$) operations. The four (46-42) operations are store instructions. This corresponds to an ILP of 4.04. There are fewer cycles in the loop because four loads can proceed in parallel. In the program on the left, the loads need to be ordered with respect to the stores.

Table 1-9 summarizes the performances with and without grafting.

Table 1-9 Effect of Copying Externals and Pointer References to Locals (-O3)

	Without Grafting	With Grafting
Figure 1-14		
External Variable in Loop	1610	760
Local Copy in Loop	1111	548
Figure 1-15		
Call By Reference Parameter in Loop	265	145
Local Copy in Loop	201	92

```
packbits(unsigned char**bytepp,int
count){      int i ;
      unsignedresult = 0 ;

      for ( i = 0 ; i < count ; i ++ ) {
          result |= **bytepp << i ;
          (*bytepp) ++ ;
      }

      return result ;
}
```

```
packbits(unsigned char**bytepp,int
count){      int i ;
      unsignedresult = 0 ;
      char *bytep = *bytepp;
      for ( i = 0 ; i < count ; i ++ ) {
          result |= *bytep << i ;
          bytep++;
      }
      *bytepp = bytep;
      return result ;
}
```

Figure 1-15 Program to Pack Bytes in a Word

Remove Function Calls

Figure 1-16 shows two programs that calculate the square of the distance from the origin for a collection of points, (x_i, y_i) . For the program to the left, 326 cycles are necessary with global optimization alone. 245 instruction cycles are necessary with global execution and grafting. The execution time is reduced only by 25 percent with grafting. This is because of the function call. Grafting does not cross function call boundaries. Each invocation of a function also adds a decision tree. In the program on the right, the function call has been inlined using the C preprocessor, **cpp**. 224 cycles are necessary without grafting and 78 are necessary with grafting (a reduction of 78%).

In the program of Figure 1-15, the function call to abort corresponds to an error (stack overflow). This occurs very rarely. Although the function does not return, the compiler does not know this. In Figure 1-17 the function call has been moved outside the loop. This removes a join node, allowing the loop to be represented by one decision tree. The 622

instruction cycles are necessary with global optimization and without grafting, compared to 1111 previously. The 418 instruction cycles are necessary with grafting.

The ILP is still limited because of the stores through pointers. Adding the restrict qualifier to the definitions of `newstkp` and `p` reduces the execution time to 320 cycles with grafting. This corresponds to a reduction of 24%. The performances are summarized in Table 1-10.

Table 1-10 Performance Summaries

	Without Grafting	With Grafting
Call + Local Copy in Loop (Table 1-9)	1111	548
Local Copy in Loop + Call Outside (Figure 1-14)	622	418
Local Copy in Loop + Call Outside + Restrict	622	320
Call in Loop (Figure 1-16, Left)	265	145
Inlining (Figure 1-16, Right)	201	92

Note

An inline directive will be supported in a future version of the TCS ♦

```
float hypot(float x, float y) {
    return x*x + y*y;
}

main() {
    float x[20], y[20], rad[20];
    int i;
    for (i=0; i<20; i++)
        rad[i] = hypot(x[i], y[i]);
}
```

```
#define hypot(x, y) (x)*(x) + (y)*(y)

main() {
    float x[20], y[20], rad[20];
    int i;
    for (i=0; i<20; i++)
        rad[i] = hypot(x[i], y[i]);
}
```

Figure 1-16 Program to Calculate Distance Vector

```

int *pusha(int nargs, int *p)
{
    int *newstkp, *oldstkp;
    extern int *stackp, stack[NSTACK];

    /* save the old stack pointer */
    oldstkp = stackp;

    newstkp = stackp;
    /* save each node pointer */
    while (nargs-- && newstkp>stack){
        *--newstkp = *p++;
    }
    if (newstkp <= stack)
        abort("evaluation stack overflow");
    stackp = newstkp;

    /* return the old stack pointer */
    return oldstkp;
}

```

Figure 1-17 Program to Push Arguments on the Stack

Pay Attention to Compile Time

Figure 1-19 shows a program that multiplies a 40 x 10 matrix by a 10 x 20 matrix, giving a 40 x 20 result ($c = a*b$). The source program is 29 lines long. Six minutes, ten seconds are necessary to compile the program.

```

int a[40][10], b[10][20], c[40][20];
main() {
    int i;
    for (i=0; i<40; i++) {
        c[i][0] = a[i][0]*b[0][0] + a[i][1]*b[1][0] + ... + a[i][9]*b[9][0];
        c[i][1] = a[i][0]*b[0][1] + a[i][1]*b[1][1] + ... + a[i][9]*b[9][1];
        c[i][2] = a[i][0]*b[0][2] + a[i][1]*b[1][2] + ... + a[i][9]*b[9][2];
        c[i][3] = a[i][0]*b[0][3] + a[i][1]*b[1][3] + ... + a[i][9]*b[9][3];
        (...)
        c[i][18] = a[i][0]*b[0][18] + a[i][1]*b[1][18] + ... + a[i][9]*b[9][18];
        c[i][19] = a[i][0]*b[0][19] + a[i][1]*b[1][19] + ... + a[i][9]*b[9][19];
    }
}

```

Figure 1-18 Matrix Multiply Program matmul_1.c

The `-vtimes` option of **tmcc** reports on the execution times of the individual phases. The `-K` option tells **tmcc** to keep intermediate output files around (that is, `matmul_1.t`, `matmul_1.s`, and `matmul_1.o`). In this case, almost all the time can be observed to be spent in the TriMedia scheduler **tmsched**. On WindowsNT and Windows 95, the program running is shown in the menu bar.

```
$ tmcc -vtimes -K matmul_1.c
cpp:      0.033
tmccom:  0.967
tmsched: 360.719
      ...
total:   363.035
```

Almost all the execution time in the program is spent in the decision tree corresponding to the `for`, `main1`. You can determine the number of operations in the decision tree by examining the tree's output file `tmsim.t` produced by the **tmcc** command. Scheduling time is nonlinear with respect to the number of operations. There are 1018 operations in `main1`.

Unusually long scheduling times are typically the consequence of feeding **tmsched** a decision tree with too many operations. In the program below, the two inner loops of the multiplication have been completely unrolled. Decision trees that are too long result in reduced performance due to scheduler spilling. Compile time is a performance indicator.

In the program, each iteration reads a row of `matrix.a` (10 accesses). The matrix `b` is read entirely (10 x 20 accesses) and a single element $c_{i,j}$ is computed for each column (20 accesses). There are 40 (200 + 10) reads and 20 writes for each of the 40 iterations. The 9200 memory accesses are necessary in total (8400 reads and 800 writes). You can determine the actual number of memory accesses by using **tmsim** with the `-v` option.

```
$ tmsim -v a.out
      (...)
data cache statistics: size 16 kB, blocksize 64 b, associativity 8
nr of accesses: 21946
```

There are 12746 (=21946- 9200) more operations than expected. Almost all the accesses are 32-bit accesses in the decision tree corresponding to the `for`. You can differentiate spills from nonspills as follows:

```
$ grep ld32 matmul_1.s | wc -l
  324
$ grep st32 matmul_1.s | wc -l
  224
$ grep "ld32.*-- SPILL" matmul_1.s | wc -l
  114
$ grep "st32.*-- SPILL" matmul_1.s | wc -l
  204
```

```
$ grep ld32 matmul_1.s | wc -l
  324
$ grep st32 matmul_1.s | wc -l
  224
$ grep "ld32.*-- SPILL" matmul_1.s | wc -l
  114
$ grep "st32.*-- SPILL" matmul_1.s | wc -l
  204
```

The number of nonspill accesses is given by subtracting the spill accesses from the number of total accesses. There are 210 (= 324 - 114) nonspill loads, and 20 (=224 - 204) nonspill stores per iteration. This corresponds to 9200 accesses in total, as expected. Spilling is responsible for 318 (=114 + 204) memory accesses per iteration. 12720 (=318 x 40) accesses correspond to scheduler spills.

Use `#pragma TCS_break_dtree`

The interrupt mechanism of TriMedia is discussed in the *TM-1000 Data Book*. Interrupts only occur when control passes from one decision tree to another. Decision-tree breaks (`#pragma TCS_break_dtree`) limit the length of a decision tree, allowing control over interrupt latency.

They can also be used to improve the performance of a program. Spilling in case of program in Figure 1-18 is a result of excessive register pressure due to the high degree of unrolling. In Figure 1-19, the compiler `pragma TCS_break_dtree` is used to remove spilling. The loop is split into two decision trees.

The performances are summarized in Table 1-11. Introducing a decision tree break completely removes the spills. The extra nonspill loads are because a matrix a row of a (ten elements) needs to be reread.

```
int a[40][10], b[10][20], c[40][20];
main() {
    int i;
    for (i=0; i<40; i++) {
        c[i][0] = a[i][0]*b[0][0] + a[i][1]*b[1][0] + ... + a[i][9]*b[9][0];
        c[i][1] = a[i][0]*b[0][1] + a[i][1]*b[1][1] + ... + a[i][9]*b[9][1];
        ...
        c[i][8] = a[i][0]*b[0][8] + a[i][1]*b[1][8] + ... + a[i][9]*b[9][8];
        #pragma TCS break dtree
        c[i][9] = a[i][0]*b[0][9] + a[i][1]*b[1][9] + ... + a[i][9]*b[9][9];
                (... )
        c[i][18] = a[i][0]*b[0][18] + a[i][1]*b[1][18] + ... + a[i][9]*b[9][18];
        c[i][19] = a[i][0]*b[0][19] + a[i][1]*b[1][19] + ... + a[i][9]*b[9][19];
    }
}
```

Figure 1-19 Matrix Multiply Program mtmul_1.c Compiler Pragma TCS_break_dtree added

You can also use decision tree breaks to prune infrequently executed branches from a decision tree. In the program of Figure 1-20, the trailing return statement is included in the decision-tree for the loop. The computations for the return increase the length of the critical path. Inserting a decision-tree break at the end of the loop reduces the execution time per iteration from seven to six cycles.

Table 1-11 Effect of Decision Tree Break on Unrolled Matrix Multiply (per iteration)

	Non-Spill Loads	Non-Spill Stores	Spill Loads	Spill Stores
without TCS_break_dtree	210	20	114	204
with TCS_break_dtree	220	20	0	0

```

void compare(int *a, int *b, int size, int *pcount)
{
    int i;
    int count;
    count = 0;
    i = 1;
    do {
        count = count + (a[i]==b[i]);
    } while (i++ < size);
    #pragma TCS_break_dtree
    *pcount = count;
}

```

Figure 1-20 Loop Pruning

Use Goto for Loops with a Trailing if Statement

The TriMedia core compiler **tmccom** transforms for and while loops into do while loops by peeling off the first iteration. This pushes the loop condition from the root of the decision tree to the end of the loop. If the loop contains a trailing if statement, there is an extra join for the loop condition after the if statement. There is an extra decision tree. You can avoid this by using goto. You can represent simple loops containing a trailing if or if else statement in one decision tree. Figure 1-21 shows the program of Figure 1-6 programmed with goto. Several tricks are used. i is post-incremented to reduce latency. A copy is made in t in parallel. The goto is replicated to avoid creating a decision tree having only a jump. Table 1-12 compares performances. You should take care when applying this. As the table shows, there is a slight performance degradation with grafting.

```

void minmax(float *a, int size, float *res) {
    int t , i = 1; float min = a[0], max = a[0];
loop: t = i;
    if (i++ < size)
        if (a[t] > max) {
            max = a[t]; goto loop;
        } else if (a[t] < min) {
            min = a[t]; goto loop;
        }
    res[0] = min; res[1] = max;
}

```

Figure 1-21 Vector Minimum and Maximum(goto)

Table 1-12 Cycles to Calculate Minimum and Maximum of 100-Element Vector (Floating Point)

	Without Grafting	With Grafting
for Loop (Figure 1-6)	905	365
if .. goto Loop (Figure 1-21)	1395	357

Loop Fusion

The program of Figure 1-22 calculates the mean and variance for an array of n elements. The mean is equal to the sum of array values divided by the array size. The variance is equal to the sum of squares divided by the array size minus the square of the mean. The first loop calculates the sum of values. The second loop calculates the sum of squares. Loop fusion merges two loops that are executed the same number of times into a single loop. Loop fusion eliminates half the overhead. The program on the right illustrates the application of fusion. Table 1-13 compares performances for an array of 100 elements. The overhead per element is reduced by 28% without grafting and by 23% with grafting.

```
float meanvar(float *a,int n,float
*var){
    float sum = 0, sumsq = 0, mean,
ninv;
    int i;
    for (i=0; i<n; i++)
        sum = sum + a[i];
    for (i=0; i<n; i++)
        sumsq = sumsq + a[i]*a[i];
    ninv = 1/n;
    mean = sum * ninv;
    *var = sumsq*ninv - mean*mean;
    return mean;
}
```

```
void vecnorm(float *a, int size) {
    int i;
    float sum = 0.0, invsum;
    for (i=0; i<size; i++)
        sum = sum + a[i];
    invsum = 1.0/sum;
    for (i=0; i<size; i++)
        a[i] = a[i] * invsum;
}
```

Figure 1-22 Loop Fusion**Table 1-13** Effect of Loop Fusion on Calculation of Mean and Variance (Instructions/Element)

	Without Grafting	With Grafting
Separate Loops to Calculate Mean and Variance	15.26	7.10
Fusion of Two Loops	11.11	5.48

Replace `||` by `|`

C has a number of constructs, including the `&&`, `|`, and `?:` operators, that were designed for efficient execution on a sequential processor. Their operands cannot be evaluated in parallel. Use of these operators can increase the number of decision trees.

You can replace the expression $(E_1 \ || \ E_2)$ by $(E_1 \ | \ E_2)$ if two conditions are satisfied. It must be evaluated for its effects on control flow. E_1 and E_2 must have no side effects. If the expression is being evaluated for its value, it can be replaced by $(E_1 \ | \ E_2) != 0$. Boolean or (`|`) operators add a decision tree to the program both when used in a control statement (`if`, `while`, `for`, or `do while`) and inside a `?:` expression.

Replace `&&` by `&` or `IZERO`

The `IZERO` custom operation has a value 0 if its first operand is zero; otherwise, it has the value of its second operand. You can replace a boolean and operator (`&&`) by `IZERO` if the expression has no side effects and it is being evaluated for its effect on control flow.

Boolean and operators add a decision tree when used in a `?:` expression or a two-sided `if` statement but not inside an `else if` statement or as the condition of a `for` or `while`.

If the value of the expression is needed, it can be replaced by `IZERO(E1, E2 != 0)` or `IZERO(E1, E2) != 0`, depending on which has the better critical path.

The program of Figure 1-23 counts the number of alphabetic characters and underscores in a string. If the operands of a `&&` operation are constrained to a boolean (0/1) value `t` you can use a bitwise “and” (`&`) operator. The operands in the figure are relationals with a boolean value. Replacing the `&&` and `||` operators by `&` and `|` reduces the execution time from 1484 to 693 cycles.

Table 1-14 Effect of Eliminating `&&` and `||` Operators (Instruction Cycles)

Program of Fig. 4-19 with <code>&&</code> and <code> </code> Operators	1347
Program with <code>&</code> and <code> </code> Operators	556

```

#define alpha(c) (((c) >= 'a' && c<='z') ||
(c>='A' && c<='Z') || c=='_')
int alphacount(char *s)
{
    int c, count;
    count = 0;
    while (c = *s++)
        count = count + alpha(c);
    return count;
}

main()
{
    alphacount("Now is the time for all good men to come to the aid of
their country");
}

```

Figure 1-23 Character Count

Using Software Pipelining

Typically, the three instructions after a jump operation are mostly unused. Data that is needed in the next iteration can be preloaded in these slots. This is called software pipelining. Figure 1-24 shows the C string comparison routine `strcmp`, before and after software pipelining. Eight cycles are needed for the loop for the program on the left with global optimization. Software pipelining allows the loop to execute in five cycles. You can also schedule long-latency operations (for example, floating point) in these slots.

A simple form of software pipelining is implemented by the global optimizer if grafting is not enabled.

```

strcmp(char *p, char *q)
{
    int c, c1;
    while ((c = *p++) == *q++ && c)
        ;
    return c - q[-1];
}

```

```

strcmp(char *p, char *q)
{
    int c, c1, cont;
    c = *p++;
    c1 = *q++;
    cont = (c == c1) & (c != 0);
loop: if (cont) {
        c = *p++;
        c1 = *q++;
        cont = (c == c1) & (c != 0);
        goto loop;
    }
    return c - c1;
}

```

Figure 1-24 Example of Software Pipelining

Use Trimedia Style Booleans in Critical Parts of the Code

In C, true and false are represented by nonzero and zero, respectively. TM-1000 uses the low-order (guard) bit of a register to determine whether a condition is true or false. Even if the expression being tested is compared against zero, a comparison operator is required. Typically, five cycles are required for a conditional jump.

If the value of the expression is known to depend only on the low order bit, the comparison is not necessary. The TriMedia C compiler recognizes expressions of the form $(E_1 \& 1)$ or $!(E_1 \& 1)$. Using these instead of $(E_1 != 0)$ or $(E_1 == 0)$ generates better code. Expressions such as $(E_1 \& 2^n)$ and $!(E_1 \& 2^n)$ are optimized also.

You can use Trimedia style booleans with MUX and FMUX. If the guard is known to depend on the first order bit, you can use the machine level pseudo operations mux and fmux instead. For example, if the variable *v* is constrained to a 0/1 value, you can replace $MUX(v != 0, E_1, E_2)$ by $mux(v, E_1, E_2)$, saving a cycle.

Loop Unrolling

You can perform loop unrolling manually. The loop in Figure 1-1 is shown unrolled in Figure 1-25, where the inner for loop is completely unrolled—that is, replaced with eight assignment statements. The outer for loop is unrolled four times. Replacing the convolution function with `unrolled_direct_convolution` gives us the new program `fir2.c`.

Note that loop unrolling is a specialized version of grafting. In loop unrolling, a conditional jump from a decision tree exit back to itself is replaced with the code for the decision tree. The main difference is that grafting replaces the jump part of the conditional jump with the destination decision tree but leaves the condition in place, which causes control dependence between one iteration of the loop to the next.

