Book 4—Software Tools

Part A:

C Language Users Guide

TriMedia

Version 2.1

Table of Contents

Chapter 1

Introduction to C Language Users Guide

Introduction

The *C Language Users Guide* provides information on how to use the TriMedia C and C++ compilers running under the UNIX (Solaris or HP-UX) and Windows operating systems.

What This Guide Provides

The following material is provided in the *C Language Users Guide*.

- How to compile and link programs.
- How to control the behavior of the compiler during compilation.
- How the C and C++ languages accepted by the compiler compare with several industry standard definitions of C and C++.
- Which programming restrictions apply when the TriMedia compiler is used and how to deal with programs that violate these restrictions.

What This Guide Does Not Provide

The following material is not provided in the *C Language Users Guide*.

- How to write C/C++ programs in general. For that purpose, a C or C++ language reference manual should be consulted.
- Specifics of using the assembler and the link editor. For these, refer to the appropriate documentation.
- How to use the various aspects of UNIX indirectly associated with compiling and executing programs, such as:
  - — Preparing programs to be input to the compiler.
  - — Using the make program.

  For any of these topics, a suitable UNIX manual should be consulted
- Methodologies for developing in a Windows environment (for example, use of an IDE versus makefiles). For these topics, third-party software documentation should be used.
- How to execute the program at run time, including
  - — How to download the program on to the host processor.
  - — How to use the debugger.
  - — How to access facilities of the real time operating system (RTOS).
  - — Manipulating file output from program execution.

Many of the issues and optimizations discussed here have pros and cons. These are indicated when applicable.

Overall Structure

Figure 1 shows the TriMedia software development flow. The most common use of the tools follows the spine of the flowchart. Other elements reflect peripheral functions.

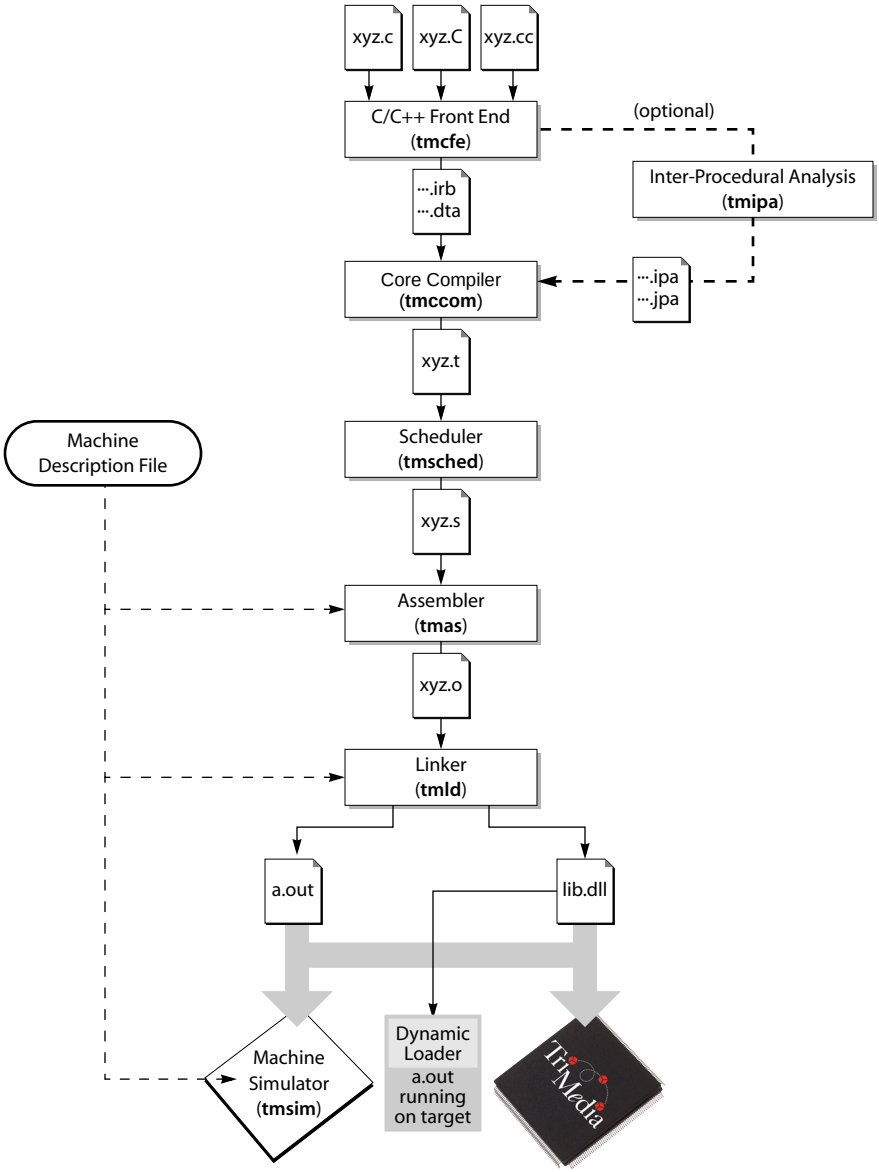

Figure 1 TriMedia Software Development Flow

The tools shown in Figure 1 operate as described below.

The compiler driver (**tmcc**) enables you to compile, assemble, and link modules in one step. See Chapter 2, *Using the C Compiler*, for more information.

The C compiler accepts C and C++ source code and produces trees code. The compiler is separated into a front end (**tmcfe**) and a core compiler back end (**tmccom**). The front end performs machine-independent processing and local optimization. The back end performs machine-dependent processing and global optimization. Between **tmcfe** and **tmccom**, there is an optional inter-procedural analysis phase **tmipa**. See Chapter 2, *Using the C Compiler*, for more information.

The Instruction Scheduler (**tmsched**) takes trees code as input. This code is unscheduled and temporary registers are not yet specified. The scheduler assigns registers and turns the code into parallel VLIW assembly. See Chapter 4, *Using the Instruction Scheduler*, for more information.

The Assembler (**tmas**) translates assembly programs and generates an object program in TriMedia's generic object format.

The Linker (**tmld**) combines object programs to create an executable file or an object program suitable for relinking. It performs link-time optimizations to improve the program. It supports both static and dynamic linking.

The Archiving Utility allows the user to build and manipulate libraries. Libraries are in UNIX library format ("ar" format).

The Downloader Library provides final downloading and patching for interfacing with the TriMedia host environment. It can be executed on either the host or TriMedia.

C Compiler Overview

Standards

C Language

The implementation of the C programming language is based on the ANSI standard for C as described in the ANSI and ISO C standards and by the IEEE standard for floating point.

The C compiler supports the concept of restricted pointers as proposed by the ANSI Numerical C Extensions group.

C++ Language

The parser of the TriMedia compiler is the EDG (Edison Design Group) front end, which tracks the evolving C++ standard. Use of the EDG front end guarantees TriMedia users that the definition of the C and C++ languages is as standard as possible.

Run-Time Support

All library functions conform to the ANSI C library standard. The C++ run-time support is based on the Modena library which tracks the evolving ANSI run-time library standard for C++.

Output Files

The following features pertain to output files created by the compiler assembly source output.

- The C compiler generates trees code.
- The instruction scheduler transforms this into VLIW assembly code.
- Both formats are user-readable.