For example, the grafting shown in Figure 1-3 is essentially loop unrolling, and it can be seen that the grafted code is still governed by a condition. Without data-flow analysis, such conditions cannot be removed and thus result in a lower performance, compared to manual unrolling.


```

void
unrolled_direct_convolution( char *a, char *b, int *c )
{
    int      k, j;

    for(k = 0; k < NROF_SAMPLES; k += 4) {

        c[0] = b[0]*a[0] + b[1]*a[-1] + b[2]*a[-2] + b[3]*a[-3]
              + b[4]*a[-4] + b[5]*a[-5] + b[6]*a[-6] + b[7]*a[-7];
        c[1] = b[0]*a[1] + b[1]*a[0] + b[2]*a[-1] + b[3]*a[-2]
              + b[4]*a[-3] + b[5]*a[-4] + b[6]*a[-5] + b[7]*a[-6];
        c[2] = b[0]*a[2] + b[1]*a[1] + b[2]*a[0] + b[3]*a[-1]
              + b[4]*a[-2] + b[5]*a[-3] + b[6]*a[-4] + b[7]*a[-5];
        c[3] = b[0]*a[3] + b[1]*a[2] + b[2]*a[1] + b[3]*a[0]
              + b[4]*a[-1] + b[5]*a[-2] + b[6]*a[-3] + b[7]*a[-4];
        a += 4;
        c += 4;
    }
}

```

Figure 1-25 Convolution Example – Loop Unrolled (Example fir2.c)

The following sequence of commands compiles fir2.c with profiling but without grafting.

```

tmcc -p fir2.c -o fir2          /* Generate program with profiling turned on */
tmsim -nomm fir2              /* Simulate intermediate code and produce
                             dtprof.out. */

tmcc -r fir2.c -o fir2        /* Recompile using profile information. */
tmsim -statfile fir2.stat fir2 /* Simulate & collect cycle accurate information. */
tmprof -scale 1 -func fir2    /* statOutput is sent to stdout. */

```

The performance of the unrolled loops is as shown in the following **tmprof** report below. The size of `text` section of the object code is 6400 bytes, which is somewhat more than the ungrafted, unrolled program fir1.c. The execution time of fir2.c is about 3.4 times faster than that of fir1.c and about 2.9 times faster than fir1.c with grafting enabled. From this we can see that grafting is not the solution to all performance problems. It helps on large parts of the code that are not very critical but still interesting, but the most critical parts can better be optimized by hand.

Funcname	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
-----	-----	-----	-----	-----
unrolled_direct_convoluti	1	7153 63.68%	300	843
initialize	1	2867 25.53%	261	84
exit	1	193 1.72%	145	22
main	1	153 1.36%	136	3
_clear_all_regs	1	129 1.15%	116	0
_clear_all_regs_1	1	129 1.15%	116	0
_pre_start	1	120 1.07%	65	45
_profile_write	1	114 1.01%	92	11
_start_1	1	89 0.79%	67	15
_start_second	1	64 0.57%	58	0
_return_custom_begin	1	62 0.55%	58	0
_default_exit_2	1	43 0.38%	0	38
_default_exit_1	1	36 0.32%	29	3
_exit	1	33 0.29%	29	0
_custom_begin	1	33 0.29%	29	0
_default_exit	1	6 0.05%	0	0
_custom_end	1	4 0.04%	0	0
_return_custom_end	1	4 0.04%	0	0
-----	-----	-----	-----	-----
total/average		11232 100.00%	1501	1064

Loop Unrolling Versus Grafting

You should apply grafting at the last stage of optimization because it is impossible to understand what is happening after grafting. You should apply loop unrolling only if it produces better performance than grafting. Consider, for example, the program in Figure 1-26, which initializes a symbol table with the list of C keywords. When compiled and run using the global optimizer (`-O3`), 1830 instruction cycles are necessary for 30 calls to `definesym` (62 cycles per call). 48 cycles per call are spent in the character copy loop. This corresponds to six cycles per copied byte.

```

#define NSYM 8
#define NFREE 100
struct symbol {
    struct symbol *next;
    char name[NSYM];
    int value;
} *avail, *symlist;

#define freesym(sym) { struct symbol *t = (sym); t->next = avail;
avail = t; }

char*keywords[] = {
    "void","char","short", "int", "long", "float", "double", "struct",
    "union", "enum", "unsigned", "auto", "extern", "static", "register",
    "goto", "switch", "case", "default", "return", "if", "else", "while",
    "do", "break", "continue", "for", "typedef", "sizeof"
    "const","volatile", 0};

struct symbol *definesym(char *str, int value) {
    int i;
    struct symbol *res = avail;
    avail = avail->next;
    for (i=0; i<NSYM; i++)
        res->name[i] = str[i];
    res->value = value;
    res->next = symlist;
    symlist = res;
    return res;
}

main() {
    int i;
    for (i=0; i<NFREE; i++)
        freesym((struct symbol *)malloc(sizeof(struct symbol)));
    for (i=0; keywords[i]; i++)
        definesym(keywords[i], i);
}

```

Figure 1-26 Symbol Table Initialization

The number of times the for loop in `definesym` is executed is known in advance. Grafting replicates the condition (`i<8`). For this reason, it is better to use loop unrolling.

Figure 1-27 shows code for the unrolled loop. Six cycles are necessary per byte using a loop. You can estimate the time corresponding to the unrolled loop as 3.25 cycles.

```

struct symbol *definesym(char *str, int value)
{
    int i;
    struct symbol *res = avail;
    avail = avail->next;

    res->name[0] = str[0]; res->name[1] = str[1];
    res->name[2] = str[2]; res->name[3] = str[3];
    res->name[4] = str[4]; res->name[5] = str[5];
    res->name[6] = str[6]; res->name[7] = str[7];

    res->value = value;
    res->next = symlist;
    symlist = res;
    return res;
}

```

Figure 1-27 Unrolled Loop

Using Restricted Pointers

C programs make heavy use of loads and stores through pointers. The C language does not allow the compiler to make any assumptions about pointers. Consider the following two assignment statements in the `for` loop of program `fir2.c`:

```

c[0] = b[0]*a[0] + b[1]*a[-1] + b[2]*a[-2] + b[3]*a[-3] + b[4]*a[-4] +
        b[5]*a[-5] + b[6]*a[-6] + b[7]*a[-7];
c[1] = b[0]*a[1] + b[1]*a[0] + b[2]*a[-1] + b[3]*a[-2] + b[4]*a[-3] +
        b[5]*a[-4] + b[6]*a[-5] + b[7]*a[-6];

```

Since `a`, `b`, and `c` are pointer parameters to the function `unrolled_direct_convolution`, in the absence of inter-procedural analysis the compiler assumes that they might refer to the same or overlapping memory locations; that is, be aliased to each other. In other words, the second assignment statement might be data dependent on the first statement. This implies that the operations of the two statements cannot be executed in parallel. However, you know that `a`, `b`, and `c` always are distinct arrays and thus never alias. You can convey this information to the compiler by declaring these pointers to be *restricted*.

Declaring pointers as restricted is a hint to the compiler that these pointers point to separate objects in memory that do not overlap with any known variable in the current context or with such an object related to any other restricted pointer. Based on this information, the compiler decides that different variables and/or restricted pointers do not alias. Note that it is your responsibility to verify that the assertion is true: proper use of restricted pointers reduces the amount of dependencies and, therefore, increases potential

parallelism. However, declaring pointers to overlapping memory regions as restricted results in an incorrect program. In our running example, we assign the type qualifier `restrict` to the declarations of `a`, `b`, and `c`, as shown in Figure 1-28.

```

restrict_direct_convolution(
    char * restrict a,
    char * restrict b,
    int * restrict c )
{
    int      k, j;

    for(k = 0; k < NROF_SAMPLES; k += 4) {

        c[0] = b[0]*a[0] + b[1]*a[-1] + b[2]*a[-2] + b[3]*a[-3]
              + b[4]*a[-4] + b[5]*a[-5] + b[6]*a[-6] + b[7]*a[-7];
        c[1] = b[0]*a[1] + b[1]*a[0] + b[2]*a[-1] + b[3]*a[-2]
              + b[4]*a[-3] + b[5]*a[-4] + b[6]*a[-5] + b[7]*a[-6];
        c[2] = b[0]*a[2] + b[1]*a[1] + b[2]*a[0] + b[3]*a[-1]
              + b[4]*a[-2] + b[5]*a[-3] + b[6]*a[-4] + b[7]*a[-5];
        c[3] = b[0]*a[3] + b[1]*a[2] + b[2]*a[1] + b[3]*a[0]
              + b[4]*a[-1] + b[5]*a[-2] + b[6]*a[-3] + b[7]*a[-4];
        a += 4;
        c += 4;
    }
}

```

Figure 1-28 Convolution Example - Restricted Pointers (example fir3.c)

The following sequence of commands compiles, profiles, and recompiles the program, and then produces a report.

```

tmcc -p fir3.c -o fir3          /* Generate program with profiling turned on.*/
tmsim -nomm fir3                /* Simulate intermediate code and produce
                                dtprof.out.*/

tmcc -r fir3.c -o fir3          /* Recompile using profile information. */
tmsim -statfile fir3.stat fir3  /* Simulate and collect cycle accurate
                                information.*/

tmprof -scale 1 -func fir3.stat /* Output is sent to stdout.*/

```

The output of **tmprof** is as follows:

Funcname	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
-----	-----	-----	-----	-----
restrict_direct_convoluti	1	3051 42.79%	174	467
initialize	1	2867 40.21%	261	84
exit	1	193 2.71%	145	22
main	1	153 2.15%	136	3
_clear_all_regs	1	129 1.81%	116	0
_clear_all_regs_1	1	129 1.81%	116	0
_pre_start	1	120 1.68%	65	45
_profile_write	1	114 1.60%	92	11
_start_1	1	89 1.25%	67	15
_start_second	1	64 0.90%	58	0
_return_custom_begin	1	62 0.87%	58	0
_default_exit_2	1	43 0.60%	0	38
_default_exit_1	1	36 0.50%	29	3
_custom_begin	1	33 0.46%	29	0
_exit	1	33 0.46%	29	0
_default_exit	1	6 0.08%	0	0
_return_custom_end	1	4 0.06%	0	0
_custom_end	1	4 0.06%	0	0
-----	-----	-----	-----	-----
total/average		7130 100.00%	1375	688

Notice that the execution speed of the unrolled loop improves by about 50% when restricted pointers are used, compared to program fir2.c. This latest version of the loop executes about 4.5 times faster than the original version in program fir1.c.

As one more improvement, we use grafting along with unrolling and restricted pointers. The following commands are used for compiling program fir3.c.

```

tmcc -p fir3.c -o fir3          /* Generate program with profiling turned on.*/
tmsim -nomm fir3              /* Simulate intermediate code & produce
                               dtprof.out.*/

tmcc -G fir3.c -o fir3        /* Recompile using profile information, perform
                               grafting.*/

tmsim -statfile fir3.stat fir3 /* Simulate & collect cycle accurate information.*/
tmprof -scale 1 -func fir3.stat /* Output is sent to stdout.*/

```

Grafting again gains performance, mainly due to the effect on other functions. The text section of the object code has a size of 7552 bytes, which is about 27% more than the original code size.

Funcname	Executions	Total Cycles (%)	I\$Cycles	D\$Cycles
-----	-----	-----	-----	-----
restrict_direct_convoluti	1	2865 52.65%	576	476
initialize	1	1376 25.28%	493	84
exit	1	182 3.34%	145	11
main	1	153 2.81%	136	3
_clear_all_regs	1	129 2.37%	116	0
_clear_all_regs_1	1	129 2.37%	116	0
_pre_start	1	120 2.21%	65	45
_profile_write	1	114 2.09%	92	11
_start_1	1	89 1.64%	67	15
_start_second	1	64 1.18%	58	0
_return_custom_begin	1	62 1.14%	58	0
_default_exit_2	1	43 0.79%	0	38
_default_exit_1	1	36 0.66%	29	3
_custom_begin	1	33 0.61%	29	0
_exit	1	33 0.61%	29	0
_default_exit	1	6 0.11%	0	0
_return_custom_end	1	4 0.07%	0	0
_custom_end	1	4 0.07%	0	0
-----	-----	-----	-----	-----
total/average		5442 100.00%	2009	686

The pointers `str`, `avail`, and `symlist` in Figure 1-27 could have identical values or differing values. They could even overlap. Because of this, the compiler must order stores through pointers with respect to other loads and it stores in strict program order. There are 11 loads and 13 stores in the procedure `definesym`. Issuing these operations in strict program order limits the ILP.

The `str` points to an array of characters and `avail` points to a symbol table entry. It seems clear they differ given the types. `avail` points to a list of available nodes and `symlist` points to a list of nodes on the symbol table. These two sets should be independent. All but two (the uses of `symlist`) of the 24 memory references can be shown to differ. The other accesses can be performed in parallel.

Figure 1-29 shows how to modify the program to use restricted pointers. 360 instruction cycles are needed for `definesym`, as compared to 1830 before and 1440 after unrolling. This corresponds to 12 instructions per call to the function. Table 1-15 summarizes the performances with restricted pointers and loop unrolling and before optimization.

```

struct symbol *definesym(char * restrict str, int value) {
    struct symbol * restrict res = avail;
    avail = avail->next;
    res->name[0] = str[0]; res->name[1] = str[1];
    res->name[2] = str[2]; res->name[3] = str[3];
    res->name[4] = str[4]; res->name[5] = str[5];
    res->name[6] = str[6]; res->name[7] = str[7];
    res->value = value;
    res->next = symlist;
    symlist = res;
    return res;
}

```

Figure 1-29 Using Restricted Pointers

Table 1-15 Instruction Cycles Per Procedure Call

	Call To definesym
No Optimization	62
Loop Unrolling	40
Loop Unrolling + Restricted Pointers	12

Be aware that unwarranted use of restricted pointers can introduce subtle bugs.

The implementation of the `restrict` keyword is based on the paper *Restricted Pointers in C* by the Numerical C Extensions Group of ANSI X3J11. This paper (X3J11/94-019, Draft 2) can be found at <http://www.lysator.liu.se/c/restrict.html>. To the best of our knowledge, restricted pointers will be in the future ANSI C standard.

Using Custom Operators

The TriMedia hardware architecture provides special operations for DSP applications. They are made available through the `custom_op` declaration. In fact, all machine operations are available through the `custom_op` mechanism, but not all of them are of use to you. The most important ones are defined in the include file `custom_defs.h`¹. We recommend that you use only the custom operators defined in `custom_defs.h`. By using only these, you can develop and execute on the host platform because a special library with the implementation of the custom operators is provided.

1. The real declaration of the custom operators is done in include file `custom_ops.h`. The file `custom_defs.h` is an abstraction from the custom operators to enable you to develop and execute on the host platform with use of the TriMedia `custom_ops`.


```

void
custom_ops_direct_convolution(
    char * restrict a,
    char * restrict b,
    int * restrict c )
{
    int    i, ib0, ib1, i0, i1, i2;
    int    * restrict ia;

    ia = (int *) a;

    /*
     * Copy b, in a new array called rev_b in time reversed order.
     * ib points this array as an integer pointer.
     * Let A = |abcd| and B = |pqrs| with where a,b,c,d,p,q,r, and s are
     * all 8 bit integers. Then
     * PACKBYTES(A,B) = |ds| and
     * PACK16LSB(A,B) = |cdrs|
     */

    ib0 = PACK16LSB(PACKBYTES(b[7], b[6]), PACKBYTES(b[5], b[4]));
    ib1 = PACK16LSB(PACKBYTES(b[3], b[2]), PACKBYTES(b[1], b[0]));

    for(i = 0; i < NROF_SAMPLES/4; i++) {
        /*
         * Let A = |abcd| and B = |pqrs| where a,b,c,d,p,q,r, and s are
         * all 8 bit integers. Then
         * FUNSHIFT1(A,B) = |bcdp|
         * FUNSHIFT2(A,B) = |cdpq|
         * FUNSHIFT3(A,B) = |dpqr|
         * IFIR8II(A,B)   = a*p + b*q + c*r + d*s
         */

        i0 = ia[i - 2];
        i1 = ia[i - 1];
        i2 = ia[i];

        c[0] = IFIR8II(ib0, FUNSHIFT1(i0, i1)) +
              IFIR8II(ib1, FUNSHIFT1(i1, i2));
        .....
        c[3] = IFIR8II(ib0, i1) + IFIR8II(ib1, i2);
        c += 4;
    }
}

```

Figure 1-30 Convolution Example - Custom Operators

Of special interest for the example are the custom_ops FUNSHIFT and IFIR8II. Let $A = labcdl$ and $B = lpqrs$ where a, b, c, d, p, q, r, s , are all 8-bit integers. Then,

```
FUNSHIFT1(A,B) = |bcdp|
FUNSHIFT2(A,B) = |cdpq|
FUNSHIFT3(A,B) = |dpqr|
IFIR8II(A,B) = a*p + b*q + c*r + d*s
```

Four multiplications and three additions in the inner loop of program fir3.c are replaced by one FUNSHIFT and one IFIR8II operation. Other usage of custom operators in the fir program are selecting bytes or half words and merging them into one word (PACKBYTES and PACK16LSB). To use these custom operators, the program must include the header file custom_defs.h. You can find this include file in the directory \$TCS/include/ops. This directory is in the default include path for the compiler driver. Most custom operators directly map to hardware operations. Access via the include file custom_defs.h ensures that your program can still run on the host system, because a library of custom operator implementation for the host system is provided.

Figure 1-30 shows the modified function, still using restricted pointers. Compiling the program fir4.c while grafting using the following commands and running **tmprof** shows the performance gain due to custom operators:

```
tmcc -p fir4.c -o fir4          /* Generate program with profiling turned on. */
tmsim -nomm fir4              /* Simulate intermediate code and produce
                               dtprof.out. */
tmcc -G fir4.c -o fir4        /* Recompile using profile information, perform
                               grafting */
tmsim -statfile fir4.stat fir4 /* Simulate and collect cycle accurate
                               information.*/
tmprof -scale 1 -func fir4.stat /* Output is sent to stdout.*/
```

The output of **tmprof** shows

Funcname	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
total/average		4965	100.00%	1972	1260
-----	-----	-----	-----	-----	-----
custom_ops_direct_convolu	1	1581	37.58%	591	211
initialize	1	1376	32.71%	493	84
exit	1	193	4.59%	145	22
main	1	153	3.64%	136	3
_clear_all_regs_1	1	129	3.07%	116	0
_clear_all_regs	1	129	3.07%	116	0
_pre_start	1	120	2.85%	65	45
_profile_write	1	114	2.71%	92	11
_start_1	1	89	2.12%	67	15
_start_second	1	64	1.52%	58	0
_return_custom_begin	1	62	1.47%	58	0
_default_exit_2	1	43	1.02%	0	38
_start	1	38	0.90%	29	5
_default_exit_1	1	36	0.86%	29	3
_custom_begin	1	33	0.78%	29	0
_exit	1	33	0.78%	29	0
_default_exit	1	6	0.14%	0	0
_return_custom_end	1	4	0.10%	0	0
_custom_end	1	4	0.10%	0	0
-----	-----	-----	-----	-----	-----
total/average		4207	100.00%	2053	437

The unrolled version of the loop now is already 7.8 times as fast as the original version, because of the use of custom operators, grafting, and restricted pointers.

Using the Global Optimizer

You can enable the global optimizer feature of the TCS to gain performance improvements. A general observation is that you should use the global optimizer with care when the program is already hand-optimized. On some functions there can be an improvement, on others a degradation. This is due to the nature of the parallel architecture. The global optimizer hoists computations out of loops, which means they become dtree global computations, that is, the values are produced in one and used in another dtree. These values end up in global registers because they have to be preserved across dtree jumps. This is in contrast to the dtree local variables used by the scheduler, which are only alive within the decision tree.

Use of global registers adds save-and-restore code to the program. If there is not much parallelism in the loop, the hoisted expression could have been recalculated in the loop at no extra cost. The current global optimizer does not know about this trade-off because it

can only be made after scheduling. For this reason, you can use pragmas to turn the global optimizer on and off selectively per function.

The main opportunities for the global optimizer in `fir4.c` are the reuse of the values `a[i-1]`, `a[i]` and `FUNSHIFTx(i1,i2)` from the previous loop iteration. The global optimizer is enabled using flag `-O3`.

The following sequence of commands is used to recompile `fir4.c`:

```
tmcc -p fir4.c -o fir4          /* Generate program with profiling turned on.*/
tmsim -nomm fir4              /* Simulate intermediate code & produce
                             dtprof.outtmcc -G -O3 -tmccom
                             -graft_tuning_file graftfile -- fir4.c -o
                             fir4 recompile using profile information,
                             perform grafting, and use the global
                             optimizer.*/
tmsim -statfile fir4.stat fir4 /* Simulate & collect cycle accurate
                             information. */
tmprof -scale 1 -func fir4.stat /* Output is sent to stdout.*/
```

Funcname	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
initialize	1	1516 39.70%	206	274
custom_ops_direct_convolu	1	1053 27.57%	259	131
exit	1	193 5.05%	145	22
main	1	153 4.01%	136	3
_clear_all_regs_1	1	129 3.38%	116	0
_clear_all_regs	1	129 3.38%	116	0
_pre_start	1	120 3.14%	65	45
_profile_write	1	114 2.99%	92	11
_start_1	1	89 2.33%	67	15
_start_second	1	64 1.68%	58	0
_return_custom_begin	1	62 1.62%	58	0
_default_exit_2	1	43 1.13%	0	38
_start	1	38 1.00%	29	5
_default_exit_1	1	36 0.94%	29	3
_custom_begin	1	33 0.86%	29	0
_exit	1	33 0.86%	29	0
_default_exit	1	6 0.16%	0	0
_return_custom_end	1	4 0.10%	0	0
_custom_end	1	4 0.10%	0	0

total/average		3819 100.00%	1434	547

Figure 1-31 shows a C program that sorts a table using the insertion sort algorithm. The procedure `insertion` is derived from one in the book “Algorithms in C” by Robert Sedgewick (published by Addison-Wesley). Before sorting, the table is in reverse order.

```

1  #include <stdio.h>
2  #define SIZE100
3
4  void insertion(int *a, int n)
5  {
6      int i, j, v;
7
8      for (i=1; i<n; i++) {
9          v = a[i];
10         for (j = i; j>0 && a[j-1] > v; j--)
11             a[j] = a[j-1];
12         a[j] = v;
13     }
14 }
15
16 main()
17 {
18     int a[SIZE], i;
19     for (i = 0; i < SIZE; i++)
20         a[i] = SIZE-i;
21     insertion(a, SIZE);
22 }

```

Figure 1-31 Insertion Sort

The program is compiled and executed as follows (user input is in italics):

```

$ tmcc insertion_sort.c
  $ tmsim -v -nomm ins1
  tmsim v19.5 of 1.0f1SunOS (Jul 10 1996 14:37:55)
  ----- starting batch execution -----
  ----- end of batch execution -----
  nr of cycles:                71240
$

```

The first command compiles the program using the TriMedia compiler driver **tmcc** with the default optimization level two (decision-tree optimization only). The output is placed in the file `a.out`. Local optimization is performed by the TriMedia core compiler, **tmccom**. The optimizations are copy propagation, common subexpression elimination, constant folding, and load elimination. They are local to a decision tree (the scheduling unit on TriMedia). Decision trees are defined in Chapter 2 of “Programming Languages and File Formats.” The local optimizer implements alias analysis and orders the scheduling of loads and stores.

The second command runs the program using the Trimedia machine-level simulator **tmsim**. The `-v` option tells the simulator to give statistics on the execution time. The release version and build date is reported by `-v`. Both instruction execution and the caches of TM-1000 are simulated by default by **tmsim**. The `-nomm` option tells **tmsim** to only count the number of instructions (cache is not simulated). The 71240 instructions are necessary to sort a 100-element table.

The program is compiled and run using the global optimizer, as follows:

```
$ tmcc -O3 insertion_sort.c
$ tmsim -v -nomm a.out
tmsim v19.5 of 1.0f1SunOS (Jul 10 1996 14:37:55)
----- starting batch execution -----
----- end of batch execution -----
nr of cycles:           51335
$
```

tmccom has a global optimizer that performs the same optimizations as the local optimizer over the whole procedure. It is invoked from **tmcc** using the `-O3` option.

51335 instructions are necessary to execute the program. This corresponds to 513.4 microseconds on a 100-Mhz TM-1000, not counting cache. The execution times without and with global optimization are compared in Table 1-16.

Table 1-16 Execution Times

Time to Sort a 100 Element Table (μsec)	
Without Global Optimizer	With Global Optimizer
712.4	513.4

Understanding a difference in performance requires analyzing the program's time behavior. To find out where the program spends most of its time, first generate two executable files, `ins.O2` and `ins.O3`, with and without global optimization.

```
$ tmcc -o ins.O2 -O2 insertion_sort.c
$ tmcc -o ins.O3 -O3 insertion_sort.c
```

Obtain the statistics on the behavior of individual decision trees by using **tmsim** with the `-statfile` option:

```
$ tmsim -statfile ins.O2.stat ins.O2
$ tmsim -statfile ins.O3.stat ins.O3
```

Extract a function-level profile of the behavior with local optimization by using **tmprof**:

```
$ tprof -scale 1 -func ins.O2.stat
```

Funcname	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
<u>insertion</u>	1	<u>70671</u>	97.47%	174	0
main	1	750	1.03%	104	31
exit	1	184	0.25%	145	13
_clear_all_regs	1	129	0.18%	116	0
_clear_all_regs_1	1	129	0.18%	116	0
_pre_start	1	120	0.17%	65	45
_profile_write	1	115	0.16%	90	14
_start_1	1	89	0.12%	67	15
_start_second	1	64	0.09%	58	0
_return_custom_begin	1	62	0.09%	58	0
_default_exit_2	1	43	0.06%	0	38
_default_exit_1	1	36	0.05%	29	3
_exit	1	34	0.05%	30	0
_custom_begin	1	33	0.05%	29	0
_start	1	33	0.05%	29	0

total/average		72506	100.00%	1110	159

The `-scale` option sets the scale to a clock cycle (the default scale is 1000 cycles). The `-func` option produces a report in which each line corresponds to a function (by default the report is by decision tree).

The overall behavior is summarized in the final line. 72506 cycles are necessary to execute the program. The difference between this figure and that reported by `tmsim` (`-nomm` option) corresponds to the cache overhead, as reported in the last two columns.

The output of `tprof` is in decreasing execution time order. 97% of the execution time of the program is spent in the procedure `insertion`. A breakdown by decision tree of the time spent there can be obtained as follows:

```
$ tprof.select insertion ins.O2.stat -scale 1
```

Treename	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
<u>insertion_DT_2</u>	4950	<u>68566</u>	97.02%	58	0
insertion_DT_1	99	1019	1.44%	29	0
insertion_DT_4	99	594	0.84%	0	0
insertion_DT_3	99	425	0.60%	29	0
insertion	1	34	0.05%	29	0
insertion_DT_5	1	33	0.05%	29	0

total/average		70671	100.00%	174	0
(...)					

The where `tmprof.select` function is a simple shell script that applies `tmprof` to a particular function. The breakdown of time in the function with global optimization can be obtained similarly:

```
$ tmprof.select insertion -scale 1 ins.O3.stat
```

Treename	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
<u>insertion_DT_2</u>	4950	<u>49144</u>	96.81%	29	11
insertion_DT_1	99	920	1.81%	29	0
insertion_DT_3	99	633	1.25%	39	0
insertion	1	34	0.07%	29	0
insertion_DT_4	1	33	0.07%	29	0
total/average		50764	100.00%	155	11

The difference in execution time can be seen to come from decision tree number two of the procedure `insertion`. In what follows, the notation `insertion2` is used for decision tree number two of `insertion`. This corresponds to the inner loop of the algorithm and takes about ten (49144/4950) cycles per iteration. Execution times for the inner loop without and with global optimization are shown in Table 1-17. The improvement in the performance is due to better alias analysis using the global optimizer.

Table 1-17 Execution Times

Clock Cycles for Inner Loop	
Without Global Optimizer	With Global Optimizer
13.85	9.92

The function `tmprof` does not report all the information about the execution behavior contained in the statistics file. The line of the statistics file corresponding to `insertion2` with global optimization can be obtained as follows:

```
$ select insertion_DT_2 ins.O3.stat
```

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopers
exopers							
<u>insertion_DT_2</u>	4950	49104	29	11	0	0	
<u>79002</u>	<u>79002</u>						

The `select` function is another simple shell script to select the lines corresponding to a function. The TM-1000 processor can execute up to five operations per instruction cycle. The last two fields correspond to the number of issued operations and executed operations. These can vary due to speculation and guarding. You can find information on these in the *TM-1000 Data Book*.

The second and third fields correspond to the number of times the decision tree is executed (`execs`) and the number of cycles not including cache overhead (`instc`). The other fields give a breakdown of cache overhead.

Roughly ten instructions on average are executed per iteration of the loop (=49104/4950). About 16 operations on average are executed per iteration (=79002/4950). The average Instruction Level Parallelism (ILP) is 1.6 operations per instruction, which is fairly low compared to the five operations per cycle available.

Jumps and memory references limit ILP. The TM-1000 processor has more processing power, but it needs the same balance of processor and memory as a conventional processor. The ILP of a decision tree can also be limited by the length of the critical path.

A summary of the memory behavior can be obtained using `tmsim`:

```
$ tmsim -v ins.03
          (...)
nr of cycles:          52573
nr of executed instructions: 51332
CPI:                  1.024

instruction cache statistics: size 32 kB, blocksize 64 b, associativity
8
nr of accesses:      51333
nr of hits:          51297
hitrate:             100 %
CPI:                 0.022
data cache statistics: size 16 kB, blocksize 64 b, associativity 8
(hierarchical lru)
nr of accesses:      15279
nr of hits:          15265
hitrate:             100 %
dconflictrate:       0 %
CPI:                 0.003
I/D overlap CPI:    0.001
```

The ratio of instructions to data accesses is 0.29 (=51333/15279) on average. This is high for TM-1000. When a significant percentage of the time spent in a program is in a decision tree with low ILP, it is a good idea to look at the generated code. The assembler output corresponding to `insertion2` is shown below. It has been condensed for layout purposes.

```

__insertion_DT_2:
(* cycle 0 *)
    asli(0x2) r34 -> r124 , isubi(0x1) r34 -> r125 ,
    uimm(__insertion_DT_3) -> r126 ,
    uimm(__insertion_DT_2) -> r127 , nop ;

(* cycle 1 *)
    asli(0x2) r125 -> r125 , isubi(0x4) r124 -> r122 , igtri(0) r125 -> r123 ,
    iadd r35 r124 -> r124 , iadd r0 r125 -> r34 ;

(* cycle 2 *)
    isubi(0x4) r125 -> r125 , ijmpf r123 r126 ,nop ,
    ld32r r35 r122 -> r122 ,nop ;

(* cycle 3 *)
    nop , nop , nop , nop , nop ;

(* cycle 4 *)
    nop , nop , nop , nop , nop ;

(* cycle 5 *)
    nop , nop , nop , h_st32d(0) r122 r124 , nop ;

(* cycle 6 *)
    nop , nop , nop , ld32r r35 r125 -> r125 , nop ;

(* cycle 7 *)
    nop , nop , nop , nop , nop ;

(* cycle 8 *)
    nop , nop , nop , nop , nop ;

(* cycle 9 *)
    igtr r125 r33 -> r125 , nop , nop , nop , nop ;

(* cycle 10 *)
    nop , IF r123 ijmpf r125 r126 , IF r123 ijmptr125 r127 , nop , nop ;

(* cycle 11 *)
    nop , nop , nop , nop , nop ;

(* cycle 12 *)
    nop , nop , nop , nop , nop ;

(* cycle 13 *)
    nop , nop , nop , nop , nop ;

```

There are two unused cycles after the load instructions (ld32r) in cycles two and seven. The three cycles after the jump operations (jmptr, jmpf) in cycle ten are wasted. When

unused cycles are observed in a section of code like this, it is a good idea to use profiling and grafting.

Using Profiling and Grafting

The Trimedia compiler can generate more parallel code by being *trained* about the behavior of the program. First, it is necessary to compile the program with the `-p` (profile option):

```
$ tmcc -p insertion_sort.c
```

This tells the compiler to add code to generate statistics to the program. Running it using `tmsim` produces a file `dtprof.out` containing more decision tree information. It can be read using `tmdtprof`:

```
$ tmdtprof dtprof.out
insertion_sort.c:main()          calls = 1      operations = 1241
insertion_sort.c:insertion()    calls = 1      operations = 76345
Function count = 2
path: insertion_sort.c main() 16/1 (dtree count = 5)
      (...)
path: insertion_sort.c insertion() 4/6 (dtree count = 7)
      dt(0)4/1 ops(11) exits(2)
          0 -> dt(1) exec count(1)
          1 -> dt(1) exec count(0)
          (...)
      dt(3)11/17 ops(15) exits(3)
          0 -> dt(3) exec count(4851)
          1 -> dt(4) exec count(0)
          2 -> dt(4) exec count(99)
      (...)

```

The underlined information gives the correspondence between a decision tree and the file and line in the source program. For example, the procedure `insertion` starts at column six of line four of “`insertion_sort.c`.” The `insertion3` function begins at line nine, column six (the for condition of Figure 1-31). The `->` lines are the exit paths (jumps) out of the decision tree. For example, on path zero, `insertion3` loops to itself 4,851 times. On the first path it goes 99 times to `insertion4`, and the second path is never taken.

The compiler implements an optimization called grafting to reduce jump latency and increase ILP. Grafting copies a decision tree at the place of a jump. The compiler selects the most frequently executed decision trees as candidates for grafting from the information in `dtprof.out`. Grafting takes place when there is a high frequency and probability of flow

along a path. For example, it is worthwhile to graft `insertion3` onto itself several times because the loop is frequently reentered.

You can compile and run the `-G` option tells the compiler to use grafting. The program can be compiled and run with grafting as follows:

```
$ tmcc -G -O3 -O ins.O3.graft insertion_sort.c
$ tmsim -v -statfile ins.O3.graft.stat ins.O3.graft
      (... )
nr of cycles:          20866
nr of executed instructions: 19142
CPI:                  1.090

instruction cache statistics: size 32 kB, blocksize 64 b, associativity 8
nr of accesses:      19143
nr of hits:          19091
hitrate:             100 %
CPI:                 0.083
data cache statistics: size 16 kB, blocksize 64 b, associativity 8
nr of accesses:      10388
nr of hits:          10373
      (... )
$ select insertion_DT_2 ins.O3.graft.stat
tree name          execs   instc   installc   dstallc   cpbacks   cnflctc   isopers
exopers
-----
__insertion_DT_2  1128   16638    87         0         0         0
56068   55718
  __clearregs_DT_ 1    32     192     58     16     0     6
512      511
  __clearregs_DT_ 1    32     928     58     24     0     1
832      831
```

Table 1-18 compares the inner loop behavior at `-O3` with and without grafting. 78% of the branches (1128 versus 4950) are eliminated due to grafting (the loop is grafted four times). The ILP increases from 1.60 to 3.36. The issued and executed operations correspond to the `isopers` and `exopers` fields of the statfile (see above).

Table 1-18 Inner Loop Behavior at `-O3` with and without Grafting

	Executions of Decision Tree 2	Instruction Cycles	Issued Operations	Executed Operations	Ops/Inst'n (ILP)
No Grafting	4950	49104	79002	79002	1.60
Grafting	1128	16638	56068	55718	3.36

Table 1-19 compares the overall behavior. The number of memory accesses is about the same with and without global optimization. However, there is a reduction of one third when using both global optimization and grafting.

Table 1-19 Overall Behavior At Different Levels of Optimization

Optimizations	Instructions	Memory Accesses	Total Cycles
Local Optimization	71237	15180	72534
Global Optimization	51332	15279	52573
Grafting and Global Optimization	19142	10373	20855

Using Unsafe Alias Analysis

The earlier section “Using Restricted Pointers” on page 1-42 describes how you can use restricted pointers to help the compiler in alias analysis¹. Good alias analysis is one of the key techniques for obtaining parallelism and the best optimization. The alias analyzer weakens the ordering of memory operations (all assignments and uses of values in C terms). A weaker ordering allows more operations to go in parallel. In C, it is important not to use pointers unless it is necessary because an unknown pointer aliases with all nonlocal nonexposed variables². Also, the use of global variables limits the abilities of the alias analyzer to disambiguate two memory locations.

The compiler currently has three levels of alias analysis. Level zero is perfectly safe, that is, no assumptions are made on any use of the ANSI C language. The two higher levels do make assumptions on the use of the language and are safe in most programs. *However, when using unsafe alias analysis, it is very important to understand the details of the program and the use of all memory references.*

You can specify unsafe alias analysis with the option `-A[012]` to the compiler. The default level is level one. You can use the pragmas to change the alias level function per function.

-
1. Alias analysis is the technique used in the compiler to determine whether two memory locations are the same or whether they overlap.
 2. Local variables are variables declared within a function scope. Nonexposed variables are variables of which the address is never taken. The compiler knows that if the address is never taken that it cannot be stored to any pointer variable and, thus, does not alias with any pointer indirection. In the absence of inter-procedural analysis, the nonexposed property can only be determined for local variables.

Level one makes the following two assumptions:

1. The memory object referred to by a certain pointer does not contain the same pointer. This means that when `p` points to a struct, it is assumed that there is no field `f` in the struct such that `p->f==p`. Also, when `p` points to an array of pointers, it is assumed that there is no index `i` such that `p[i]==p`. Similar assumptions are made for arbitrary nesting of arrays and structs within the memory objects that `p` refers to. See Figure 1-32, for example.
2. It is assumed that it is senseless to address outside any variable, and that it is impossible to 'reach' a variable through a pointer that does not already 'point' somewhere in the same variable. This means that no assumptions are made on the relative positions at which the variables are mapped in memory, and that no attempt is made during execution to determine these relative positions.

These assumptions are used by the alias analyzer when trying to determine possible aliasing in case of a memory reference through a pointer (with a certain offset) and to a variable: if, given that the pointer points 'somewhere' in the variable, the memory reference via the pointer and with the given offset would result in addressing (partially) outside that variable, no aliasing is assumed. In mathematical terms: if `p` is a pointer and `a` is a variable the memory region `[*p + sizeof(a), *p + sizeof(*p)]` does not alias with `a`. See Figure 1-33.

Level two assumes everything from level one and globals are modified only by accessing the global itself; that is, the address of a global is never taken. See Figure 1-34 for an example.