Stabs Format

The assembly representation and the object file format use UNIX debugging representation stabs. Stabs allow source-level access from the debugger.

Object Format

The object format is generic across supported platforms and is endianness-independent. It is supported by the downloader library of the TriMedia run-time environment.

Compiler Interface

The following features pertain to interfacing with the compiler.

- The compiler tools include a driver that can be used to compile, schedule, assemble and link programs in a single step.
- UNIX tradition is followed for command-line options.
- TriMedia calling conventions are documented and straightforward for interfacing with assembler.
- Machine operations can be programmed using a mechanism called *custom-ops*. These permit efficient programming of multimedia algorithms in C.
- Exception and interrupt handlers can be programmed in C.

Implementation

The following features pertain to the implementation of the compiler.

- The C and C++ front ends are integrated with the preprocessor and parser.
- Stand-alone preprocessing is possible.
- Preprocessed input is accepted.

For more information, see *Options That Control Preprocessor Operation* in Chapter 2.

Optimization

The following features pertain to the optimizations of the compiler.

- Performance can be significantly improved by function inlining, loop unrolling, and grafting.
- The compiler integrates a full global optimizer, including alias analysis.
- Compiler optimizations are intelligently applied using information from profiling.
- The compiler and scheduler use advanced algorithms for granularity partitioning that are unique to TriMedia.
- Near optimal schedules are obtained for code segments of hundreds or thousands of instructions.
- Register allocation and scheduling are integrated in the compiler and scheduler.

For more information about the TriMedia compiler's optimization techniques, see Chapter 3, *Using the Optimizer*.

Environments and Compatibility

The following features pertain to supported environments and compatibility.

- On SPARC, the TriMedia compiler supports both Solaris 1 (SunOS 4.1.x) and Solaris 2.x.
- On HP-UX, version 10 of the operating system is supported.
- The Microsoft Windows version supports Windows 95 and Windows NT.
- The calling convention is compatible with version 1.1Z of the TriMedia SDE, except when the **-g** option is used.
- Assembler syntax and object format is compatible with version 1.1Z.
- The intermediate representation is compatible with extensions.

Compiler Architecture

The following features pertain to the architecture of the compiler.

- The compiler breaks up each procedure.
- The scheduler operates on individual decision trees.
- The compiler-scheduler concentrates on the global behavior and tries to obtain a breakup into decision trees that is as good as possible.
- The interface is a language called *trees code*.
- Object files and libraries (.o and .a files) compiled using **-g** are unique to the TCS release.
- Mixing other object files and libraries is not recommended.
- Both make use of the large register set.
- Trees code has run-time information for optimization.
- For speed, trees code can be programmed by the user.
- Trees code is as efficient as assembler, but easier to program and maintain.

Quality Assurance

The following features pertain to quality assurance.

- An extensive quality analysis (QA) process is applied to the compiler as part of product qualification process.
- QA includes functionality, performance, compatibility, and regression tests.
- The product is validated on the Windows, Solaris, and HP-UX platforms.

A Three-Minute Guide to the TriMedia Compiler

This section offers a quick introduction to the TriMedia compiler.

The compiler supports cross compilation. The compiler can be used on SPARC, for example, to produce an executable that is downloaded using the SDE on Windows. Program source, compiled assembly, and object code can safely be transported from one operating system to another. The object format is common across platforms and is independent of host endianness.

The TriMedia C compiler strongly follows UNIX tradition for compiler options and can accept **-c**, **-g**, **-o**, **-E**, **-I**, **-D**, **-L**, **-O***n*, **-S**, and **-U.** For areas where there is not an established UNIX tradition, options unique to the TriMedia compiler are used.

The program must be compiled with the **-g** option to use the debugger. The debugger has a Graphic User Interface (GUI), as well as everything necessary to control downloading

and execution. It allows access to the source program and machine resources. Using the debugger, the optimization level is limited to **-O1**.

The **-p** option can be used to produce a dtprof.out file on execution. This file can be used to improve loop unrolling, inlining, and grafting.

The **-ptm** option links the program with a library to produce a "mon.out" file. The **tmprof** utility reports on where the program spends its time. The **-g** and **-ptm** options cannot be used together.

For information on using the compiler, see the man pages for **tmcc**, **tmar**, as well as Chapters 2 and 3 of this manual.

The compiler driver **tmcc** can be used to compile assembly programs (.s files) and programs in trees format (.t files).

Overall control of optimization is done with the **-O***n* option, where **n** can range from 0 to 5. The **-O3** level is the default and is a very useful level of optimization. Levels **-O4** and **-O5** invoke inter-procedural optimization and require extra care in managing recompilation. If **-g** is specified, the optimization level is forced to **-O1**. Performance is often significantly better at **-O5** but code size typically increases.

To optimize a given program, we recommend compiling it initially with "typical" compiler optimization enabled. Specifically, build your program with **-O3**. Use the profiling options of the compiler (**-p, -g -r**) to get a first idea of the effectiveness of grafting and experiment with inlining. Loop unrolling is applied automatically. Use the **-ptm** option to find out where your program spends its time. The **tmsize** command will tell you the code size (**tmsize name.o**).

The startup code and I/O depend on the host; the **-host** option to **tmcc** specifies the host (**-host serial**, **-host Win95**, **-host WinNT**, **-host tmsim**, **-host nohost**). The **-target** option specifies the TriMedia processor being used (**-target tm1**, **-target tm2**).

The default TriMedia C mode is ANSI-compatible with some relaxed requirements. TriMedia C++ offers several modes for dealing with the differences between **cfront**-like C++ and ARM or ANSI C++. The default mode is ANSI-compatible with a number of extensions.

The TriMedia compiler requires that certain restrictions on programming usage (as specified in the ANSI C and C++ standards) be met to apply full optimization. Some programs contain latent violations of these ANSI restrictions and may fail when high degrees of optimization are applied. A systematic process of fixing or working around such violations may then be necessary.

Unix-based **tmcc** uses temporary files in the directory defined by **P_tmpdir** in stdio.h. Windows-based **tmcc** uses temporary files in the directory specified by environment variable TMPDIR (or in the current directory if TMPDIR is not set). Changing the location of temporary files by specifying TMPDIR can be very useful. For instance, if the source files are on a slow network disk, you can specify TMPDIR to keep temporary files on a fast local disk.

Chapter 2

Using the C Compiler

Introduction

This chapter provides all the information needed to compile and link programs.

The **tmcc** compiler driver (**tmCC** or **tmcpp** for C++) provides one-point coordination of all the compiler phases. The compiler phases are as follows:

- To compile and link a C or C++ source program, the C front end **tmcfe** is run. An integrated preprocessor is incorporated into **tmcfe**.
- The back end **tmccom** applies optimizations and produces trees code.
- The Instruction Scheduler accepts trees code and produces VLIW assembly code.
- The Assembler generates an object file in TriMedia's object format.
- The Link Editor links relocatable object files into a single executable file. Both dynamic and static linking are supported. The linker can also be invoked separately.

This chapter provides documentation of the most common compiler options. Options related to optimization are discussed in Chapter 3, *Using the Optimizer*.