Note

Level two is only implemented with global optimizations on, that is, optimization level three (`-O3`). ♦

```
int *p, *q;
program fragment:
    *p = 3;
    q = p;
```

At unsafe alias analysis level `-A1`, we assume that `*p != p`, so the value of `p` does not have to be reloaded after the assignment `*p = 3`.

Figure 1-32 Example of Point One for Unsafe Alias Analysis Level One

```
typedef struct some_type{
    char    c;
    short   s;
    int     i;
} ;
int a;
struct some_type *p;
```

Figure 1-33 Example of Point Two for Unsafe Alias Level One

At any program point, $p \rightarrow c$ and $p \rightarrow s$ will alias with a (for instance when we initialize $p = (\text{struct some_type}) \&a$), but $p \rightarrow i$ will not alias with any load or store to a at unsafe alias -A1. This is because “ a ” can not be packed into a larger object (like $*p$). However, due to the casting possibility in C, the compiler still lets both objects alias at the start (or at the address) of the object.

Note that the assumption that $p \rightarrow i$ and a do not alias is not valid when we initialize $p = (\text{some_type } *) (\&a - 1)$. But then a store to $*b$ would arbitrarily overwrite something in memory. Program constructs like this do occur, but rarely.

```
int *p;
int a;
```

Figure 1-34 Example of Point Three for Unsafe Alias Level Two

a and $*p$ will not alias at unsafe alias level -A2 (in -O3) because the compiler assumes that the address of a is never taken and can thus never have been assigned to p .

Figure 1-35 shows a program that initializes four arrays. Decision tree statistics for the loop at -O3 are shown.

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopers	exopers
__clearregs_DT_1	32	<u>928</u>	58	24	0	1	<u>832</u>	831

```

#define NREG 32
typedef struct rtl *rtl;
int *qty_first_reg, *qty_last_reg, max_qty = NREG;
rtl *qty_const, *qty_const_insn;
main()
{
    qty_first_reg = (int *)malloc(NREG * sizeof(int));
    qty_last_reg = (int *)malloc(NREG * sizeof(int));
    qty_const = (rtl *) malloc(NREG * sizeof(rtl));
    qty_const_insn = (rtl *) malloc(NREG * sizeof(rtl));
    clearregs();
}

clearregs()
{
    int i;
    for (i = 0; i < max_qty; i++) {
        qty_first_reg[i] = i;
        qty_last_reg[i] = i;
        qty_const[i] = 0;
        qty_const_insn[i] = 0;
    }
}

```

Figure 1-35 Initializing Four Arrays

The loop is executed 32 times and there are four stores per iteration. This corresponds to 128 memory accesses. The compiler does not know whether the addresses of the external variables (`max_qty`, `qty_first_reg`, `qty_last_reg`, `qty_const`, `qty_const_insn`) have been assigned to a pointer. For example, the loop is executed 32 times. However, `qty_first_reg` could point to `max_qty` at the start of the loop. If this is true, the loop should only be executed once. Five load accesses are necessary per loop iteration because of the pointer aliasing. 160 memory accesses are added to the program.

The `-A2` option of `tmcc` relaxes the rules for alias analysis. The compiler can assume that the accesses to `extern` and `static` variables do not alias with stores through pointers. The option `-O3` must be specified if the `-A2` option is to have effect. The decision tree statistics with `-A2` are shown below:

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopersexopers
-----	-----	-----	-----	-----	-----	-----	-----
__clearregs_DT_1	32	<u>192</u>	58	16	0	6	<u>512</u> 511

Table 1-20 compares the performance with and without relaxed aliasing. Six instruction cycles (=192/32) are necessary per loop iteration with the -A2 option. 29 instruction cycles (=928/32) are necessary without the -A2 option. There are five variables and a load of each requires two operations. This corresponds to the ten (=5x2) fewer operations in the loop using relaxed aliasing. Only 272 cycles, as opposed to 1011 including cache overhead, are necessary with relaxed aliasing.

Table 1-20 Performance With And Without Relaxed Aliasing

	Instruction Cycles Per Loop	Operations	Memory Accesses	ILP	Total Cycles
Without Relaxed Aliasing (no -A2 option)	6	16	128	2.66	1011
With Relaxed Aliasing (-A2 option)	29	26	288	0.89	272

Take care when using relaxed aliasing. Do not use it if a global variable's address is taken. Figure 1-36 shows how to use relaxed alias analysis for an individual procedure or function.

```
clearregs()
{
    int i;
    #pragma TCS_A2
    for (i = 0; i < max_qty; i++) {
        qty_first_reg[i] = i;
        qty_last_reg[i] = i;
        qty_const[i] = 0;
        qty_const_insn[i] = 0;
    }
}
```

Figure 1-36 Locally Relaxed Aliasing

Using a Dirty Float

Usually compiler optimizations on floating point expressions are illegal. This is because all commutative and associative properties that hold for integer operations like addition and multiply do not hold for floating point operations. You can give the compiler more freedom in expression optimizations and program transformations by using the `dirty_float` option.

You can give the option on the command line with `dirty_float <nn>` or with pragma's `TCS_dirty_float0, ..., TCS_dirty_float2`. There are three levels with the following meaning:

- At level zero there are no optimizations performed on floating point expressions.
- At level one the compiler folds constant floating point expressions and introduces conversions for `if` statements containing floating point expressions. Expressions remain ordered against read and write to PCSW
- At level two, besides the operations performed at level one, the compiler performs tree height reduction and reordering of floating point expressions to increase parallelism. Also, otherwise illegal optimizations like rewriting the expression `d != d` (check for NaN) to false are performed.

Note

This option only has an effect at optimization level three, and *might* cause incorrect results. ♦

Using Cache Optimization

Several of the techniques discussed in the preceding sections, including the use of grafting, loop unrolling, and inlining, result in an increase in the size of the program code, which in turn, increases the number of instruction cache stalls. You must pay attention to the code size because the I-cache stalls can become an important factor. This section addresses techniques to enhance data cache utilization, thereby improving the overall program performance.

Vary the Right-Most Array Index in the Inner Loop

The program on the left of Figure 1-37 zeroes a byte array. **tmprof** output from running it is shown below:

Treename	Executions	Total Cycles	(%)	I\$ Cycles	D\$ Cycles
_____	_____	_____	_____	_____	_____
__main_DT_2	19200	734120	99.79%	29	618827
__main_DT_1	64	256	0.03%	0	0
_exit	1	178	0.02%	132	37
	(...)				
total/average		735686	100.00%	1022	619015

Most of the execution time (619015 out of 735686 cycles) is lost in data cache stalls. Almost all the stall cycles are in the decision tree `main2`. This corresponds to the inner for loop. It is executed 19200 (64 x 300) times.

The stalls in the program of Figure 1-37 are data cache write miss stalls. Figure 1-38 shows code that you can use to instrument the program. The list of events is in Chapter 3 of the *TM-1000 Data Book*. You must add two lines before the first for to generate and count data cache write missed events:

```
_MMIO_base[MEMORY_EVENTS>>2] = MM_WRTMISS;
monitor(TMR_CACHE1);
```

You also need to add instrumentation code after execution of the relevant section of the program:

```
printf("cache misses = %d\n", events());
```

The instrumentation reports 19200 data cache write misses. There is a cache miss for every access.

In C, the rightmost subscript of a multidimensional array varies fastest as elements are accessed in storage order. Each execution of the inner loop clears a single byte. Consecutive accesses by the program of Figure 1-37 are spaced by 64 bytes. This is the size of a cache line. Each cache miss corresponds to 64 bytes, only one of which is used.

The program on the right is equivalent to the program on the left, with the order of the for loops interchanged. 124320 cycles are necessary to execute the program on the right. Only 4314 cycles are lost in data cache stalls. Instrumentation allows the number of data cache stalls in the program on the right to be measured also. Measurement shows 300 data cache stalls. 300 data cache lines correspond to 19200 (=300 x 64) bytes of data. This corresponds to all the data in the array. Each miss costs 14.38 (=4314/300) cycles, compared to 32.24 cycles previously. This is because the misses can be overlapped with the execution of the program. The program on the left of Figure 1-37 generates too many stalls, so no overlap is possible after the first miss. Table 1-21 summarizes the effects of interchanging the loops.

```
#include <stdio.h>
#include "tml/mmio.h"

chara[300][64];

main()
{
    int k, l;

    for (l=0; l<64; l++)
        for (k=0; k<300; k++)
            a[k][l] = 0;
}
```

```
#include <stdio.h>
#include "tml/mmio.h"

chara[300][64];

main()
{
    int k, l;
    for (k=0; k<300; k++)
        for (l=0; l<64; l++)
            a[k][l] = 0;
}
```

Figure 1-37 Loop Interchange

```

#include "tml/mmio.h"

#define TMR_RUN    1
#define TMR_CACHE1 6
#define MM_WRTMISS 4
struct timer { int modulus, value, ctl; };
#define TIMER ((struct timer *)((char *)_MMIO_base + TIMER1))
#define events() (TIMER->value)

void monitor(int event) {
    struct timer * tp = TIMER;
    tp->ctl = 0; tp->value = 0;
    tp->modulus = -1; tp->ctl = event << 8 | TMR_RUN;
}

```

Figure 1-38 Instrumentation Code**Table 1-21** Data Cache Write Misses Clearing a 64 x 300 Array of Characters

	Write Misses	Write Miss Stall Cycles
First Index Varies in Inner Loop	19200	619015
Second Index Varies in Inner Loop	300	4314

Pack Data as Tightly as Possible

Figure 1-39 shows a procedure to look up the name of the city closest to a point. A city is represented by a data structure containing the x and y coordinates (2 x 4 bytes) and the name of the city (64 bytes).

```

#define distance(x1, y1, x2, y2) ((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1))
#define NCITY 256
struct city { int x; int y; char name[64]; } cities[NCITY];
char * closest(int x, int y) {
    int max, dist, here = 0, i; CITY *ap;
    max = dist(x, y, cities->x, cities->y);
    for (i = 1, ap = &cities[1]; ap<&cities[NCITY]; ap++, i++) {
        dist = distance(x, y, ap->x, ap->y);
        if (max > dist) { max = dist; here = i; }
    }
    return cities[here].name;
}

```

Figure 1-39 Linear Search

The distance needs to be computed from each city. The key fields (x, y) of the data structure are referenced 256 times for each call. The name of the city is consecutive with those in memory. The cache brings both into memory. However, the name is referenced only once at the end of the procedure. Each access during the search accesses 64 bytes, only eight ($=2 \times 4$) of which are used. 6251 cycles are necessary in the procedure, of which 3365 correspond to data cache miss stalls.

In Figure 1-40, the data structure has been modified so that the fields not accessed during the search are stored apart. The key fields (x, y) have also been packed into shorts (2×2 bytes). The 3571 cycles are necessary after data restructuring. The 656 cycles are data cache stalls. Table 1-22 summarizes the effect.

Table 1-22 Data Cache Performance for a 256-Element Linear Search

	Write Miss Stall Cycles
Key and Value Stored Together (8 Bytes of Key)	3571
Key and Value Stored Separately (4 Bytes of Key)	656

```
struct city { short x; short y; } cities[NCITY];
char city_names[NCITY][64];
```

Figure 1-40 Modified Data Structure

Trade CPU Cycles for Cache Cycles

Figure 1-41 shows two programs to calculate the sieve of Eratosthenes. The program on the left represents the sieve by an array of bytes. The program on the right represents the sieve by a bit vector. Using a bit vector saves space but requires more operations to set and test an element.

Table 1-23 compares performance for calculating the 6542 primes between one and 65536. The figures shown correspond to the number of instruction cycles and cache stall cycles for the inner loop. Even though the program on the right is more complex, the number of instructions is identical. This is because there is spare processing power available. The store to sieve in the program on the right is more complex, but it can be executed partially in parallel. At the end of most decision trees there are available slots. Part of the store also executes in these slots.

Only 8K bytes are necessary to represent 65536 primes using a bit vector. Represented this way, the sieve fits in the cache. Represented as an array of bytes, it does not fit in the cache. This explains the difference in performance. Five opportunities are lost to issue

operations for every stall cycle. It is worth increasing the number of instructions if the working set fits in the cache as a result.

Table 1-23 Inner Loop of Sieve of Erasosthenes (Primes from one to 65536)

	Instruction Cycles	Data Cache Cycles
Sieve Represented as a Byte Vector	992975	1722483
Sieve Represented as a Bit Vector	992975	0

```
#define MAXPRIME 1000000
charsieve[MAXPRIME+1];
main(int argc, char *argv[])
{
    int i, j, sum, maxprime;

    maxprime = atoi(argv[1]);
    for (i=2; i<=maxprime; i++)
        sieve[i] = 1;
    sieve[0] = sieve[1] = 0;
    for (i=2; i <= maxprime>>1; i++){
        if (sieve[i]) {
            for (j=2*i; j<=maxprime; j+=i){
                sieve[j] = 0;
            }
        }
    }
}
```

```
#define MAXPRIME 1000000
charsieve[(MAXPRIME+7)/8];
main(int argc, char *argv[])
{
    int i, j, sum, maxprime;

    maxprime = atoi(argv[1]);
    for (i=0; i<=(maxprime+7)/8; i++)
        sieve[i] = -1;
    sieve[0] &= ~3; /* 0 and 1 aren't
prime */
    for (i=2; i <= maxprime>>1; i++) {
        if ((sieve[i>>3] >> (i&7)) & 1) {
            for (j=2*i; j<=maxprime; j+=i) {
                sieve[j>>3] &= ~(1 << (j&7));
            }
        }
    }
}
```

Figure 1-41 Sieve of Erasosthenes

Watch for Cache Set Hotspots

Figure 1-42 shows a procedure that sums up a column of an $n \times 128$ -element matrix. Performance figures for different values of n are given in Table 1-24. They correspond to 16 consecutive columns.

```

nt colsum(int col, int step) {
    int i, sum = 0;
    int *pcol;
    pcol = &matrix[0][col];
    for (i=0; i<128; i++) {
        sum += *pcol;
        pcol += step;
    }
    return sum;
}

```

Figure 1-42 Column of $n \times 128$ -Element Matrix

Table 1-24 Performance Figure for Values of N

Matrix Dimensions	Stride	Data Cache Stall Cycles	% Total Cycles
128 x 64	256	22752	62
128 x 65	260	3048	18
128 x 80	320	1574	10
128 x 512	2048	22679	62
128x 513	2052	23868	63
128 x 1040	4160	1594	10

The percentage of data cache stall cycles varies depending on the row dimension in a ratio from one to fifteen. Accesses to an array in row order, address consecutive bytes. In column order the accesses are separated by a *stride* equal to the size of the element multiplied by the row length. For example, for a 256 x 1040 array of integers, column accesses are separated by 4160 bytes.

There are 256 lines in the cache of 64 bytes each. These are organized into 32 sets capable of holding eight elements each. The set number is given by address bits six through eleven. The byte offset inside a line is given by bits zero to five.

The accesses in Table 1-24 are separated by a stride of more than 64 bytes. Each references a different line. The contents are reused only after an entire column has been traversed. Satisfactory performance for this program requires that 128 lines be held in the cache.

For a 256 x 64 matrix, imagine that the first access hits a particular set (say 29). The stride is an exact multiple of the line size ($256 = 4 \times 64$). The next access hits the set number + 4 modulo 32 (say 1). After referencing eight elements the accesses wrap around to set 29. The 128 accesses only use 64 (8×8) of the 256 lines of the cache. The working set of the program is 128 lines. This explains the poor performance. The performance is the same regardless of the starting set.

For a 256 x 1040 matrix, imagine that the first access also hits set 29. The stride is also an exact multiple of the line size. ($4160 = 65 \times 64$). The next access hits set 30 ($94 \text{ modulo } 32$). The next access hits set 31. The 128 accesses can fully use the cache. Again, the performance does not depend on the number of the first set.

For a 256 x 65 matrix, the stride (260) is not an exact multiple of 64 bytes. Accesses are made to set numbers separated by four ($260/64$). However, every 16 accesses ($64 / (260 \text{ mod } 64)$), the set number is also incremented. This allows the 128 accesses to be distributed among all the sets.

For a 256 x 513 matrix, every 16 accesses the set number is also incremented by one ($260 \text{ mod } 64 = 2052 \text{ mod } 64$). However, accesses are made to set numbers separated by 32 ($2052/64$), so the 16 accesses all hit the same set. The 128 accesses are distributed among only eight of the 256 lines of the cache.

```
float a[96][96], b[96][96], c[96][96];
main()
{
    int i, j;
    for (i=0; i<96; i++)
        for (j=0; j<96; j++)
            ci,j = ai,0*b0,j + ai,1*b1,j + ai,2*b2,j + ... + ai,94*b94,j +
            ai,95*b95,j;
}
```

Figure 1-43 Dot Product Matrix Multiply

Blocking

Figure 1-43 and Figure 1-44 show different algorithms to multiply two 96 x 96 square matrices. $a_{i,j}$ is used as shorthand for $a[i][j]$ in the figures. The algorithm of Figure 1-43 uses an unrolled dot product. This gives a high degree of parallelism. However, the blocked algorithm has better register and cache reuse.

A 96-element row of the array a is brought into memory to be multiplied with a column of the array b. The values cannot be reused until the entire dot product has been calculated. The blocked algorithm works using 6 x 6 pieces of the two matrices. 72 ($2 \times 6 \times 6$) values need to be brought in from memory.

There are 216 (6^3) product terms with a total of 432 (2×216) operands. Each input value can be reused six ($432/72$) times. Blocking allows 5/6ths of the load instructions to be eliminated.

This also gives better cache locality. This is because the dot product reads 96 elements of *b* in column order. The effect of this is limited here because the elements can fit in the cache.

```
void block(float (*restrict a)[96], float (*restrict b)[96], (float
(*restrict c)[95]){
    float (*restrict d)[96];
    d = c;
    c0,0 = c0,0+a0,0*b0,0+a0,1*b1,0+a0,2*b2,0+a0,3*b3,0+a0,4*b4,0+a0,5*b5,0;
    d0,1 = d0,1+a0,0*b0,1+a0,1*b1,1+a0,2*b2,1+a0,3*b3,1+a0,4*b4,1+a0,5*b5,1;
    (...)
    d0,5 = d0,5+a0,0*b0,5+a0,1*b1,5+a0,2*b2,5+a0,3*b3,5+a0,4*b4,5+a0,5*b5,5;
    c1,0 = c1,0+a1,0*b0,0+a1,1*b1,0+a1,2*b2,0+a1,3*b3,0+a1,4*b4,0+a1,5*b5,0;
    (...)
    d5,5 = d5,5+a5,0*b0,5+a5,1*b1,5+a5,2*b2,5+a5,3*b3,5+a5,4*b4,5+a5,5*b5,5;
}
float a[96][96], b[96][96], c[96][96];
main() {
    int i, j, k;
    memset(c, 0, sizeof(c));
    for (i=0; i<96; i+=6)
        for (j=0; j<96; j+=6)
            for (k=0; k<96; k+=6)
                block(&a[i][k], &b[k][j], &c[i][j]);
}
```

Figure 1-44 Matrix Multiply with Blocking

Table 1-25 compares the performance of the blocking and the dot product algorithms.