Invoking the Compiler Driver for C

The compiler driver for C is called **tmcc**. To invoke **tmcc** for a C program, enter

```
tmcc [options .. ] file names ...
```

tmcc	Command that coordinates invocation of the compiler components.
options	Options that affect the operation of the components.
file names	One or more C source files, trees code source, assembly source, or object files..

For example, if you want to compile two files named main.c and fibonacci.c and produce an object file fibonacci, you enter

```
tmcc main.c fibonacci.c -o fibonacci
```

As **tmcc** encounters each source file, it prints the C file name followed by a colon. The example below uses the **–V** option of **tmcc** to print version information.

```
tmcc -V main.c fibonacci.c -o fibonacci
tmcc: V5.4.9 of tcs2.00031Win95
main.c:
fibonacci.c
```


Source files and object files are distinguished based on the suffix (.c, .o), as follows.

Suffix	Description
.c	A C source file.
.i	cpp output file
.t	tmccom output file
.s	tmsched output file
.o	tmas output file
a.out	tmld output file. The file name is always "a.out" unless you specify another output file name.

The files need not be in the current directory.

Invoking the Compiler Driver for C++

The compiler driver for C++ is called **tmCC**. To invoke **tmCC** for a C++ program, enter

```
tmCC [options .. ] file names ...
```

tmCC	Command that coordinates invocation of the compiler components.
options	Options that affect the operation of the components.
file names	One or more C source files, C++ source files, trees code source, assembly source, or object files.

For example, if you want to compile two files named main.C and fibonacci.C and produce an object file fibonacci, you enter

```
tmCC main.C fibonacci.C -o fibonacci
```

To aid those who work in environments where the command line is not case-sensitive, a alternate driver **tmcpp** is available. You can invoke **tmcpp** exactly the same way you invoke **tmCC**. For the above example, enter

```
tmcpp main.C fibonacci.C -o fibonacci
```

C++ source files are distinguished by the extensions .C, .cc, and .cpp. Other suffixes are common.

tmcc, **tmCC**, and **tmcpp** are, in fact, the same program. The name **tmCC** (or **tmcpp**) is used so that C++ knows the appropriate run-time library to link. The options for **tmcc**, **tmCC**, and **tmcpp** are the same.

Note
In the remainder of the chapter, the term **tmcc** is used interchangeably for both compilers, unless specified otherwise. Use **tmCC** or **tmcpp** at link time to specify the C++ library for object files.

Using the Compiler

Compiler behavior can be controlled at several levels:

- Compiler driver options allow control over the overall process of compilation.
- Phase-specific options allow control over the behavior of the front and back ends.
- Instruction Scheduler-specific options allow control over the back end of the C and C++ compilers.
- Source language pragmas allow control at the file, routine, or statement level.

Compiler-specific options related to optimization are defined in Chapter 3, *Using the Optimizer*.

Instruction Scheduler options are defined in Chapter 4, *Using the Instruction Scheduler*.

Predefined macros, language pragmas, and compiler options other than optimization are defined in this chapter.

Compiler Driver Options

The default behavior can be changed and/or extended in various ways by the use of options. The options described below can be given. Files and options that are not recognized are passed through by the driver to the Link Editor.

Options That Help Understand Compiler Operation

The following options help understand the compiler's operation and also what compiler you are using.

Option	Description
-h	Help. Prints help message and exits. Try: tmcc -h. The -? option is equivalent to the -h option.
-V	Version. Prints tmcc version number, as in **tmcc**: V5.4.9 of tcs2.00031Win95 where: V5.4.9 is a version number internal to **tmcc** tcs2.0 corresponds to this release (V2.0) 0031Win95 is the build target and number
-v	Verbose. Prints name and arguments of each executed phase.
-K	Keep. Keeps intermediate files. tmcc generates intermediate files with extensions as given above rather than using temporary files to pass information between phases.

The **-v** and **-K** options are useful in combination. The **-v** option prints the name of each pass as it executes. The **-K** option keeps the temporary files in the current directory. For example,

```
tmcc -v -K t.c
```

produces the following (the output has been extensively edited):

```
tmcfe -Xc=cp -XYc=mixed -b ... -O3 -eb -target tm1 t.c
```

The C compiler front end is invoked with options that define the configuration. The output is produced in three temporary files (**-b** options).

```
tmccom -b ... -o t.t -O3 -eb -target tm1
```

The C compiler back end is invoked. Trees code output is produced in the file t.t. This is because of the **-K** option.

```
tmsched -o=t.s -eb C:\TriMedia\lib\tm1.md t.t
```

The Instruction Scheduler is invoked. Assembly source is produced in t.s. This is because of the **-K** option. The directory C:\TriMedia is the installation directory.

```
tmas -o=t.o -eb C:\TriMedia\lib\tm1.md t.s
```

The assembler is invoked to produce an object file t.o.

```
tmld -o=a.out -btype boot -exec ... t.o -start=__start ...
```

The Link editor is invoked. The arguments (not shown) correspond to startup files, configuration information, and libraries.

Pros

- The **-V** option is useful for providing information to TriMedia technical support. All tools in the SDE support it.
- The **-K** option provides easy access to compiler intermediate files.
- The **-v** option is useful for seeing where time is being spent, and for understanding **tmcc** operation.

Cons

- The **-v** option produces verbose output.
- The **-K** option clutters the current directory.
- The **-K** option does not produce a .i file. Use **tmcc –P**.
- The intermediate files between **tmccom** and **tmcfe** are binary and are not produced with **-K**.

Options That Stop Compilation After a Particular Phase

The following options stop compilation after a specified phase. The **-c** (lower case) and **-S** (upper case) options are the same as in UNIX compilers.

Option	Description
-t	Trees. Compiles to intermediate format (.t) only. The default output file name is "name.t."
-S	Schedule. Compiles and schedules but does not assemble. The default output file name is "name.s."
-c	Compile only. Compiles to .o, but does not link. The default output file name is "name.o."
-Qphase	Quit. Quits after the specified phase, which should be one of **tmcfe**, **tmipa**, **tmccom**, **tmsched**, **tmas**, or **tmld**.

See *Implementation Specifics* in Chapter 8 for differences in code generation when the **-t** option is specified.

Pros

- You can get just one intermediate file (.t, .s) without cluttering.
- Breaking the compilation into phases is good for makefiles and saves compile time.
- The **-S** and **-c** options are UNIX-compatible.

Cons

- Two commands are necessary for a compilation.
- Two commands are necessary to get two intermediate files, as compared to one command with **-K**.
- When using inter-procedural analysis (**-04**, **-05**), there may be some dependencies between files.

Options That Produce More (or Less) Information

The following options have an incidence on the errors and warnings produced by the compiler and on the format of the executable.

Option	Description
-w	Warnings. Suppresses warning messages.
-g	Debug. Produces debug information. This may enable other special compilation conditions (use **tmcc** -v). The debug information produced is in stabs format
-p	Profile. Generates profiling information to file dtprof.out. This option is for the compiler to guide optimization, specifically loop unrolling, function inlining, and grafting. The dtprof.out file is used by the **tmcc** -G and -r options.
-ptm	Profile. Generates profiling information to file mon.out. This option is for the user, to find out what the program is doing. The mon.out file from -ptm is used by tmprof(1).
-R	Report log information from the instruction scheduler tmsched in file.schedlog.
-Xdiag=n	Set diagnostic level to n. -Xdiag with no level specified means -Xdiag=2

Diagnostic levels are defined below. If no option is specified, the default level is 1.

Level	Description
0	Errors and fatal errors .
1	Warnings, errors, and fatal errors.
2	Remarks, errors, and fatal errors.

There is a single level of warnings.

Pros

- Compiling with **-w** allows older C code containing non-portable constructs to be compiled without excessive warnings.
- Compiling with **-p** allows the subsequent speeding up of the program. This requires a second compile step.
- Compiling with **-ptm** allows you to understand where the program is spending its time.
- Compiling with debug information allows full debugging.
- Access to the source is possible during debugging.

Cons

- With the **-w** option, important warnings may be lost.
- Run-time overhead is added with **-p**, **-ptm** and **-g**.
- All of these options increase the size of the executable.

Options That Control Preprocessor Operation

The C compiler includes an integrated preprocessor for macro definitions and file inclusion (**#define**, **#include**). The following options control the operation of the preprocessor. It can be invoked so that it only does preprocessing. These options are UNIX-compatible.

Option	Meaning
-Dname[=value]	Define. Defines macro name to the C front end, **tmcfe**, with the specified value, if given.
-E	Expand. Runs the C/C++ front end, **tmcfe**, only and prints output to the standard output.
-I<path>	Include. Passes the given include path to the C/C++ front end, **tmcfe**.
-P	Preprocess. Runs the C/C++ front-end **tmcfe** only. Places the output in name.i.
-U<name>	Undefine. Undefine macro name to the C front end, **tmcfe**.
-nostdinc	Do not search the standard include for standard C headers. The **-nostdinc** option removes dependencies on system headers. This is useful for stand-alone applications. However predefined macros are not affected (see below)
-Xinclpath=absolute	When processing a **#include**, look in the directory of the original source files.
-Xinclpath=relative	When processing a **#include**, look in the directory of the file that contains the **#include** statement.

- The **-Xinclpath** option applies to the processing of include files with relative path names.
- **-Xinclpath=relative** is the default behavior.

  Note
  Preprocessing (the **-E** and **-P** options) eliminates dependencies on makefiles on your system. This is very useful when submitting a test case to TriMedia technical support.

- The **-g** option cannot be used in combination with these optimizations:

  -G, **-graft**, **-if_convert**, and **-full_if_convert**.

- If you attempt to pass **-D** arguments that include quote marks, be aware that double quotes and backslashes are interpreted by the shell (DOS command.com or MKS ksh) before being passed to the invoked **tmcc** command. For example,

```
$ tmcc -c -DFNAME=\\\"test.h\\\" prog.c          [DOS shell]
```

or

```
$ tmcc -c '-DFNAME=\\\"test.h\\\"' prog.c        [MKS ksh]
```

have the same effect as adding

```
#define FNAME "test.h"
```

to program prog.c. Note that the backslash and the double quotes must be escaped for the DOS shell and then single quotes are added for MKS ksh.

Options That Control Optimization

The following is a summary of compiler options that affect optimization. A complete list is given in Chapter 3, *Using the Optimizer*.

Option	Description
-G	Grafting. Reads program profile information from file dtprof.out.
-Xunroll=n	Performs loop unrolling with unroll factor of **n**.
-Xunroll=0	Disables loop unrolling.
-Xunroll=1 **-Xunroll**	Automatic unroll factor is computed.
-p	Profile. Generates profiling information to file dtprof.out.
-r **-r="file"**	Reads profiling information from file dtprof.out (or from **file**, if specified. This option controls function inlining, loop unrolling, and grafting.
-O0	No optimizations.
-O1	Local (per basic block) optimizations only. Variables in stack.
-O2	Local (per decision tree) optimizations. Variables in registers.
-O3	Global optimizations.
-O4	Interprocedural analysis and inlining.
-O5	More extensive inlining and global optimizations.
-Xalias=level	Define degree for alias analysis. Level is from **0** to **5**.
-A[012]	Define degree (0, 1, or 2) for unsafe alias analysis.
-Xmemlimit=n	Memory requirement in megabytes.

O[<n>] passes optimization level **n** to the C compiler **tmccom**. The default optimization level is **-O3** without **-g**, and **-O1** with **-g**. Unrolling is activated by default at this level.

A profiled library can be compiled (**tmcc -r -c**) and then linked separately.

Pros

- More optimization and more alias analysis speed up the program.

Cons

- Optimization slows down compile time.
- Inlining, unrolling, and grafting increase the size of the program.
- Unsafe alias analysis can cause problems with some programs.

Options That Control the Link Editor

The compiler driver invokes the Link Editor, **tmld**, by default. The following options are passed to **tmld**.

Options	Description
-nostdlib	Does not link with any library except those specified explicitly.
-nocompact	Disable linker reordering optimizations.
-b [**deferred** | **download** | **embed** | **export** | **immediate** | **type** | **embed** | **compact** | **foldcode** | **removeunusedcode**] **arg**	Passes linker-specific option to the **tmld** link phase.
-l<file>	Library. Passes library libfile.a to the linker **tmld**.
-L<path>	Library path. Adds the given path to the library path, which tmcc searches for libraries specified with -l options.
-o file	Output. Renames output to file.
-partial	Partial. Performs partial linking. Partially linked files may be used in subsequent linking.
-s	Strip. Strips the generated executable with **tmstrip**.

Cache performance can be affected by linker reordering.

Options That Define the Compilation Target

The following options define the machine being compiled for. In TriMedia terminology, the *target* processor is the TriMedia processor that executes the program. The *host* processor provides the execution environment (access to input/output). The *board* means the TriMedia circuit board, including daughter boards. Refer to the information about board support packages in Book 3, *Software Architecture,* Part A, for more information..

Options	Description
-target <mach>	Generate code for tm1 or tm2.
-host <mach>	Generates an executable suitable for the given host. The parameter can be **Win95**, **WinNT**, **MacOS**, **Solaris**, or **nohost**.
-e[bl]	Generate big-endian or little-endian code. The user must be careful to use the same endianness in all compilation and simulation phases.
-board=<path>	Specify a board support package for the target. The default reverts to a list of known boards.
-Xalign	Align arrays on word boundaries.

Options That Determine the C Language Dialect

Three dialects of C are supported by TriMedia compilers: strict ANSI, Kernighan and Ritchie (K & R) C, and ANSI C with extensions. These last two dialects are for compatibility with older compilers.

Option	Description
-Xc=ansi	C dialect is ANSI C. The compiler complies completely with the ANSI and ISO C standards (ANSI X3.159-1989 and ISO/IEC 9899:1990(E)) as a "conforming hosted implementation." That is, it supports all of the language, standard header files, and run-time environment.
-Xc=knr	C dialect is K&R C. This is closely compatible with the UNIX **pcc** compiler.
-Xc=mixed	C++ dialect is ANSI C with extensions. This eases the job of porting K&R C code to TriMedia.
-Xchar	Signedness of plain char.
-Xsizet	Definition of type **size_t**.