Table 1-25 Performance of Blocking and Dot Product Algorithms

	Instruction Cycles	Memory Accesses	Misses (Data Cache)	Miss Cycles (Data Cache)	ILP (Inner Loop)
Dot Product	966710	1778714	56549	10024263	4.74
Blocking	613710	589878	19053	306674	4.63

Two-Level Blocking

The blocking algorithm of Figure 1-44 brings in 144 bytes ($6 \times 6 \times 4$) of the *a* and *b* arrays into memory. Blocks of the *b* array are read in column order. By adding a second level of blocking, you can improve the cache locality. Figure 1-45 gives the algorithm. You can use the `block` procedure of Figure 1-44 (just change the dimension).

Three levels of loops, corresponding to the iteration variables (*i*, *j*, *k*) have been added inside the loop. The innermost (*k*) loop has been unrolled to reduce overhead. The three outer loops process 30×30 square blocks. The inner loops process 6×6 subsquares. The order (*i*, *j*, *k*) of the inner loops should be the same as the outer loops so that the *c* result is accumulated in the same order. Floating point addition is not associative. Comparative performance for single- and two-level blocking is given in Table 1-26. Although the extra loop levels increase the number of instructions, the overall performance is nearly doubled.

Table 1-26 Performance for Single and Two-Level Blocking

	Instruction Cycles	Memory Accesses	Misses (Data Cache)	Miss Cycles (Data Cache)	ILP	CPI
One Level Blocking	9555784	9216056	635049	9576916	4.39	2.002
Two Level Blocking	11029758	11061102	54423	1095367	4.52	1.099

```

void block(float(*restrict a)[262],float (*restrict b)[262],(float
(*restrict c)[262]);
float a[262][262], b[262][262], c[262][262];
main() {
    int i, j, k,, ii, jj;
    memset(0, c, sizeof(c));
    for (i=0; i<240; i+=30)
        for (j=0; j<240; j+=30)
            for (k=0; k<240; k+=30)
                for (ii=i; ii<i+30; ii+=6)
                    for (jj=j; jj<j+30; jj+=6) {
                        block(&a[ii][k],    &b[k][jj],    &c[ii][jj], &c[ii][jj]);
                        block(&a[ii][k+1],  &b[k+1][jj],  &c[ii][jj], &c[ii][jj]);
                        block(&a[ii][k+2],  &b[k+2][jj],  &c[ii][jj], &c[ii][jj]);
                        block(&a[ii][k+3],  &b[k+3][jj],  &c[ii][jj], &c[ii][jj]);
                        block(&a[ii][k+4],  &b[k+4][jj],  &c[ii][jj], &c[ii][jj]);
                        block(&a[ii][k+5],  &b[k+5][jj],  &c[ii][jj], &c[ii][jj]);
                    }
            }
}

```

Figure 1-45 Blocking Matrix Multiplication

Watch for Data Cache Bank Conflicts

The parameters to the program in Figure 1-46 are sets represented as bit vectors. The program tests whether one vector is included in another. 2958 cycles are necessary to test for inclusion of a 1024-word vector in another. 1024 of these are data cache stalls. The **tmsim** statfile line corresponding to the loop is given below:

tree name	execs	instc	istallc	dstallc	cpbacks	cnflctc	isopersexopers

_ _subset_DT_1	128	1792	87	1024	0	1024	5888 5887

The **cnflctc** column of the statfile is for data cache bank conflicts. All the stalls are bank conflicts.

There is a bank conflict whenever two memory accesses are made in the same cycle and bits two to four of the address are identical. This is the case in the program of Figure 1-46. The procedure `malloc` returns a pointer whose value is $\text{four mod } 2^n$, 2^n being the power of two immediately greater than or equal to the size. The pointers to `p` and `q` are, therefore, equal mod 2^{12} . The same index is used inside `subset` to reference both arrays. The two loads are scheduled in the same instruction because of scheduling latency constraints. A cycle is added for every access to the two arrays as a result. Adjusting the addresses of `p` and `q` by allocating an extra word and incrementing one of them so that bits two to four differ eliminates the conflicts.

```

int subset(int *b, int *a, int size)
{
    int i, result = 1;

    for (i=0; i<size; i+=8)
        result &= !(b[0] & ~a[0]) & !(b[1] & ~a[1]) & !(b[2] & ~a[2]) &
        !(b[3] & ~a[3]) &
                !(b[4] & ~a[4]) & !(b[5] & ~a[5]) & !(b[6] & ~a[6]) &
        !(b[7] & ~a[7]) ;
    return result;
}
main()
{
    (void)subset( (int*)malloc(1024*sizeof(int)),
                (int*)malloc(1024*sizeof(int)),
                1024);
}

```

Figure 1-46 Set Inclusion

Try -noloadspec When Thrashing

A program that initializes a hash table is shown in Figure 1-47. The table is heavily loaded (4000 out of 7300 values) and its size is bigger than the data cache. The **tmprof** output from compiling the program with profiling and grafting is shown below:

Treename	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
-----	-----	-----	-----	-----
__rt_umod_DT_0	4000	185685 39.79%	132	1553
__lookup_DT_1	4000	152565 32.69%	155	99101
__lookup_DT_4	1490	61097 13.09%	126	40906
__main_DT_1	4000	49052 10.51%	59	2499
(...)				
total/average		466655 100.00%	1578	166757

The TriMedia scheduler speculates load instructions because of latency. The effect of this is accentuated with grafting. This degrades the performance in this example because the data cache is already thrashing. The scheduler option `-noloadspec` prevents load speculation. The **tmprof** output from compiling the program with `-noloadspec` is shown below. Using the option saves about 50000 cycles. Note that a significant number of cache misses are attributed to main even though the memory accesses are not performed there. This is because the load of `val[i]` at the end of hash terminates after returning from the function.

Treename	Executions	Total Cycles (%)	I\$ Cycles	D\$ Cycles
-----	-----	-----	-----	-----
__rt_umod_DT_0	4000	185463 43.77%	132	1331
__lookup_DT_1	4000	116778 27.56%	180	59939
__lookup_DT_4	1490	52101 12.30%	145	30318
__main_DT_1	4000	51137 12.07%	59	27078
__lookup	4000	16058 3.79%	58	0
	(...)			
total/average		423696 100.00%	1622	118831

```

#define HSIZE 7300
#define HINCR 421
int key[HSIZE];
int val[HSIZE];
int lookup(unsigned k, short v) {
    int *keyp = key, *valp = val;
    int i = k % HSIZE;
loop: if (keyp[i] == k)
        return valp[i];
    else if (keyp[i]) {
        i += HINCR;
        if (i >= HSIZE)
            i -= HSIZE;
        goto loop;
    }
    keyp[i] = k; valp[i] = v; return v;
}

main() {
    unsigned i, seed = 0;
    for (i = 0; i < 4000; i++) {
        (void) lookup(seed, i); seed = seed*31415927U + 2818281;
    }
}

```

Figure 1-47 Initializing a Hash Table

Summary

The TriMedia Compilation System is geared toward profile-based program optimization methods. Some methods, such as grafting, are done automatically by the compiler taking into account the user directives in the form of grafting parameters. Other techniques, such as manual loop unrolling, the use of restricted pointers, and custom operators, currently need user intervention. It is expected that future versions of the compiler will include interprocedural optimization, source to source transformations, automated loop unrolling methods, and better alias analysis. However, mechanisms like restricted pointers to pass specific user knowledge to the compiler, and the use of custom operators to exploit the TriMedia architecture to the maximum extent are likely to remain in the application programmer's domain.

To get started in optimization, you should first compile the program and run it with the `-O3` and `-p` (profile) option to make the decision tree frequencies and probabilities appear in the tree code. It is a good idea to look at the assembly code and because cache misses are a significant performance factor in most applications. You can then find the hotspots using **tmprof**. The C source and the tree file (.t) and assembler (.s) output corresponding to the hotspots should be examined together.

An important number of `after` keywords in the tree code for loads and stores usually indicates a need to use restricted pointers. The shape of the decision trees (number of leaves, branch structure, probabilities) provides important information about program restructuring. Frequently executed decision trees containing only a few operations indicate a control flow problem.

You must also pay attention to the memory behavior of the program. Unfortunately, memory statistics are only provided on a global basis by the current version of the SDE. You can recognize problems in the critical path, including aliasing, from sequences of `nops` in the assembler code.

You should restructure the C code, compile it with profiling, run it again with **tmprof**, and recompile and analyze it as many times as necessary. You need to apply grafting last because it is impossible to understand what is happening after grafting. You should apply loop unrolling only if it produces better performance than grafting. To determine this, measurement is necessary. It is a good idea to prepare a sample input smaller than the full set so as not to lose time running **tmsim**. On a Sparcstation 20, the ratio of real time to simulated time is about 36000 to 1. Understanding the instruction set helps in optimization. The ILP factors reported in the statfile and by **tmprof** can include operations that become redundant after optimization. These figures should be taken as estimates.

There are tools for performance analysis not mentioned in this chapter. For instance, there are tools to investigate produced schedules in detail by report options to **tmsched** and use of the tool **tmcritpath** to investigate the critical path of a schedule.

Besides the support for optimizing programs, the TriMedia Compilation System offers support for system level programming. Interrupt service routines can be programmed in C and support is added for using the most interesting cache instructions.

Chapter 2

System Programming Support

Topic	Page
Programming Support	2-2
Interrupt Service Routines and Exception Handlers	2-2
Using MMIO Locations	2-11

Programming Support

The TriMedia Compilation System offers system level programming at the C level. For instance, interrupt service routines and fine control of the data cache are supported. The toolset comprises interrupt latency inspection and offers support for interrupt latency control. This section describes what the toolset offers to programmers needing one of these features.

Interrupt Service Routines and Exception Handlers

The TriMedia C compiler allows the implementation of interrupt service routines (*handlers*, for short) and exception handlers entirely in C. The distinction between interrupt handlers and exception handlers is made clear in the next section, “User View.” First, we talk about the general mechanism and do not distinguish between the two types of handlers.

The compiler allows maximal flexibility in handlers, and transparently generates code that uses the appropriate return address and does the additional register saving that is required for certain types of handlers. Additionally, just as in the case of normal functions, the compiler attempts to minimize the calling overhead of handlers. Because handlers are nonstandard, this section contains some implementation detail to explain what you can expect from them.

User View

For your purpose, the only difference between handlers and functions is the way in which they are activated. You must explicitly call functions, whereas you must activate handlers upon an interrupt. Note that the compiler checks the type and parameter list of a handler but does not check erroneous calling of interrupt handlers. Normal functions which attempt to mimic a handler cause failures under certain conditions, because handlers have different register saving requirements.

Except from the fact that handlers are not allowed to return a result, there are no further differences between functions and handlers. Any legal resultless function with the specified number of parameters can be declared as a handler and therefore, the handler’s body can range from simple updates to some flag in shared memory to complex control flow using conditionals, loops, and calls to other functions. However, as is the case for any C function, the calling overhead is strongly dependent on the complexity of the handler. See also the description of the calling sequences generated by the compiler for functions and handlers in “Declaring Interrupt Service Routines” on page 2-5.

Handlers come in three varieties: *interruptible*, *non-interruptible*, and *exception*. The first two are interrupt handlers that you can use for any of the vectored interrupts specified for the TriMedia processor. You can use an exception handler for any type of exception, such as misaligned store exception, floating point exceptions, and so on. The interrupt handlers have no parameters and come in a noninterruptible and an interruptible form. The difference is that interruptible handlers allow service of new interrupts of any kind during their invocation (that is, *nested* interrupts), while noninterruptible handlers clear the interrupt enable bit (**IEN**) in the processor status word during their invocation and, therefore, can only be interrupted by *nonmaskable interrupts* (NMIs). This simple distinction between interrupt handlers is useful in many cases. However, sometimes you might require a finer level of interrupt masking. You must explicitly code such finer level masking using saving, modifying and restoring of the **IMASK**. For details on this, see *TM-1000 Data Book*.

Exception handlers are interruptible and get one parameter, the value of **spc** (saved program counter). You should install exception handlers with care because they might interfere with the exception handler installed by the debugger. The debugger uses an exception handler to single-step (dtree steps) through a program. So any user program should be certain not to destroy the debugger's handler.

Handlers make use of the stack of the process that was active at the moment of the interrupt. *There is no automatic escape to a system stack.*

```

#include <tm1/MMIO.h>

volatile int s;

void handler1(void)
{
#pragma TCS_handler
s++;
}

void handler2(void)
{
#pragma TCS_handler
int i;

for (i=0; i<100; i++)
    s += i;
}

void handler3 (void)
{
#pragma TCS_interruptible_handler
do_the_work_while_allowing_interrupts();
}

/* ----- */

void install_handler(
    int          nr,
    handler_type handler)
{
    base_of_mmio[ INT_VECS + nr ] = handler;
}

/* ----- */

main() {
install_handler(1,handler1);
install_handler(2,handler2);
install_handler(3,handler3);
}

```

```

{__handler1;}
{__handler1_DT_0;}
3 uimm (_s);
2 ld32 3;
4 iaddi (1) 2;
5 st32 3 4
        after 2;
6 readdpc;
cgoto 6;
endtree (*__handler1_DT_0*)

```

Figure 2-1 Sample Interrupt Handler

Saving/Restoring Behavior

Similar to functions, handlers save and restore all the callee-saved registers that they use (including the frame pointer). Contrary to functions, handlers obtain their return address from the processor's destination program counter (DPC). In case a nested interrupt is possible during the execution of the handler, the DPC is also saved at entry.

Unlike normal functions, caller-saved registers that might be modified by executing the handler are also saved and restored at the handler's entry and exit. For handlers that call other functions, this means that the entire set of caller-saved registers is saved and restored. For handlers that do not call other functions, this means that caller-saved registers are treated as callee-saved registers, that is, only saved and restored when used by the handler itself. Note that argument registers and the return pointer register are special cases of caller-saved registers.

Additionally, the code generated for noninterruptible handlers can save and restore the interrupt enable bit (IEN) in the processor status word. Any other change to the processor state during the handler's invocation remains visible after termination of the handler. This especially holds for the *source program counter* (SPC), which is used during exception processing to determine the decision tree in which the exception occurred.

Declaring Interrupt Service Routines

Interrupt handlers must be defined as parameterless, resultless functions, with either `#pragma TCS_handler`, or `#pragma TCS_interruptible_handler` (as appropriate) in the beginning of the function body. By default (that is, declared as `TCS_handler`), handlers are noninterruptible.

You must define exception handlers as a resultless function with a single `void *` parameter, using the `#pragma TCS_exception_handler`. Remember the warning that any explicit installation of an exception handler might cause the debugger to malfunction.

Figure 2-1 shows some examples of handlers, and of a simple generic function that is used to install them in the interrupt vector region of the MMIO space. See the *TM-1000 Data Book* and "Using MMIO Locations" on page 2-11.

Included in the figure is a sample translation to decision-tree intermediate code of a very simple handler.

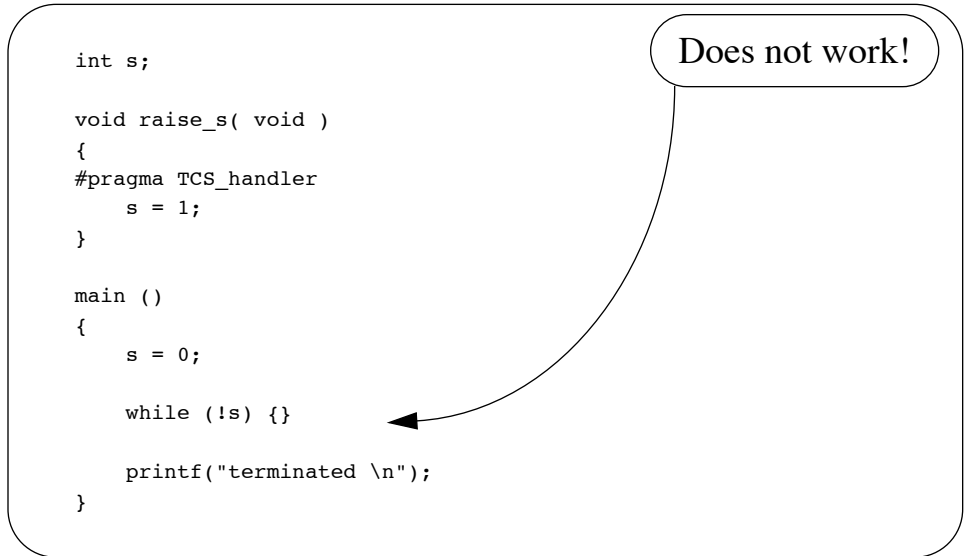


Figure 2-2 Use Volatile Shared Variables

Usage Notes

You must be aware that interruptible handlers, allowing nested interrupts, require some additional care. First, you must make reentrant, a nested invocation of a same handler does not corrupt the invocation state of the interrupted one. Second, with the possibility of nested interrupts, it has become essential to guarantee that the occurrence rate of interrupts does not exceed the system's capacity to service them for any longer amount of time. Where such a situation only results in slow response or the loss of events in systems that do not allow nesting of interrupts, it might cause a crash in systems that do.

Any data that is shared between handlers and the mainstream program, or between handlers and other handlers, must be declared volatile to prevent surprising effects caused by optimizations. The optimizer might decide to replace loads from nonvolatile variables by earlier results, which might not be desired for shared data. An example is shown in Figure 2-2, which shows an illegal way for synchronization on an event: because the shared variable `s` is nonvolatile, the optimizer propagates the earlier assigned value 0 to the loop test condition. This results in a never ending loop, even when the handler is triggered.

Interrupt-Latency Support

Real-time system programmers have to be sure that the interrupts that occur during execution are handled within a certain number of cycles. Because on TriMedia decision trees are executed as large chunks of critical sections (noninterruptible code), special care is taken in the hardware as well as in the toolset. This section discusses how you can find out interrupt latencies for your particular program. No automatic support to guarantee a certain interrupt latency is given¹.

This section also addresses how you can find out whether the interrupt latency is more than a given threshold and how you can modify the code to reduce the interrupt latency. The support offered is threefold. First, there is a means to inspect violations of a certain threshold of cycles executed between interruptible jumps. Second, there are statistics from the simulator that produce a raw data histogram describing how many dtrees are executed with a certain number of cycles between interruptible jump. Also, the last dtree that executed that many cycles is shown. Third, there is a pragma, `TCS_break_dtree`, honored by the compiler with which you can force the compiler to create smaller critical sections.

Supporting the Machine Level Simulator: `tmsim -il`

To get more insight into interrupt latency of a program, you can run the simulator `tmsim` with the option `-il`. This produces a report of the following form (in the reportfile when specified, otherwise on standard output).