Option	Description
-Xwchart	Definition of type **wchar_t**. Normally, **wchar_t** is defined in a system include file.
-B	Accepts C++ style online commenting (delimited by "//") in C sources.
-varargs	Passes **-varargs** option to **tmccom**. This option must be used when compiling a source that includes **varargs** functions. (Two mechanisms exist in C to specify functions with a variable number of arguments, **stdargs** and **varargs**. Because functions with a variable number of arguments are not explicitly declared, the varargs mechanism is not portable. The newer mechanism is **stdargs**.)

- An additional value **noknr** can be added to the mixed or ANSI C modes. For example

```
-Xc=mixed+noknr
```

- The **-Xwchart** and **-Xsizet** options change the compiler's built-in expectations. Normally, **size_t** and **wchar_t** are defined in system include files. Possible values for **wchart** are **uint**, **ulong**, **ushort**, **uchar**, **int**, **long**, **short**, **char** or **schar**. Possible values for **sizet** are **uint**, **ulong**, **ushort**, **uchar** (**sizet** is not allowed to have signed types).
- Additional values **const**, **volatile**, and **signed** may be added in K&R C mode. For example, **-Xc=knr+const** specifies K&R C with support for the **const** keyword.
- An additional value **inline** can be given with all C modes, to disable **inline** as a keyword. For example, use **-Xc-=inline** to disable the **inline** keyword.
- The default is **-Xc=mixed+inline**.

Pros

- Use of the less restrictive modes (**-Xc=mixed**, **-varargs**) allows for easier porting.
- The **-B** option allows an easy way of commenting out code sections.

Cons

- Strict ANSI compatibility eliminates possible errors and guarantees portability.
- Code generated using **varargs** is less efficient.

Options That Determine C++ Language Dialect

Since the ANSI C++ standard is fairly recent, the C++ compiler allows for several anachronisms by specifying a specific dialect. The following table lists the options that influence the behavior of the compiler with regard to the C++ language.

Option	Description
-Xc=arm	C dialect is as per Annotated Reference Manual (ANSI), the most recent definition of the language.
-Xc=cp	ANSI C dialect, with extensions for older programs. Allows for several anachronisms.
-Xc-=rtti	Disables RTTI (Runtime Type Identification) keywords. RTTI keywords are on by default with all C++ modes and must be specifically disabled by using the option **-rtti**. For example: **-Xc-=rtti** or **-Xc=cp-rtti**.
-Xc+=c_func_decl	Relaxes prototype requirements for extern "C." This option relaxes the prototype requirements of the C++ language to those of the C language for functions declared within an extern "C" block. This value is not meant for direct use but to enable use of C style system include files with C++.

- Programs that compile under both **arm** and **cp** modes will behave identically.
- By default, the compiler recognizes array new and array delete operators. To disable this feature, use **-Xc-=array_nd** or **-Xc=arm-array_nd**, for example.
- By default, **wchar_t** is defined as a distinct built-in type. To define as a typedef, use **-Xc-=wchar_t**, for example. This is different from **-Xwchart**, which defines the type to use.
- By default, **bool** is recognized as a keyword. Use **-Xc-=bool** to disable this feature.
- By default, data structures are generated, and you are protected, if other code throws an exception. Use **-Xc-=exceptions** in an exception-free environment

The following preprocessor macros are defined for type definitions.

wchar_t	bool
_WCHAR_T	_BOOL_DEFINED
__WCHAR_T_IS_KEYWORD	__BOOL_IS_KEYWORD

These can be used to protect your own definition of bool, as shown below:

```
#ifndef _BOOL_DEFINED
typedef unsigned char bool;
#define _BOOL_DEFINED   1
#endif
```

Pros

- Use of the less restrictive modes (**-Xc=cp**) allows for easier porting.

Cons

- Strict ANSI compatibility eliminates possible errors and guarantees portability.
- Objects compiled with stdlib are generally not compatible with objects compiled without **stdlib**.

Options That Control Template Instantiation

This section explains how to control the **-Xtmpl** command-line option for template instantiation. Possible values are given in the table below.

Option	Description
-Xtmpl=none	Does not generate code for needed instantiations.
-Xtmpl=used	Generates code for needed instantiations. Declare as **extern**.
-Xtmpl=local	Generate code for needed instantiations. Declare as **static**. This is the default value.

Suppose a template is used in two source files (main.C and f.c) and is defined as shown:

```
template <class TYPE>

void copy(TYPE a[], TYPE b[], int n){
   for( int i = 0; i < n; i++)  a[i] = b[i];
}
```

- Compiling both files with **-Xtmpl=used** generates a symbol redefinition error from **tmld**.
- Compiling both files with **-Xtmpl=local** generates an extra copy function.
- Compiling one file with **-Xtmpl=none** and the other with **-Xtmpl=used** eliminates the redundant code.

Options That Control Floating Point Operations

This section explains how to control the **-dirty_float** and the **-uselongdoub64** command-line options. Possible values are given in the table below.

Option	Description
-dirty_float 0	Do no optimization on floating point expressions.
-dirty_float 1	Introduce **if** conversions and ignore the effects on the PCSW.
-dirty_float 2	Do tree height reduction on floating point expressions to increase parallelism. Rounding errors can be introduced at level 2.
-uselongdouble64 -useld64	Use 64-bit long doubles (not 32-bit).

- The **-dirty_float** option must be bracketed with **–tmccom** –– (i.e. this option must be given as **–tmccom -dirty_float N** –– on the command line.)
- **-dirty_float 0** is the default option.
- With **-uselongdouble64**, the compiler still generates TriMedia hardware floating point operations for float and double floating point arithmetic operations, but it uses a software library to perform long double arithmetic operations. Thus, long double operations are more accurate but considerably slower than than the corresponding float or double operations.

Predefined Macros

The C/C++ front end always defines the standard macro **__STDC__** (to indicate ANSI/ISO C Standard compliance). When compiling a C++ source file, it also defines **__cplusplus** to indicate C++ compilation.

In addition, the TriMedia compiler **tmcc** driver instructs **tmcfe** to define the standard macro **__TCS__** and either **__BIG_ENDIAN__** or **__LITTLE_ENDIAN__**, depending on the endianness of the compilation. These macros may be used in preprocessor directives to control conditional compilation of sources.

The following macros are always defined by **tmcc**.

Macro Name	Data Type	Meaning	Example Value
__DATE__	char *	date compiled	Mar 20 1998
__TIME__	char *	time compiled	17:00:51
__FILE__	char *	current src file name	foo.c
__LINE__	int	current line number	42

Macro Name	Data Type	Meaning	Example Value
__TCS__	int	TCS compiler	1
__STDC__	int	ANSI C	1
__TCS_V2__	int	Version of TCS compiler	1

These macros are conditionally defined by **tmcc**, based on **tmcc** options used.

Macro Name	Data Type	Meaning	Example Value
__BIG_ENDIAN__	int	if -eb	1
__LDBL_DP__	int	if -uselongdouble64	—
__LDBL_LIBC__	int	if -uselongdouble64	—
__LITTLE_ENDIAN__	int	if -el	1
__TCS_Win95__	int	if -host Win95	1
__TCS_WinNT__	int	if -host WinNT	1
__TCS_tmsim__	int	if -host tmsim	1
__TCS_nohost__	int	if -host nohost	1
__TCS_tm1__	int	if -target tm1	1

TCS Specifics

The macro **__TCS__** is always defined in the tmconfig file. **tmcc** accepts the **-host <host>** and **-target <target>** options and accordingly defines the macros **__TCS_<host>__** and **__TCS_<target>__**. The defaults for the host and target are **tmsim** and **tm1**, respectively. Thus by default the two macros **__TCS_tmsim__** and **__TCS_tm1__** are predefined.