1. The rationale for this is that automatic support has to be based on worst-case assumptions for all instructions executed. This is a *very* unrealistic situation, especially when assuming each load/store leads to cache miss, on top of losing the maximum number cycles for arbitration and the maximum number of requests serviced before getting the bus. Our experiments with a major application like MPEG-1 + RTOS showed that the number of cycles executed between two interruptible jumps was worst case 30 μ s, while the TM-1000 hardware survives 300 μ s. The TM-1000 DMA-based peripherals have no short-term real-time constraints (order of several milliseconds) and the most critical peripheral is the synchronous serial interface SSI.

```

tmsim -il -mm a.out
interrupt latency distribution
000004 002644 __vfprintf_DT_16 + 1c
000005 002592 ____sinit_DT_0 + 1b84
000006 000799 ____swrite_DT_2 + 28
000007 000173 __fflush_DT_3 + 26
000008 002584 __vfprintf_DT_10 + 39
000009 000462 __vfprintf_DT_34 + 5c
*** several lines deleted
000223 000001 __fwalk_DT_0 + 106
000289 000001 __latency_isr_DT_0 + 10b
000411 000001 __fwalk_DT_4 + 17b
000419 000001 __memchr_DT_2 + 304
000758 000001 __foo_DT_0 + 457

```

The first column in the report is the number of cycles between two interruptible jumps. The second column is the number of times a dtree was executed with that number of cycles. The last column names the dtree and address of the exit from the dtree last executed for the given number of cycles between interruptible jumps. For example, the last line means that the execution of the tree `__foo_DT_0` required 758 cycles and the number of cycles between two interruptible jumps of 758 was seen only once during the execution of the program. Similarly, during the execution of the entire program there were 462 instances where the number of cycles between two interruptible jumps was equal to 9. Among these, the last time this happened was while executing the dtree at `__vfprintf_DT_34`.

Breaking Decision Trees: `#pragma TCS_break_dtree`

When you find the interrupt latency is too high, you can control the latency by changing the way the program gets compiled. A high interrupt latency implies a large decision tree. The reason for the presence of a large decision tree could be too much grafting, or it could be a large decision tree even without grafting, where you might have hand unrolled loops to gain performance, something that is not too uncommon in DSP programming.

If grafting is the cause of large interrupt latency, you can use grafting parameters to reduce the amount of grafting performed. Because this might have a performance impact, you should exercise care in achieving a balance between performance and interrupt latency.

If the code has large decision trees even without grafting, you can use the `pragma TCS_break_dtree` to break the dtree at appropriate places. This also might have a performance impact. You must take special care to minimize the number of values living across the break of the dtree. These values now have to be stored in the global register set of the compiler with the accompanying save and restore code.

Supporting Cache Control

You can use the cache operations as specified in the *TM-1000 Data Book*, `dcb`, `dinvalid`, `iclr`, `rdstatus`, and `rdtag` through the `custom_op` mechanism discussed in “Using Custom Operators” on page 1-46. However, these `custom_ops` are *not* available through the `custom_defs.h` include file and are *not* directly supported by the compiler. This means that when you use these instructions directly (through a `custom_op` declaration), the ordering of memory accesses and the cache custom operations does not work.

```
void _cache_invalidate(void *address, int size);
void _cache_copyback(void *address, int size);
void _cache_allocate(void *address, int size);
```

However, the TriMedia C library supports a more ‘user view’ model for using the most interesting custom operations at the C level, for example, copying back, allocating, or invalidating a piece of memory.

As an example, take the entry point `_cache_invalidate`. The semantics are: invalidate the piece of memory [`address`, `address + size`¹). *The entire contents of the cache blocks in the range will disappear. Any dirty data will be lost.* Calls to `_cache_invalidate` are translated to tissues of the appropriate number of `dinvalid`s. *The object referenced by the pointer should be cache aligned with respect to its upper and lower bounds.*

The `_cache_copyback` entry point flushes dirty data back to the cache. This can be used prior to starting DMA or before a `cache_invalidate` if required. Unlike `invalidates`, `copybacks` are not destructive and the range does not need to be aligned. The `_cache_allocate` entry point resets the dirty bit in all data in the range. The memory range should be cache aligned.

The same routines are available as custom operators. These should only be used for extremely time critical code. A code explosion is possible if the area is of significant size. The second argument is the number of `dinvalid` operations. They are made available through the include file `<ops/custom_defs.h>`, as custom operators `ALLOCATE`, `INVALIDATE`, and `COPYBACK`. The ordering of all memory operations that alias with the memory region (`address`, `address + number_of_cache_blocks * size_of_cache_line`) is respected.

The example from Figure 2-3 compiled with `tmcc -O3 -t example.c` translates into the trees code in Figure 2-4. In the example, note that `*c = 3`; aliases with all other memory locations because the address of `c` is unknown. The `INVALIDATE` call at the C level is translated into two `dinvalid` operations, which are ordered among all aliasing memory operations. The store to value `a[1]` is ordered only against the first `dinvalid` operation,

1. Currently, the cache block size is 64 bytes.

because the compiler assumes¹ it overlaps only with the first cache line. In general, the input ordering of all cache operations is maintained. Note that the assignment `b[10] = 3;` is free to move across the `dinvalid` operations.

```
#include <custom_defs.h>
chara[1000];
charb[1000];
char*c;

foo()
{
    *c = 3;
    b[10] = 4;
    INVALIDATE(a, 2);
    a[1] = 1;
    return a[1];
}
```

Figure 2-3 Example of Use of Cache `custom_ops`

1. Note that the assumption might not be true when the address (the first parameter to the `invalidate` call) is not cache-block aligned, as in this case!

The other cache operations are completely analogous to the `invalidate custom_op` and are not discussed any further.

```

{__foo;}
{__foo_DT_0;}
entree (0)
    2 uimm (_c);
    1 ld32 2;
    4 iimm (0x3);
    3 st8 1 4 (* *c = 3; *)
      after 1;
    6 uimm (_b);
    8 iimm (0x4);
    7 st8d (10) 6 8(* b[10] = 4; *)
      after 3;
    10 uimm (_a);
    11 uimm (0x40);
    9 dinvalid(0) 10(* invalidate [a, a+64] *)
      after 3;
    12 uimm (64 + _a);
    13 dinvalid(0) 12(* invalidate [a+64, a+128] *)
      after 9 3;
    16 rdreg (1);
    15 st8d (1) 10 16(* a[1] = 1; *)
      after 9;
    18 wrreg (5) 16;
    19 rdreg (2);
    cgoto 19
endtree (*__foo_DT_0*)

```

Figure 2-4 Intermediate Representation for Cache `custom_op` Example

Using MMIO Locations

Writes to MMIO locations do not take effect immediately. For example, if there is a write to the `IPENDING` location in cycle i that generates an interrupt, the interrupt is not triggered if an `ijmpi` operation is executed in cycle $i+1$. The interrupt is taken if the `ijmpi` operation was executed in cycle $i+2$. The amount of delay required for a write to an MMIO location is dependent on the location, this data will be available soon. In the next full release, automatic support will be given for the scheduling delays.

Chapter 3

Case Studies

Topic	Page
Introduction	3-2
Special-Purpose Block Filter	3-2
Fixed-Point Arithmetic	3-4
IFIR16 Custom Operations	3-5
Dual-Phase Loop	3-6
Critical Path	3-8
Algebraic Transformation	3-9
Balancing the Critical Path	3-10
More Unrolling	3-11
Matrix Transpose	3-12
Divide and Conquer	3-14
Using Custom Operations	3-15
Inlining and Shrink-Wrapping	3-16
Cache Alignment	3-19

Introduction

Figure 3-1 shows the source code for a block FIR filter with floating-point arithmetic. The filter has been structured as a general-purpose library routine. The array of filter coefficients are supplied in an argument. The filter components are computed element by element. A separate function `dotap` is used to compute an element.

```
void blkfir(float *input, float *state, float *coeff, float *output,
           int npoints, int ntaps)
{
    int i;
    for (i=0; i<npoints; i++) {
        output[i] = dotap(input[i], state, coeff, ntaps);
    }
}

float dotap(float input, float *state, float *coeff, ntaps)
{
    int i;
    float sum = 0.0;
    state[0] = input;
    for (i = ntaps; i>0; i--) {
        state[i] = state[i-1]; /* slide window */
        sum = sum + state[i] * coeff[i];
    }
    return sum;
}
```

Figure 3-1 General-Purpose Block Filter

Special-Purpose Block Filter

TM-1000 speed is critical when writing a routine that is specialized for a particular purpose. Implementation of a filter requires memorization of state information. You must use an array to represent the state if the size is arbitrary. Fixing the length allows it to be stored in scalars. These can be allocated to registers. The TM-1000 has 128 general-purpose registers. If the length is variable, a loop is needed to evaluate the output value. This adds a control dependence that limits ILP. By fixing the state length beforehand, you can use a closed form expression. This has more ILP. In the example of Figure 3-1, the program is divided into two functions. This interferes with the optimizing ability of the compiler.

In the program of Figure 3-1, the array `coeff` is provided as a parameter. Only two accesses to memory can be made per instruction on TM-1000. If the routine is specialized for a particular set of coefficients, these can be placed as constants in the instruction stream. This reduces memory accesses and eliminates latency.

```
void blkfir(float *input, float *output, npoints)
{
    int i, j;
    float state1, state2, state3, state4, state5, state6, state7, state8;
    state2 = state3 = state4 = state5 = state6 = state7 = state8 = 0.0;
    for (i=0; i<npoints; i++) {
        state1 = input[i];
        output[i] = state1 * 0.5 + state2*0.25 + state3*0.125 + state4*0.0625 +
            state5*0.03125 + state6*0.015625 + state7*0.0078125 + state8*0.00390625 ;
        state8 = state7;    state7 = state6;    state6 = state5;    state5 = state4;
        state4 = state3;    state3 = state2;    state2 = state1;
    }
}
```

Figure 3-2 Specialized Filter

Figure 3-2 shows a specialized version of the routine for a state length of eight and a fixed set of coefficients. Eight scalar variables are used to represent the state. The two functions have been collapsed into one. Table 3-1 compares the performance of the two programs. After elimination of the loop and the arrays for `coeff` and `state`, only 1129 instruction cycles are necessary, compared to 5611 previously.

```
void blkfir(int *input, int *output, npoints)
{
    int state1 = 0, state2 = 0, state3 = 0, state4 = 0;
    int state5 = 0, state6 = 0, state7 = 0, state8 = 0;
    for (i=0; i<npoints; i++) {
        state1 = input[i];
        output[i] = IMULM(state1, 0x10000000) + IMULM(state2, 0x08000000) +
            IMULM(state3, 0x04000000) + IMULM(state4, 0x02000000) +
            IMULM(state5, 0x01000000) + IMULM(state6, 0x00800000) +
            IMULM(state7, 0x00400000) + IMULM(state8, 0x00200000);
        state8 = state7; state7 = state6; state6 = state5; state5 = state4;
        state4 = state3; state3 = state2; state2 = state1;
    }
}
```

Figure 3-3 Filter with Fractional Arithmetic

Table 3-1 Special-Purpose Versus General-Purpose Filter

	ILP	Instruction Cycles		
		Per Tap	Per Input	Total
General Purpose Filter	1.06	17.5	140	5611
Special Purpose Filter	1.42	3.52	28.2	1129

Fixed-Point Arithmetic

Seven additions are necessary for each iteration of the loop of Figure 3-2. These have a latency of three cycles. Floating-point addition is commutative but not associative, for example, $10^8 + (-10^8 + 1)$ is not the same as $(10^8 + -10^8) + 1$. The C language requires that, in the absence of parentheses, floating-point arithmetic be executed in strict left to right order. In the program of Figure 3-2, this means that the additions must be executed in sequential order. A total of 21 cycles (7×3) is necessary to sum the seven products in sequential order using floating point.

Integer addition is both commutative and associative, so the compiler can balance the chain of additions in a tree, reducing dependences and increasing parallelism. Seven addition operations can be represented in a binary tree of height three. Integer addition has a latency of only one cycle. Three cycles (3×1) are necessary to sum the seven products in parallel. You can use integer arithmetic can be used by changing to a fixed point representation.

You can represent fixed point numbers in what is called $Q.n$ representation. The binary point is after the n th least significant bit. The bits to the right of the binary point correspond to the fractional part of the number. The most significant bit corresponds to the sign. The number of bits available for the integer part depends on the word length (16, 32, or 64 bits).

For this filter, the inputs are specified to be between -1 and +1. You can represent them in $Q.31$ form. The coefficients are between 0 and 1. The output of the filter is a sum of eight products between -1 and +1. It is between -8 and +8. Three bits are sufficient to represent the integer part ($Q.28$ form). The product of two numbers in $Q.n$ and $Q.m$ form is in $Q.n+m$ form. If we represent the coefficient in $Q.29$ form and the input in $Q.31$ form, the 64-bit product is in $Q.60$ form. The high order 32 bits are given by the TM-1000 `IMULM` instruction. This gives us a result in $Q.28$ form (60-32), as desired. You can use `IMULM2` as a custom operation in the program by including the header file `<ops/custom_defs.h>`.

Figure 3-3 shows the source program after recoding to use fractional arithmetic. Table 3-2 shows the improvement due to the introduction of fixed-point arithmetic. Execution time is more than doubled, as is the ILP.

Table 3-2 Fixed Versus Floating Point Arithmetic

	ILP (Inner Loop)	Instruction Cycles
Special Purpose + Floating Point	1.42	1129
Special Purpose + Fixed Point	3.33	489

IFIR16 Custom Operations

Changing to a fixed-point representation permits use of data-parallel custom operations. You can compute the sum of two products in a single IFIR16 instruction. Recoding the algorithm to use IFIR16 involves changing the representation from 32 to 16 bits. You must represent the inputs in Q.15 form, the outputs in Q.12 form, and the coefficients in Q.13 form. The smallest coefficient is 2^{-8} , which fits in 13 bits. The state and coefficients are represented in halfword pairs. The high order halfword corresponds to the first element and the low order halfword corresponds to the second element of the pair.

Representing elements in halfwords complicates the handling of the state. When there is a variable per-state element, shifting the state corresponds to seven register moves and one load. Up to five register moves can execute in parallel on TM-1000. When each register has two elements, shifting the state requires matching up the second element of each pair with the first element of the next. This corresponds to extracting the middle 32 bits of the 64-bit concatenation of the two pairs. This is possible with the TM-1000 FUNSHIFT2 instruction.

Figure 3-4 shows the source program after recoding to use IFIR16 and FUNSHIFT2. To increase efficiency, the coefficient constants have been moved to registers. Table 3-3 shows the comparative performance with and without IFIR16. *The number of instruction cycles is reduced by 40%.*

Table 3-3 Comparative Performance

	ILP (Inner Loop)	Instruction Cycles
Special Purpose + Fixed Point	3.33	489
Special Purpose + Fixed Point + Ifir16	3.28	289

```

void blkfir(short *input, int *output, int npoints)
{
    int i;
    int state01 = 0, state23 = 0, state45 = 0, state67 = 0;
    int coeff01 = 0x10000800, coeff23 = 0x04000200, coeff45 = 0x01000080,
        coeff67 = 0x00400020;
    for (i=0; i<npoints; i++) {
        state01 = FUNSHIFT2(input[i],state01); /* state1 = state0 */
                                                /* state0 = inputi */
        output[i] = IFIR16(state01, coeff01) + IFIR16(state23, coeff23) +
                    IFIR16(state45, coeff45) + IFIR16(state67, coeff67);
        state67 = FUNSHIFT2(state45, state67);/* state67 = state56 */
        state45 = FUNSHIFT2(state23, state45);/* state45 = state34 */
        state23 = FUNSHIFT2(state01, state23);/* state23 = state12 */
    }
}

```

Figure 3-4 Filter with Custom Operations

Dual-Phase Loop

Several factors still limit the performance of the inner loop. The key factor is the 16-bit alignment of the state because of `IFIR16`. Shifting the state using `FUNSHIFT2` is cumbersome and slow. Only a halfword of data is read per cycle. A minimum of five cycles is necessary per output element because of the loop. You cannot reduce the overhead by unrolling because there is a dependence on the state.

```

void blkfir(int *input, int *output)
{
  int coeff01 = 0x10000800, coeff23 = 0x04000200, coeff45 = 0x01000080,
      coeff67 = 0x00400002;
  int x01, x23, x45, x67, y01, y23, y45, y67, i, statenew;

  y01 = *input++; y23 = y34 = y56 = 0; /* y2..7 = 0 ; y1 = input1 ; y0 = input0 */
  x01 = PACK16MSB(y12, 0); x23 = x45 = x67 = 0; /* x1..7 = 0 ; x0 = input0 */
  for (i=0; i<npoints; i+=2) { /* npoints must be even */
    statenew = *input++;
    *output++ = IFIR16(x01, coeff01) + IFIR16(x23, coeff23) +
                IFIR16(x45, coeff45) + IFIR16(x67, coeff67);
    *output++ = IFIR16(y01, coeff01) + IFIR16(y23, coeff23) +
                IFIR16(y56, coeff45) + IFIR16(y67, coeff67);
    y67 = y45; y45 = y23; y23 = y01; y01 = statenew;
    /* y2..7 = y0..5 ; y1 = inputi+1 ; y0 = inputi */
    x67 = x45; x45 = x23; x23 = x01; x01 = PACK16MSB(y01, y23<<16);
    /* x2..7 = x0..5 ; x0 = y1 ; x1 = y2 */
  }
}

```

Figure 3-5 Two-Phase Loop

You can sidestep all these restrictions by separating the execution of the loop into two phases. The x phase corresponds to the even-numbered outputs (output₀, output₂, output₄). The second phase corresponds to the odd-numbered outputs (output₁, output₃, output₅). Each has its own state. The state of the y phase corresponds to the state of the x phase shifted one input element. Two elements are processed per loop iteration. This allows register copies to be used instead of FUNSHIFT2 for the state. Doubling the number of elements divides jump overhead by two. One half as many memory accesses (1 x 32 bits instead of 2 x 16) are made in the dual phase loop. Figure 3-5 shows the source program after recoding. Table 3-4 compares performances of the single and dual-phase loops.

Table 3-4 Single- and Dual-Phase Loop Comparison

	ILP (Inner Loop)	Instruction Cycles
Special Purpose + Fixed Point + Ifir16	3.28	289
Special Purpose + Fixed Point + Ifir16 + Dual-Phase Loop	4.61	173

Table 3-5 summarizes the improvements in performance due to successive refinements of the program. There is a reduction by a factor of 32 in the execution time. The final program has more than 90% issue-slot utilization.

Table 3-5 Performance Improvements by Program Refinement

	ILP	Instruction Cycles		
		Per Tap	Per Input	Total
General Purpose + Floating Point	1.06	17.5	140.2	5611
Special Purpose + Floating Point	1.42	3.5	28.2	1129
Special Purpose + Fixed Point	3.33	1.52	12.2	489
Special Purpose + Fixed Point + Ifir16	3.28	0.90	7.2	289
Special Purpose + Fixed Point + Ifir16 + Dual-Phase Loop	4.61	0.54	4.3	173

Critical Path

Horner's algorithm for evaluating a polynomial is shown in Figure 3-6. The array `P` gives the coefficients. $P(x) = (x+1)^{20}$. Thus, $P(-1) = 0$, $P(0) = 1$, and $P(1) = 2^{20}$.

```
#include <stdio.h>
#define DEGREE20
float P[DEGREE+1] = {
    1, 20, 190, 1140, 4845, 15504, 38760, 77520, 125970, 167960, 184756,
    167960, 125970, 77520, 38760, 15504, 4845, 1140, 190, 20, 1
};
float poly_eval(float *a, int size, float x) {
    float result = 0;
    while (size >= 0) {
        result = result * x + a[size];
        --size;
    }
    return result;
}

main() { printf("y = %f\n", poly_eval(P, DEGREE, 1.0)); }
```

Figure 3-6 Polynomial Evaluation Using Horner's Algorithm

You can estimate the degree of ILP in the program by running `tmsim` with the `-statfile` option and examining the resulting file. Table 3-6 gives the line of the file corresponding to the `while` loop of the function `poly_eval`. Without grafting, the issue-slot utilization of 1.49 (=188/126) is very low.

With grafting, the issue slot utilization is 1.61 (208/129), so it seems there is more ILP. However, more instructions are necessary to do the same task (129 versus 126).

The ILP is limited here by the length of the critical path. In the loop, the critical path corresponds to the calculation of a new value for the variable `result`. Each calculation requires a floating point multiplication, a floating point addition, and a coefficient load. The multiplication and the load both have a latency of three cycles. However, they can proceed in parallel. The addition has a latency of three cycles and depends on the other two operations. It is the `1sum` of latencies ($3 + 3 = 6$) that determines the execution time. 126 cycles are necessary to execute 21 iterations of the loop.

Grafting reduces execution time when the ILP is limited by control flow. In this case, the three extra cycles with grafting are due to speculative evaluation during the final iteration. There are 21 values per result and two are evaluated per iteration.

Table 3-6 Line of the File Corresponding to the `while` Loop of the Function `poly_eval`

	Execs	Instc	Istallc	Dstallc	Cpbacks	Cnflctc	Isopers	Exopers
No Grafting	21	126	29	24	0	0	189	188
Grafting	6	129	88	31	0	0	211	208

Algebraic Transformation

Horner's algorithm is optimal in terms of the number of operations, but it is inherently sequential. By means of an algebraic transformation, you can multiply the parallelism by two. You can decompose polynomial $P(x)$ into the sum of two polynomials, $Q(x)$ and $Q'(x)$, corresponding to the even and odd powers of x , respectively. It is then possible to substitute $x * R(x)$ for $Q'(x)$, where $R(x)$ is a polynomial having only even powers of x also. At this point, we can substitute $y = x^2$ in $Q(x)$ and $R(x)$, because both polynomials contain only even powers of x . For example:

$$P(x) = a_0 * x^0 + a_1 * x^1 + a_2 * x^2 + a_3 * x^3 + a_4 * x^4 + a_5 * x^5$$

$$Q(y) = a_0 * y^0 + a_2 * y^1 + a_4 * y^2$$

$$R(y) = a_1 * y^0 + a_3 * y^1 + a_5 * y^2$$

You can evaluate the polynomials $Q(x)$ and $R(x)$ in parallel using Horner's rule, doubling the parallelism. Figure 3-7 shows source for a parallel version of `poly_eval`. An adjustment is necessary for the case where there is an odd number of coefficients (in this case, Q and Q' have differing degrees). This corresponds to a problem that occurs frequently when programming an unrolled loop with a variable size input in TriMedia. There is a reduction in the number of cycles from 129 to 101 for the parallel version. This is somewhat disappointing.

```

float poly_eval(float *a, int size, float x)
{
    float result1, result2, y;
    int adj;

    y = x * x;
    adj = (size+1) & 1;
    size -= adj;
    result1 = IZERO(adj, a[size+1]);
    result2 = 0;
    while (size > 0) {
        result1 = result1 * y + a[size-1];
        result2 = result2 * y + a[size];
        size -= 2;
    }
    return result1 + result2 * x;
}

```

Figure 3-7 Parallel Polynomial Evaluation

Balancing the Critical Path

Looking at Figure 3-7, note that `result1` and `result2` are calculated from `a[size]` and `a[size-1]`. The reference to `a[size]` corresponds to the scaled index addressing mode on TM-1000. Calculating `a[size-1]` requires one more cycle for the subtraction. The critical path is unbalanced as a result. There are several ways to balance the critical path. You can use pairs of index variables (`a[size]`, `a[size1]`), for example. In this case, the best solution is to adapt the algorithm to use pointers instead of indices for the arrays. The modified source code is shown in Figure 3-8. The references to `ap[0]` and `ap[-1]` correspond to the displacement addressing mode on TM-1000. Using pointers, only 81, compared to 101, cycles are necessary.

```

float poly_eval(float *a, int size, float x)
{
    float result1, result2, y, *ap;
    int adj;
    y = x * x;
    adj = (size+1) & 1;
    size -= adj;
    ap = &a[size];
    result1 = IZERO(adj, ap[1]);
    result2 = 0;
    while (ap > a) {
        result1 = result1 * y + ap[0];
        result2 = result2 * y + ap[-1];
        ap -= 2;
    }
    return result1 + result2*x;
}

```

Figure 3-8 Balanced Critical Path

More Unrolling

Figure 3-9 shows the source for `poly_eval` when the loop has been unrolled to evaluate four polynomials in parallel. With unrolling, only 61, as compared to 81 cycles, are necessary to evaluate the polynomial. If the coefficients and degree of a polynomial are fixed in advance, more reduction in execution time is possible. Only 31 cycles are necessary to evaluate a polynomial of degree 20 on TM-1000. Source is given in Figure 3-10. Table 3-7 summarizes the time required to evaluate $(x+1)^{20}$, depending on the algorithm.

Table 3-7 Time Required to Evaluate $(x+1)^{20}$

Calculation Of 21-Point Polynomial	Instruction Cycles
Horner's Algorithm (Figure 3-6)	126
Two-Way Parallel (Figure 3-7)	101
Two-Way Parallel with Pointers (Figure 3-8)	81
Four-Way Parallel with Pointers (Figure 3-9)	61
Fixed $(x+1)^{20}$ Algorithm (Figure 3-10)	31

```

float poly_eval(float *a, int size, float x)
{
    float result1, result2, y, *ap;
    int adj;
    y = x * x;
    adj = (size+1) & 1;
    size -= adj;
    ap = &a[size];
    result1 = IZERO(adj, ap[1]);
    result2 = 0;
    while (ap > a) {
        result1 = result1 * y + ap[0];
        result2 = result2 * y + ap[-1];
        ap -= 2;
    }
    return result1 + result2*x;
}

```

Figure 3-9 Balanced Critical Path

```

float poly_eval(float x) {
    float result1, result2, result3, result4;
    float x2 = x*x, x3 = x2*x, x4 = x2*x2, x8 = x4*x4;
    result1 = 1 + x4*4845 + x8*125970 + x4*x8*125970 + x8*(x8*4845 + x8*x4);
    result2 = 20 + x4*15504 + x8*167960 + x8*x4*77520 + x8*x8*1140;
    result3 = 190 + x4*38760 + x8*184756 + x8*x4*38760 + x8*x8*190;
    result4 = 1140+ x4*77520 + x8*167960 + x8*x4*15504 + x8*x8*20;
    return (result1 + result2*x) + (result3*x2 + result4*x3);
}

```

Figure 3-10 Source

Table 3-8 Instruction Cycles by Calculation

Calculation of 21-point polynomial	Instruction Cycles
fixed $(x+1)^{20}$ Algorithm (Figure 3-9)	31

Matrix Transpose

Computing the transpose of a matrix is useful in image processing. If the row and horizontal indices correspond to the x and y axis, transposition corresponds to a reflection about the x - y diagonal. Figure 3-11 shows a program. The dimension is coded as a power of two. The routine is used as follows:


```
#define SIZE 4 /* for a 16 by 16 matrix */
char matrix[1<<SIZE][1<<SIZE];
...
transpose(matrix);.
```

Table 3-9 indicates performance figures for different sizes. They were obtained with **tmprof** and **tmsim**.

The total execution time is the sum of the instruction cycles, the data cache miss cycles, and the instruction cache overhead (about 1000 cycles). The number of instructions and the number of memory accesses grows with the square of the image size. This is as expected. However, there is an explosion in the data cache overhead for a matrix of size 256 x 256. This is because the inner loop accesses the array in both row and column order. Each access to a byte in a row of the array brings in 63 other bytes. For the column order accesses, the data is used only after a full iteration of the outer loop. For n=256, an iteration of the outer loop overflows the 16K data cache. Also, each access only fetches a byte, even though 32 bits are available. This means that 75% of the memory bandwidth is wasted. Memory bandwidth is the critical limiting factor of this application. Accesses have a latency of three cycles. Cache misses have a latency of about 11 cycles for the critical word and about 30 cycles for the whole line.

```
void transpose(char *in)
{
    int i, j, t;
    for (i=0; i< (1<<SIZE); i++)
        for (j=0; j<i; j++) {
            t = in[(i<<SIZE) + j];
            in[(i<<SIZE) + j] = in[(j<<SIZE) + i];
            in[(j<<SIZE) + i] = t;
        }
}
```

Figure 3-11 Iterative Matrix Transposition

Table 3-9 Performance Figures by Size

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	574	1271	205
32 x 32	2142	4191	476
64 x 64	8350	15408	1420
128 x 128	33054	59344	5929
256 x 256	131614	233232	983218

Divide and Conquer

Two problems have to be dealt with. In a case where both the number of cache misses and the number of instructions need to be reduced, you should address the cache issues first because reducing cache overhead requires rethinking the algorithm. Figure 3-12 shows a solution to the matrix transposition problem, using the divide and conquer approach.

Table 3-10 Performance Figures by Size

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	2534	2830	382
32 x 32	9654	10406	889
64 x 64	37718	40150	3380
128 x 128	149142	158006	11113
256 x 256	592045	627155	57520

```

void transpose(char * in, char * out, int step)
{
    if (step == 0) {
        int t = in[0];
        in[0] = out[0];
        out[0] = t;
    } else {
        transpose(in, out, --step);
        transpose(&in[(SIZE + 1) << step], &out[(SIZE + 1) << step], step);
        transpose(&in[1<<step], &out[SIZE<<step], step);
        if (in != out)
            transpose(&in[SIZE<<step], &out[1<<step], step);
    }
}

```

Figure 3-12 Recursive Matrix Transposition

The matrix is divided into four equal-sized squares. The two squares along the diagonal are transposed in place. The two other squares are interchanged and transposed. On the initial call, the entire matrix is transposed in place:

```
transpose(matrix, matrix, SIZE);
```

For the recursive step, the two squares along the x - y diagonal are transposed in place. The two squares along the other diagonal are interchanged and transposed.

The parameter `step` indicates the array dimensions as a power of two. This is a naive algorithm that simply recurses until a 1 x 1 matrix is found. Table 3-11 indicates performance figures for different image sizes. For a 16 x 16 matrix, there are about five times as many memory accesses and 2.5 times as many memory instruction accesses, compared to the iterative algorithm. However, the execution time is better for a 256 x 256 matrix because of better locality.

Using Custom Operations

The TM-1000 has instructions that merge and pack bytes in registers in parallel. You can apply one of these instructions in this case to speed up the manipulation of bytes that are packed into words. Imagine that our task is to transpose a four-by-four matrix.

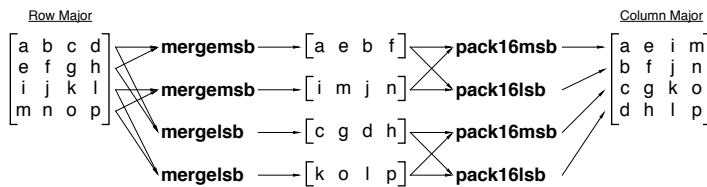


Figure 3-13 4 X 4 Transpose

Table 3-11 Performance Figures by Size

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	362	448	237
32 x 32	1218	1252	949
64 x 64	4466	4268	3028
128 x 128	17106	16274	9292
256 x 256	65871	65948	43819

Figure 3-13 shows how you can use custom operations. Figure 3-14 extends the solution to a $2^n \times 2^n$ matrix. The elementary step is on four machine words. Table 3-12 shows the performance of the routine. For a 256 x 256 array, the overall execution time is ten times less than the iterative algorithm. For a 16 x 16 matrix, the execution time is two times less.

```

#include <ops/custom_defs.h>
#define WSZ(SIZE/sizeof(int))
void transpose(int * in, int * out, int step) {
    if (step == 0) {
        int im0 = MERGEMSB(in[0*WSZ], in[1*WSZ]), im1 = MERGEMSB(in[2*WSZ], in[3*WSZ]),
            im2 = MERGELSB(in[0*WSZ], in[1*WSZ]), im3 = MERGELSB(in[2*WSZ], in[3*WSZ]);
        int om0 = MERGEMSB(out[0*WSZ], out[1*WSZ]), im1 = MERGEMSB(out[2*WSZ],
out[3*WSZ]),
            om2 = MERGELSB(out[0*WSZ], out[1*WSZ]), im3 = MERGELSB(out[2*WSZ],
out[3*WSZ]);
        out[0*WSZ] = PACK16MSB(im0, im1); out[1*WSZ] = PACK16LSB(im0, im1);
        out[2*WSZ] = PACK16MSB(im2, im3); out[3*WSZ] = PACK16LSB(im2, im3);
        in[0*WSZ] = PACK16MSB(om0, om1); in[1*WSZ] = PACK16LSB(om0, om1);
        in[2*WSZ] = PACK16MSB(om2, om3); in[3*WSZ] = PACK16LSB(om2, om3);
    } else {
        transpose(in, out, --step);
        transpose(&in[(SIZE + 1) << step], &out[(SIZE + 1) << step], step);
        transpose(&in[1<<step], &out[SIZE<<step], step);
        if (in != out)
            transpose(&in[SIZE<<step], &out[1<<step], step);
    }
}

```

Figure 3-14 $2^n \times 2^n$ Matrix

Inlining and Shrink-Wrapping

The algorithm reads and writes the entire matrix once. For a 256 x 256 byte matrix, there are 16,384 word accesses for reads and 16,384 word accesses for writes ($=256 \times 256/4$). A total of 32,768 memory accesses are necessary. The 65,871 memory accesses are necessary running the program. Most of the other 33,103 accesses are spills of registers to the stack. These are generated by **tmccom**, the TriMedia core compiler.

You can use two techniques to reduce the number of stack spills. Inlining one level of recursion reduces the function call overhead by a factor of about three. You can remove the spills for applications of the elementary step using a technique known as shrink-wrapping.

Shrink-wrapping works by splitting a function into two parts. The first part contains the part of the function that calls other functions or itself (the nonleaf part). The second part contains the part of the function that corresponds to a leaf. This code is placed in a separate function. The compiler can use caller-saved registers instead of caller-saved registers here because it is a leaf. Overhead is also reduced in the nonleaf part because only its variables need to be spilled. The function call overhead in the leaf part is minimal.

You can use different techniques to inline a function, including using a preprocessor such as KAP, using the C preprocessor, and hand inlining. Both using the C preprocessor and hand inlining were tried with this example. Naively using the C preprocessor gave poor code.

The elementary step in transpose calls `transpose_leaf`. It operates on an 8 x 8 matrix. The code in `transpose` is as follows:

```
if (step==1)
    transpose_leaf(in, out);
else { ..
```

Figure 3-15 shows source for `transpose_leaf`. Figure 3-16 shows the C preprocessor macros used to operate on 4 x 4 submatrices. `READ4` reads a submatrix into four temporary variables. `WRITE4` writes out the transposed result. `MERGE4` corresponds to the intermediary step.

Table 3-12 shows the performance of the routine after inlining and shrink-wrapping have been applied. For a 256 x 256 array, 8194 out of 40962 memory accesses can be attributed to spills. This corresponds to an overhead of 25%, which is acceptable. Eliminating spills divides the number of instructions by almost a factor of two.

Table 3-12 Performance After Inlining and Shrink-Wrapping

	Memory Accesses	Instruction Cycles	D-Cache Miss Cycles
16 x 16	262	294	207
32 x 32	768	684	1084
64 x 64	2834	2320	2757
128 x 128	10594	8864	9363
256 x 256	40962	36464	45202

```

void transpose_leaf(int * in, int * out) {
    int i0, i1, i2, i3;
    int im0, im1, im2, im3, im4, im5, im6, im7;
    int im8, im9, im10, im11, im12, im13, im14, im15;
    READ4(&in[0]); MERGE4(im0, im1, im2, im3);
    READ4(&in[1]); MERGE4(im4, im5, im6, im7);
    READ4(&in[4*WSZ]); MERGE4(im8, im9, im10, im11);
    READ4(&in[4*WSZ+1]); MERGE4(im12, im13, im14, im15);
    READ4(&out[0]); WRITE4(&out[0], im0, im1, im2, im3);
    MERGE4(im0, im1, im2, im3); WRITE4(&in[0], im0, im1, im2, im3);
    READ4(&out[1]); WRITE4(&out[1], im8, im9, im10, im11);
    MERGE4(im8, im9, im10, im11);
    READ4(&out[4*WSZ]); WRITE4(&out[4*WSZ], im4, im5, im6, im7);
    MERGE4(im4, im5, im6, im7);
    WRITE4(&in[1], im4, im5, im6, im7);
    WRITE4(&in[4*WSZ], im8, im9, im10, im11);
    READ4(&out[4*WSZ+1]); WRITE4(&out[4*WSZ+1], im12, im13, im14, im15);
    MERGE4(im12, im13, im14, im15);
    WRITE4(&in[4*WSZ+1], im12, im13, im14, im15);
}

```

Figure 3-15 Source for transpose_leaf

Table 3-13 compares the performances of the original and final versions of the program, in cycles. An instruction cache overhead of 1000 cycles in both cases is assumed. Depending on the size of the input, the improvement in performance varies between 1.6 and 16.

```

#define READ4(x)    i0 = (x)[0*WSZ]; i1 = (x)[1*WSZ]; i2 = (x)[2*WSZ]; i3 =
(x)[3*WSZ];
#define MERGE4(i,j) v0 = MERGEMSB(i0, i1); v1 = MERGEMSB(i2, i3); \
                    v2 = MERGELSB(i0, i1); v3 = MERGELSB(i2, i3);
#define WRITE4(x,i,j,k,l) (x)[0*WSZ] = PACK16MSB(i, j); (x)[1*WSZ] = PACK16LSB(i, j);\
                          (x)[2*WSZ] = PACK16MSB(k, l); (x)[3*WSZ] = PACK16MSB(k, l);

```

Figure 3-16 C Preprocessor Macros Used To Operate On 4 X 4 Submatrices

Table 3-13 Performance of Original and Final Versions

	Original Program	Final Program
16 x 16	2476	1504
32 x 32	5667	2675
64 x 64	16828	5419
128 x 128	65634	17410
256 x 256	1226450	74336

Cache Alignment

Cache accesses have a granularity of 64 bytes and are aligned at 64-byte boundaries in memory. Fetching a structure of 64 bytes aligned at 64-byte boundary requires a single cache access compared to two for an unaligned access. Fetching an unaligned 32-byte structure requires one and one half cache accesses on average, compared to one for an aligned access. If the matrix is allocated on the heap, it can be aligned to a cache boundary. The number of memory accesses increases from 44041 to 68162 for a 256 x 256 matrix if it is not cache-aligned. The TriMedia C library routine `_cache_malloc` can be used for this. Code for the transposition routine to align the matrix is shown below. The second argument is the set number (0-31, -1 means any cache set).

```
#define    LINESIZE    64
a = (char *)_cache_malloc(SIZE*SIZE , -1);
    < ... initialize matrix >
transpose((int *)a, (int *)a, STEP);
```


Chapter 4

Performance Analysis on the Hardware

Topic	Page
Overview	4-2
Terminology	4-3
Reasons for Long Interrupt Latencies	4-5
Clearing the IEN	4-6
Changing the Global Interrupt Priority	4-7
Individual Disabling	4-7
Preventing Task Preemption	4-7
Interrupt Latency Sampling	4-8
Using the Sampler	4-9
Detection of Latency Violators	4-9
Latency Sampler Code	4-10

Overview

Like in any other real time system, TM-1000 based multimedia applications are mostly driven by time critical events. Such events are passed between the application and its environment by means of interrupts, which often also announce or request data. For instance, an MPEG-2 decoder is continuously reacting on interrupts announcing new MPEG data to decode, on interrupts requesting new frames to display, on interrupts notifying that a VLD- or ICP operation or a DMA transfer has completed, or on interrupts providing real time synchronization.