For little endian compilation, the macro **__LITTLE_ENDIAN__** is defined. For big endian compilation the macro **__BIG_ENDIAN__** is defined.

For compilation with **-uselongdouble64**, the compiler defines the macro **__LDBL_DP__** to indicate that it represents long doubles using IEEE 754 double precision representation. It also defines the macro **__LDBL_LIBC__** to indicate that the standard library supports long double versions of some <math.h> and <stdlib.h> routines, as described in Chapter 9, *Library Functions*.

C Language Pragmas

Supported pragmas are the *handler* pragmas (see Chapter 10, *Porting and Optimizing Programs,* of Book 2, the *Cookbook*, Part D) for interrupt service or exception handling routines, optimization level pragmas, unsafe or safer alias analysis pragmas, dirty float pragmas, grafting pragmas, an atomic function pragma, and a caller save pragma.

Except for the handler pragmas, pragmas are basically used to overwrite command-line options for particular functions. They apply to the function in which they are specified.

If more than one pragma is specified to overwrite the same option, the last one is taken. ***Pragmas specified outside the function scope have undefined result.*** The exact names and semantics of supported pragmas are described in Table 1.

Table 1 Supported Pragmas

Name	Semantics	Scope
TCS_handler	Function should be compiled as non-interruptible interrupt service routine. The IEN bit is used to mask interrupts.	Function
TCS_interruptible_handler	Function should be compiled as interrupt service routine. Interrupts are not masked.	Function
TCS_exception_handler	Function should be compiled as exception handler.	Function
TCS_O0,TCS_O1,TCS_O2,TCS_O3,TCS_O4,TCS_O5	Overwrite the optimization level for this function only.	Function
TCS_A0, TCS_A1, TCS_A2	Overwrite the unsafe alias analysis for this function only,. See *Unsafe Alias Analysis,* beginning on page 53.	Function
TCS_graft, TCS_no_graft	Graft or do not graft this particular function. **TCS_no_graft** has higher precedence than the graft parameters from a graft-tuning file.	Function
TCS_caller_save, TCS_no_caller_save	Use or do not use caller save register for this function. Note: This will only have effect when the compiler would have decided to use or not use caller save register based on the command-line or default option.	Function
TCS_dirty_float0, TCS_dirty_float1, TCS_dirty_float2	Control the level of dirty float optimizations.	Function
TCS_break_dtree	End a dtree. This is useful in very large linear pieces of code where possibly scheduled code is more optimal on smaller dtrees, or when guaranteeing interrupt latency. When using the debugger, the breakpoint for the dtree created by the TCS_break_dtree pragma should appear on the next line of executable code.	Line

Table 1 Supported Pragmas

Name	Semantics	Scope
TCS_atomic	Function should be compiled with noninterruptible jumps only. Calling other functions is only allowed when these other functions are also atomic and a command-line switch **allow_atomic_calls** is given. The function executes in scheduler registers.	Function
-allow_atomic_calls	Functions compiled with the **TCS_atomic** pragma can call other functions. The user has to guarantee that the callee is also atomic. By default, atomic functions are not allowed to call any function. Note that some C-operators are mapped to non-atomic function calls, like the integer division '/'.	Function
TCS_inline=func;func…	Inline the specified routine (source or library intrinsic function).	Line
TCS_noinline=func;func…	Do not inline the specified routine.	Line
TCS_inllev=<n>	Specifies inlining level for ordinary routines. Level is 0 through 5, with an increasing level of inlining.	File
TCS_char=signed	In C/C++, char is signed by default.	File
TCS_char=unsigned	In C/C++, char is unsigned by default.	File
TCS_deflib=<n>	Specifies inlining level for library intrinsic functions. A higher level means more aggressive inlining.	File
TCS_domain=0	Applies the main optimizations to each outermost loop separately.	Routine
TCS_diag=<n>	Sets diagnostic level (0 to 2). Corresponds to **-Xdiag** command line switch.	Line
TCS_safeintr	Intrinsic library functions can be assumed not to modify external variables (for example, errno).	Loop
TCS_tmpl=none	Do not instantiate C++ templates.	Compilation
TCS_tmpl=used	Creates all instantiation that are needed. If templates are used across multiple source files, this may cause errors for symbol redefinition.	Compilation
TCS_tmpl=local	Templates are created as local variables or functions. This increases code size, but avoids the possibility of symbol redefinitions.	Compilation

Table 1 Supported Pragmas

Name	Semantics	Scope
TCS_unroll=0	Does not unroll loops.	Loop
TCS_unroll=1	Enables automatic inlining.	Loop
TCS_unroll=<n>	Unrolls loops <n> times.	Loop
TCS_unrollexact=0	Does not make assumptions about the iteration count of loops.	Loop
TCS_unrollexact=1	The iteration count of loops. Can be assumed to be a multiple of the value of **TCS_unroll**.	Loop
TCS_align	Align arrays to word boundaries	Compilation

Chapter 3

Using the Optimizer

Introduction

The TriMedia compiler integrates a state-of-the-art global optimizer.

This chapter tells you how to control the optimizer's behavior and what optimizations are applied by the compiler. The optimizer's behavior can be controlled at several levels.

- The optimization level can be set.
- Loop unrolling can be enabled automatically or manually.
- Inlining can be enabled automatically or manually.
- Memory aliasing overhead can be reduced with pragmas or the qualifier restrict.
- Source-language pragmas allow control at the file, routine, loop or statement level.

Using the optimizer is straightforward.

- Most optimizations are controlled by the overall optimization level (**-O4**, **-O5**).
- Information from the program's behavior is used.
- Trade-offs between code size and speed are applied.

Refer to Chapter 10 of Book 2, the *Cookbook*, to find out what parts of your program will benefit the most from optimization.

Controlling the Overall Level of Optimization

The most important control you have over compiling is to set a value together for a whole group of optimizations. The table below briefly describes the optimizations that take place at each level.

Level	Description
-O0	No optimization.
-O1	Local optimization.
-O2	Variables in registers.
-O3	Increased global optimization.
-O4	Inter-procedural global optimization, inlining.
-O5	Increased inter-procedural global optimization, inlining.

Each optimization level includes all the optimizations of lower levels.

Optimizations at Level 3

The default value for optimization is **-O3**. The global optimizer is invoked at this level with the following optimizations.

- Alias analysis for variables, arrays, and structures with constant subscripts.
- Global Constant Propagation.
- Global Copy Propagation.
- Forward Code Motion for loops without control flow.
- Control Flow Optimization.
- Common Sub Expression Elimination.
- Backward Hoisting of Code out of Loops.
- Strength Reduction.
- Reassociation.
- Loop Unrolling.
- "Extra" optimizations.

Additional Optimizations at Level 4

The following optimizations are applied at level 4 (**-O4**).

- Function inlining.
- Inter-procedural global optimization.

Additional Optimizations at Level 5

The following optimizations are applied at level 5 (**-O5**):

- More extensive inlining.
- Strength reduction and re-association for loops with very complex control flow.
- Loops with embedded branches are not unrolled during grafting.
- When a loop body contains a function call that is very likely to be executed, unrolling is disabled.
- After unrolling, the loop body is optimized for the iterator range.
- Inlining is a function of the size of the callee, the frequency level, the nesting level of the inlining, and the level of optimization.

Optimization Pros

- Optimized code runs faster, generally speaking.
- Globally specifying the optimization level gets fast code faster.
- Optimized code may have smaller code size.

Optimization Cons

- Using optimization can sometimes produce a dramatic increase in compile time.
- You are limited to **-O1** with debugging.
- Programs generated with intra-procedural optimization must all be recompiled together.

Global Optimization Issues

Caveat Regarding Global Optimizations

- An individual optimization can be enabled both globally and by explicit setting. In this case, the relative order is significant. For example,

```
tmcc -c -Xunroll=0 -O5 program.c
```

  generates an error message:

```
Overriding explicit setting of control unroll.
```

- Loop unrolling is enabled automatically at **-O5**.
- Specifying **-O5** later in the command line disables the earlier setting. To work, the relative positioning should be inverted, as shown below:

```
tmcc -c -O5 -Xunroll=0 program.c
```

Machine-Dependent Options

The following machine-dependent options can be specified to the compiler.

Option	Description
-tmccom -no_caller_save ––	Do not use caller save registers.
-tmccom -no_dtrees ––	Limit decision trees to basic blocks.
-tmccom -serial ––	Generate scheduling constraints that enforce the original source ordering of all load and store operations.

- By default, the compiler uses caller save registers for leaf functions.
- Code size and execution time is increased with the **-no_dtrees** and **-serial** options.
- These options are for compiler and systems debugging.

Loop Optimization

Loop overhead is very important in many programs.

Taken as a group, the following optimizations allow the overhead of loops to be reduced. They will be discussed individually.

Optimization	Syntax
Loop unrolling	(automatic at -O3)
Loop unrolling	**#pragma TCS_unroll** **#pragma TCS_unrollexact**
Profile-driven compilation	**tmcc -p** **tmcc -r**
Grafting	**tmcc -G**

Automatic Loop Unrolling

Overhead is required in loops (to increment loop variables) and in branches. Loop unrolling reduces overhead by replicating the body of the loop.

- Loop unrolling is applied automatically at **-O3**.
- Loop unrolling is a machine-independent optimization.

The program below calculates the sum of squares of an array (vector distance).

```
main(){
   int v[64];
   (void)norm(v);
}
norm(int *v){
   int i, f = 0;
   for( i=0; i<64; i++ ) f += v[i] * v[i];

   return f;
}
```

There are 64 iterations of this loop. When applied to this example, the loop is unrolled eight times, thus reducing overhead.

Using loop unrolling, the transformation above can be obtained without source changes. The program can be compiled with loop unrolling as follows.

```
tmcc -host tmsim -o unroll.out unroll.c
```

The default optimization level is **-O3**.The program can be compiled without loop unrolling, as follows.

```
tmcc -host tmsim -Xunroll=0 -o dont_unroll.out unroll.c
tmsim dont_unroll.out
```

To measure the benefit of automatic unrolling, use the following.

```
tmcc -O3 -Xunroll=1
```

To measure the code size increase, use **tmsize(1)**.

Pros

- Automatic unrolling is straightforward.
- Automatic unrolling uses information from the program's real behavior.
- Unrolling by the compiler is cleaner than changing your source code.

Cons

- With automatic unrolling, the compiler sometimes does not know whether the step count is exact (see below).

Manual Loop Unrolling

The program above can be compiled with manual unrolling, as follows:

```
main(){
   int v[64];
   (void)norm(v);
}
norm( int *v ){
   int i, f = 0;
#pragma TCS_unroll=64
   for( i=0; i<64; i++ ) f += v[i] * v[i];

   return f;
}
```

The pragma is applied to the following loop. The loop is replaced by a single statement block. Exact unrolling must be turned off.

Adding the following statement to your program, disables loop unrolling:

```
#pragma TCS_unroll=0
```

Adding the following statement to your program calculates the unroll count automatically.

```
#pragma TCS_unroll=1
```


Exact Unrolling

When the loop count is variable, the compiler must add fix-up code for the last iteration, in case the loop count is not an exact multiple of the step. The pragma **TCS_unrollexact** tells the compiler not to do this. The pragma has loop scope.

```
norm(int *v, int n){
   int i, f = 0;
#pragma TCS_unroll=8
#pragma TCS_unrollexact=1
   for (i=0; i<n; i++) f += v[i] * v[i];

#pragma TCS_unrollexact=0
   return f;
}
```

Pros

- Manual unrolling is good if you need control in a critical function or loop.

Cons

- With **TCS_unrollexact**, if the step is not exact, the code is wrong.

Profile-Driven Compilation

The first step in profile-driven compilation is to obtain profile information about the program. You can then perform grafting, loop unrolling, and function inlining.

The optimizations in the TriMedia compiler chain are built upon profiling. Loop unrolling and grafting use execution frequencies and branch probabilities. Function inlining uses execution frequencies at the call site. Decision tree construction uses profile information for **if** conversion.

The effectiveness of all these optimizations is increased using profile-driven compilation. The program must be run twice. The first run is with the **-p** option and produces a file "dtprof.out" file on execution.

```
tmcc -p -o profiled.out unroll.c
```

The purpose of using the **-p** option is to generate information to be used by the compiler. The code generated includes instructions to perform measurements.

- Some optimizations may be turned off by default when **-p** is given.
- The target can be on a PC host or the simulator (**-host** option).
- Compile your program at the default optimization level (**-O3**).
- Do not use the **-g** option (debugging).
- Do not use the **-G** or **-R** options.
- Do not confuse **-p** option with the **-ptm** (performance analysis) option.

The second run uses the **-r** option. This reads the information from the file dtprof.out.

```
tmcc -r -o optimized.out unroll.c
```

- Loop unrolling is improved with profile information.
- Use the **-G** option to enable grafting.
- Use the **-O4** option, to enable function inlining.
- If you need to keep profile information around, use the **-r=file.dtprof** option.
- To determine the speed up, use **tmsim -statfile** or **tmcc -ptm**.
- The same options and optimization levels should be used for both compilation runs.

The dtprof.out file contains information about the behavior of the program, including how often each function is executed, how often each loop is executed, and the probability of the guard of an **if** statement being true or false.

This information is be used by the compiler to make important decisions.

- Whether to optimize for code size or for speed.
- Whether to inline a function, depending on the frequency at the call site.
- Whether to unroll, depending on the frequency of the loop count.
- How much to unroll.
- Choices between optimizations (which algorithm to use, for example).

Pros

- There is no need to manually specify optimizations.

Cons

- No control flow optimization.
- The first run (**tmcc -p**) runs slower.
- Some algorithms are data-dependent.
- If you change your source, you need to reobtain the profile information.
- No global optimizations.

Grafting

The use of grafting is a straightforward way to improve the performance of your program. It reduces loop overhead (loop unrolling, for example); it lends itself to more general control structure; and it is also a mechanism of code replication.