For many interrupts it is extremely important that they are handled and acknowledged in time. There are a number of reasons for this: first, contrary to software, which can implement various buffering schemes to overcome transient timing problems (*jitter*), hardware is relatively simple in nature. When an interrupt is not served in time, input data might get lost or a device might go into error because it did not get instructions on what to do next. As second reason for timely handling interrupts, especially in high frequency systems: any delay in handling the interrupt reduces the time available for processing the related event, thereby increasing the probability of real time problems 'higher up' the chain.

The meaning of the term 'in time', and the severity of the 'real time problems' is strongly de-pendent on the application and the devices which it uses. For example, in video capturing, all timings are related to the input frame rate so that 'in time' will probably be in the order of magnitude of several milliseconds, which is in contrast to e.g. an ssi interrupt, which must be served strictly within a few hundred microseconds. Similarly, the penalty of occasional timing problems in displaying video might only be some short, hardly noticeable reduction in video quality, while an occasional timing problem in an audio renderer might enrage the listener.

This appnote deals with interrupt latencies, as being an important concept in application timing. It describes a mechanism to measure interrupt latencies, giving insight in the timing aspects of applications. It also describes how to find the cause of long latencies, and concludes with latency information of a number of Trimedia applications and libraries.

Terminology

The following terminology is related to interrupt handling and application timing:

- An *interrupt handler* is a parameterless C function that is triggered by an interrupt. It should be compiled with a `#pragma TCS_handler` or `#pragma TCS_interruptible_handler`, and installed as corresponding to a specific interrupt using the `tmInterrupts` functions in the TriMedia device library. The difference between these two pragmas is that the `TCS_handler` causes the interrupt enable bit to be cleared for the duration of the handler, thereby disabling nested interrupts, while a `TCS_interruptible_handler` runs with interrupts on.
- Decision trees (dtrees) are TM-1000 instruction sequences generated by the compiler which terminate in jump instructions (to the beginning of other dtrees). Interrupts will never take control during execution of a dtree. Instead, pending interrupts may take control only during jumps to other dtrees.
- More precisely, interrupts may take control only during the interruptible jump instructions generated by default by the TriMedia C compiler. Interrupt handling can be prevented even while jumping to other dtrees by using noninterruptible jump instructions. This special class of jump instructions is sometimes used in hand-coded assembly, e.g. to allow loop pipelining. See also TM-1000 Data Book, Chapter 3.
- *Grafting* is a technique, exploited by the TriMedia C compiler, to enlarge dtrees by merging it with copies of jump targets. It increases instruction level parallelism at the cost of (moderately) longer dtrees.
- The interrupt enable bit (IEN) in the TM-1000 processor status word (PCSW) determines whether asserted interrupts are kept pending, or lead to invocation of their interrupt handler at the next jump instruction. The IEN controls all interrupts of interrupt priority 6 and lower (see further). It has no effect on interrupts of priority 7.
- An interrupt priority is a number in the range 0..7 (on the TM-1000) assigned to each interrupt, and which controls the relative importance of the interrupt as follows. First, the hardware guarantees that, when multiple interrupts are pending at a particular jump instruction, an interrupt with highest priority value is selected for taking control. Second, the `tmInterrupts` functions of the TriMedia device library implements a scheme on top of the IMASK (see further) by which all interrupts of a specific priority or lower can be disabled “en masse” while leaving the higher priority interrupts enabled, by setting a global interrupt priority level.
- The IMASK is a bitvector on the TM-1000 by which interrupts can be specifically enabled or disabled. It should be accessed only via the `tmInterrupts` functions of the TriMedia device library. Contrary to the IEN bit, also interrupts of priority 7 can be disabled using the IMASK.
- Anon maskable interrupt (NMI) is an interrupt of priority 7. It is called this way because it cannot be disabled via the IEN. Because it is common practice to ‘disable all

interrupts' using the IEN, nonmaskable interrupts should only be used with extreme care.

- Disabling an interrupt is defined as any measure by which the interrupt's handler is prevented from taking control. Taking control is then postponed, and the interrupt remains pending. A particular interrupt is disabled (or masked, or blocked) during any of the following:
 - During execution of a dtree
 - During a noninterruptible jump
 - When the IEN is cleared and when the interrupt's priority is lower than 7.
 - When the corresponding bit in the IMASK is cleared. In terms of the tmInterrupts library, this is the case when the interrupt is not yet opened, or otherwise when the interrupt has been individually disabled or when the global interrupt priority level is larger than the interrupt's own priority.
- A *latency* of an interrupt is the difference in time between the moment at which the interrupt is asserted and the moment at which its handler starts executing. In other words, it is the time after asserting at which the interrupt handler is started. Any (noticeable) latency is caused by the application, by having disabled the particular interrupt.
- An *overrun* is a condition in which input data of a particular device (typically announced via an interrupt) is not timely consumed, and overwritten by subsequent data. Overruns are generally caused by interrupt latency problems.
- An *underrun* is a condition in which output data has not been given in time to a particular output device, causing the device to halt, or to continue with old, stale, or undefined data. Similar to overruns, underruns are generally caused by interrupt latency problems, for instance because response to a previous data request interrupt from the device was too late.

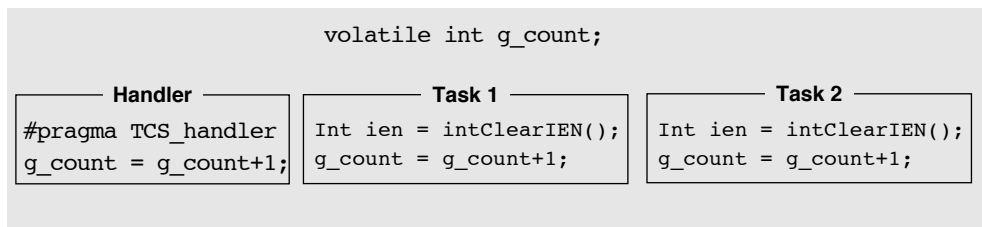
Summarized, using above terminology, longer latencies in an application may be harmful, since they reduce real time response. This may result in overrun errors of input devices, in which captured data is lost because the processor was notified too late to timely read it away and process it; or it may result in output device underrun errors in which no new output data has been made available in time because the processor has been too slow in reacting on a previous device notification.

Reasons for Long Interrupt Latencies

Interrupt latencies in the order of magnitude of about 10 microseconds and higher may theoretically be caused by long dtrees, especially in grafted code generated by the compiler.

However, it appears that such latencies very often are caused just by the application itself, by carelessly disabling and enabling interrupts.

Disabling all interrupts, or disabling one or several particular interrupts specifically, is generally applied to create critical sections for accessing global data structures which might also be accessed by interrupt handlers or by other tasks. This is best illustrated by means of a toy example:



In this example, several tasks and an interrupt handler each modify a 'global data structure', `g_count`. The classical problem is that this modification involves a read of the old value, followed by a write of a new (incremented) value, and that a race condition results when this sequence is interrupted by one of the others between the read and the write. The following interruptions are possible:

- Task1 by Task 2, due to a pSOS timer interrupt which ends Task 1's time slice in favor of Task 2,
- Task 2 by Task 1 in a similar way
- Task 1, or Task 2, by Handler due to occurrence of its interrupt

Both tasks and the handler prevent such interruptions from happening during `g_count`'s update by simply disabling 'all' interrupts; the tasks by calling the functions intended for this in the `tmInterrupts` library, and the handler by making use of compiler support via the `TCS_handler` pragma. This interrupt disabling is very effective, since it prevents time slicing because the pSOS timer interrupt is disabled, and it prevents the handler from interrupting the tasks and from interrupting itself (via a nested interrupt) because the handler interrupt is disabled.

Note that it is good practice to not simply enable the interrupts again at the end of a critical section which is started with a `intClearIEN`; rather, the old IEN should be restored because one can not always be sure that the interrupts were not already disabled. Also note that also handlers can create critical sections using `intClearIEN` and `intRestoreIEN`: use of

these functions, in combination with `pragma TCS_interruptible_handler`, allow finer grain interrupt disabling in longer interrupt handlers. Especially note the following:

WARNING

it is not disallowed to call other functions when interrupts are disabled, but never call a function which might deschedule the current task when running under a multitasking operating system like pSOS. ♦

Although interrupt disabling as described above is extremely effective for creating critical sections, it is also a very coarse method which should be avoided for critical sections longer than a few microseconds. The reason of this is that it might also lock out unknown interrupts with possible stringent latency requirements. Such an interrupt probably will not interfere at all with the critical section, and disabling it might unnecessarily increase its latency. The following sections go over the different mechanisms by which interrupts can be disabled. Some of these are more selective, and should be considered as an alternative.

Clearing the IEN

As mentioned above, disabling of an interrupt may be achieved by clearing the IE bit in the PCSW. Massively disabling all interrupts in this way, and later enabling them again is the usual way to achieve without much overhead a critical section in which a device, or global data structure can be accessed without the danger of a task context switch or a new intervening interrupt. Manipulating the IEN can be explicitly performed using the functions `intClearIEN`, `intSetIEN` and `intRestoreIEN` exported by the `tmInterrupts` device library. Two compiler- supported mechanisms provide an effect similar to clearing the IEN:

- Defining an interrupt handler as using a `pragma TCS_handler` (in contrast to a `TCS_interruptible_handler`). The generated code for such a handler clears the IEN at the start, to be enabled at the end of the handler.
- Defining a function or handler as a `TCS_atomic`. For these functions, the compiler will generate non-interruptible jumps.

NOTE

further that explicit use of non-interruptible jumps in handcoded assembly also locks out interrupts in a similar way. ♦

Changing the Global Interrupt Priority

Interrupt disabling can also be achieved by raising the global interrupt priority to a higher value. This mechanism is generally used in interrupt handlers, to let serving not be disturbed by ‘less urgent’ interrupts, while still allowing ‘more urgent’ ones. So while the IEN is generally used to achieve atomicity, disabling based on interrupt priority is used to (temporarily) allocate processor cycles only to a certain minimal urgency. Although similar to clearing the IEN, raising the interrupt priority might also lock out unknown interrupts, it selects on a notion of urgency and for this reason it is less likely that interrupts with stringent latency requirements will be involuntarily locked out.

The global interrupt priority can be modified by means of a call to `intSetPriority` from the `tmInterrupts` library.

Individual Disabling

Interrupts can also be individually disabled. For instance, using a call to `intInstanceSetup` from the `tmInterrupts` library, interrupt `intVIDEOIN` can be individually disabled; regardless of its priority, and it has no effect on other interrupts.

Preventing Task Preemption

Individual disabling, and raising the global interrupt priority level may be used to lock out all interrupts which might interfere with a particular critical section. However, it provides no control over task preemption. In other words, even with all “nasty” interrupts locked out, the current task might still be preempted by pSOS in favor of another which might enter the same critical section. Note that the actual problem here is that the identity of the pSOS timer interrupt and its priority are hidden.

Task preemption can be (temporarily) prevented by means of the pSOS function `t_mode`. This function does not disable any interrupt at all, but just prevents scheduling. In libraries or applications which may run either under pSOS or in stand-alone mode, it is better to use the `AppModel` functions, which can be used to abstract from the currently running operating system, as follows:

Task 1

```
#include <tmlib/AppModel.h>;
...
AppModel_suspend_scheduling();
g_count= g_count+1;
AppModel_resume_scheduling()
```

Task 2

```
#include <tmlib/AppModel.h>;
...
AppModel_suspend_scheduling();
g_count= g_count+1;
AppModel_resume_scheduling()
```

Interrupt Latency Sampling

Applications might encounter interrupt latency- related problems, even in case interrupts have been disabled with extreme care. Libraries might have to be certified on “decent” interrupt latency behavior. And both applications and libraries might be investigated on the (timing) effects of running them together with other applications or libraries. This section describes a sampling method which can be used for all these situations for gaining interrupt latency information. See also the source listing in Appendix A, and a toy demonstration program which samples latencies during generation of a cosine table, in Appendix B. This source is copied from a corresponding example provided with the TriMedia SDE, in `$TCS/examples/misc/latency_sampler`.

The latency sampling method records the interrupt latencies encountered by a periodic timer interrupt over a specified duration of time while the sampled application is running. Using a timer-based interrupt has the pleasant property that the times at which it is raised are known exactly (up to a few cycles), so that the latency can be easily obtained by subtracting this time from the actual time of handler invocation. The obtained timer interrupt latencies are recorded in a bucket array, where each bucket represents the number of timer interrupt latencies encountered during sampling. After termination of sampling, a latency histogram can be obtained by printing the values in the bucket array.

Although the latencies are measured for the timer interrupt only, they can be interpreted more generally: each measured latency would have been the latency of any interrupt which was also enabled at the moment at which the timer interrupt occurred. The sampler as shown installs the timer interrupt at (lowest) priority 0, and hence, for any interrupt i which is not individually disabled: i is enabled whenever the timer interrupt is enabled, and this means that at any moment, i 's latency is smaller than the timer interrupt latency. In other words, the measured interrupt latencies form a lowerbound, or worst case information, on the interrupt latency of any interrupt which is not individually disabled. This lowerbound could be tightened by running the timer interrupt at a higher interrupt level, thereby disregarding interrupt latencies encountered by non time critical interrupts.

In an application with one time critical, high priority interrupt, the sampled latencies are lowerbounds also in another sense: a measured latency could be caused by the high priority interrupt handler itself, because it was invoked at elapse of the sample timer. In this case the latency which was encountered by the timer interrupt would obviously not have been encountered by the high priority interrupt itself.

By the above, the described sampling method can be used to obtain information on interrupt latencies encountered by interrupts which have not been individually disabled.

Using the Sampler

Sampling can be performed simply by compiling and linking the listed C code to the application, and by calling function `init_latency` when sampling should start. This function clears the bucket array, allocates a timer, and sets it up to start sampling. After “some” time, sampling can be stopped by `term_latency`, which deallocates the timer and prints the histogram on the standard output.

Note that, being sampling based, the reliability of the obtained information is dependent on the sample frequency, the sample duration, and the code coverage of the application during sampling. For instance, no guarantee is given that the largest measured latency indeed is the theoretical worst case latency.

No analysis is made in this document on this reliability.

Detection of Latency Violators

The listed sampler can also be used to detect the causes of long latencies, as follows: upon any sampled latency larger than `NROF_BUCKETS * 2 LOGS`, the function `LATENCY_VIOLATOR_DETECTED` is called. This can be used to detect the part of the application which was responsible for this long latency, by placing a breakpoint in this function using the TriMedia debugger `tmdbg`. When hitting this breakpoint, the application completely stops with all interrupts disabled. A stack traversal will reveal the function which ended the violating critical section.

Latency Sampler Code

```

/*----- includes -----*/
#include <tml/tmTimers.h>
#include <tml/tmTimersmmio.h>
#include <tml/mmio.h>
/*----- local definitions -----*/
#define NROF_BUCKETS 1000 /* number of sample buckets */
#define LOGS 4 /* binary logarithm of sample bucket size */
#define SAMPLE_PERIOD 1000 /* cycles */
static Int buckets[NROF_BUCKETS];
static Int sample_timer;
static Int last_tick;
custom_op Int cycles(void);
/*----- utility functions -----*/
LATENCY_VIOLATOR_DETECTED()
{
intClearIEN();
/* Place a breakpoint here */
intSetIEN();
}
static void
sampler(void)
{
#pragma TCS_handler
Int now = cycles();
Int sample_timer_value = timGetVALUE(sample_timer);
Int this_tick = now - sample_timer_value;
Int latency = now - last_tick - SAMPLE_PERIOD;
Int bucket_nr = latency >> LOGS;
last_tick = this_tick;
if (bucket_nr >= NROF_BUCKETS) {
buckets[NROF_BUCKETS - 1]++;
LATENCY_VIOLATOR_DETECTED();
}
else if (bucket_nr < 0) {
buckets[0]++;
}
else {
buckets[bucket_nr]++;
}
}
Bool
init_latency()
{
timInstanceSetup_t setup;
if (timOpen(&sample_timer) != TMLIBDEV_OK) {
return False;
}
else {
memset((Pointer) buckets, 0, sizeof (buckets));
}
}

```

```

last_tick = cycles();
setup.source = timCLOCK;
setup.prescale = 1;
setup.modulus = SAMPLE_PERIOD;
setup.running = True;
setup.handler = sampler;
setup.priority = intPRIO_0;
timInstanceSetup(sample_timer, &setup);
return True;
}
}
void
term_latency()
{
Int i;
timClose(sample_timer);
for (i = 0; i < NROF_BUCKETS; i++) {
if (buckets[i]) {
printf(" %7d : %7d\n", i << LOGS, buckets[i]);
}
}
}
Appendix B: Sample Sampled Application
/*----- includes -----*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*----- functions -----*/
main()
{
int i;
init_latency();
for (i = 1; i < 1000; i++) {
printf("cos(%d)= %e\n", i, cos(i));
}
term_latency();
exit(0);
}

```


Appendix A

Shell Scripts

tmprof.select

```
sed -n -e "1,2p; /$1/p" <$2 >/tmp/ts.$$
shift
shift
tmprof $* /tmp/ts.$$
rm /tmp/ts.$$
```

select

```
sed -n -e "1,2p; /$1/p" <$2
```