Figure 2 shows a decision tree ending in a branch. The actual code is not important in this example. The decision tree **__ip_DT_1** has two exits, one leading back to itself and the other leading to another tree. The back edge to itself has a probability of 0.98.

```
{__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 2 Decision Tree Ending in a Branch

In this case, the decision tree has an execution count of 50. These statistics are derived from a profiling run (**tmcc -p** option).

The **-G** option tells the compiler to use grafting. Use grafting in combination with the **tmcc -r** option, which tells the compiler to read the dtprof.out file to guide grafting. The target can be the PC host or the simulator (**-host** option).

You need to compile your program a first time, as explained previously. To compile using grafting, use the following command.

```
tmcc -G -host tmsim -o grafted.out unroll.c
```

Manual Grafting

Manual grafting can be applied on a function-by-function basis.

- Use **#pragma TCS_graft** to apply grafting.
- Use **#pragma TCS_nograft** to disable grafting.
- The pragma overrides the command-line option.

Pros

- Grafting applies to more general control structures.

Cons

- Grafting reduces overhead less than loop unrolling.
- Grafting increases code size because the loop test must be replicated.
- Loop unrolling is inhibited by grafting, resulting in a performance reduction in some cases.
- Because of compiler algorithms, **-g** cannot be used in conjunction with grafting.

Graft Tuning File

The graft tuning file allows specification of parameters for individual functions. Default parameters are as shown below:.

Parameter	Default Value	Description
Graft Enable	1	Boolean flag enabling or disabling grafting.
Codesize	20	Maximum factor to increase code size by grafting.
Depth	20	Number of times grafting along a particular path.
Probability	0.4	Probability of a branch must exceed this threshold.
Execution Count	10	Execution count must exceed this threshold.

Specify this file with the command line below.

```
-tmccom -graft_tuning_file <file> --
```

The first field in the line specifies **<default>** or the function. An example graft tuning file is shown below.

```
fibonacci       0       20      20      0.4     10
<default>       1       4.0     2       0.4     10
```

- Grafting is disabled on the function **fibonacci**.
- For other functions, the default code size is decreased.

Pros

- Instruction cache misses can be reduced in some cases.

Cons

- The compiler applies a sophisticated heuristic automatically.
- Reducing code size can increase ILP because of branches.

Other Optimizations

The following optimizations also improve the performance of loops.

Cross Iteration Hoisting

Cross Iteration Hoisting moves code from one iteration of to the dynamically previous one. This applies to long latency operations (such as loads). Use **-Xcih**. This is set at **-O3**.

Forward Code Motion

Forward Code Motion moves operations such as stores past the loop. Use **-Xfcm**. This is set at **-O3**.

Induction Variable Replacement

Induction Variable Replacement reduces the number of loop control variables. Use **-Xivrep**.

Function Inlining

Function inlining is an easy way to improve the performance of your program.

Inlined functions are safer and more portable than manual inlining and preprocessor macros. Variables are exposed to the global optimizer (thus enabling other optimizations), procedure call and return overhead are eliminated, and interruptions in control flow caused by branches are eliminated.

The following sections explain three ways of applying function inlining: automatic inlining, pragma-controlled inlining, and command-line controlled inlining.

Automatic Inlining

Automatic inlining is the easiest way to obtain the benefits of inlining. It is applied automatically at **-O4** and **-O5**. A simple example is shown below.

```
main(){
   int i, sum = 0;

   for( i=0; i<5; i++ ) sum += prod(i, i, i);

   printf("%d\n", sum);
}
prod(int a, int b, int c){
   return a * b * c;
}
```

This program can be compiled with automatic inlining as follows.

```
tmcc -O4 inline.c -o inline.out
```

The function **prod** is inlined into the body of the main procedure. If the function is static, the definition is removed.

Automatic Inlining with the -p Option

Inlining is based on an analysis of the source program. The performance of inlining can be improved by using the program's run-time behavior.

- The program above can be compiled with the **-p** and **-O3** options, then executed on the host processor or the simulator.
- Recompiling with the **-O4** and **-r** options improves the inlining.

Automatic Explicit Inlining

Inlining is enabled automatically at **-04**. The compiler distinguishes between three classes of procedures and functions.

- Ordinary routines.
- Routines declared inside a C++ class or in C with the **inline** keyword.
- Library intrinsic functions.

The table below shows how to explicitly enable inlining from the command line at lower optimization levels. These apply to the entire file. Level 0 corresponds to no inlining. The higher the level, the more aggressive the inlining.

Options	Description
-Xdeflib=0 to 3	Inlining of library intrinsic functions. **-Xdeflib** is equivalent to **-Xdeflib=2**. The pragma equivalent is **TCS_deflib.**
-Xinllev=0 to 5	Automatic inlining of ordinary routines. **-Xinllev** is equivalent to **-Xinllev=4.** The pragma equivalent is **TCS_inllev**
-Xsinllev=0 to 5	Automatic inlining of "inline" or C++ classes. **-Xsinllev** is equivalent to **-Xsinllev=4.** The pragma equivalent is **TCS_sinllev.**

Using the Inline Keyword

Inlining can be applied automatically using the **inline** keyword. The following is an example for the C library **getc** function.

```
static inline int getc(FILE *fp) {
   if (--fp->count < 0)
      return _getc(fp);
   else
      return *fp->ptr++;
}
```

The **inline** keyword indicates the function is to be expanded at the point of call. If the inlined function calls a function that is inlined, it will be inlined also. Recursive functions cannot be inlined.

The word **inline** is a keyword in C by default. Use **-xc-=inline** to disable it.

Pros

- Inlined functions are safer and do not have side effects that macros have.
- Declaring the function as **inline** makes it debugger-visible.

Cons

- Definition-controlled inlining is applied systematically whenever the **inline** compiler keyword is encountered.

Pragma-Controlled Inlining

An example of TCS pragma-controlled inlining is shown below.

```
#pragma TCS_inline=prod;sum
main(){
   printf("%d\n", sum(2, 3, 4, 5, 6, 7));
}
sum(int a0, int a1, int b0, int b1, int c0, int c1) {
   return prod(a0, a1) + prod(b0, b1) + prod(c0, c1) ;
}
prod(int x, int y) {
   return x * y;
}
```

- This compiles into a single function.
- The elements of the pragma list can include a priority (higher means more inlining).
- There are pragma equivalents of automatic inlining.
- The syntax is **#pragma TCS_deflib**, **#pragma TCS_inllev**, **#pragma TCS_sinllev**.
- The pragma **TCS_noinline** specifies no inlining (see below).

Command-Line Controlled Inlining

Functions to be inlined can also be specified on the command line, as shown below. To escape the semicolons, the lot should be quoted.

Command Line	Description
-Xinline="procedure;procedure;..."	Inline procedures specified.
-Xnoinline="procedure;procedure;..."	Do not inline procedures specified.

Inlining can be applied without specifying the **inline** keyword, as shown below.

```
tmcc -tmccom -Xinline=prod -- prog.c
```

The argument to the **-Xinline** option is a list, separated by semicolons. Each element can be a procedure name or a pair. The second element is the priority.

The **-Xnoinline** option specifies procedures not to be inlined.

The following command compiles the previous program with automatic inlining.

```
tmcc -tmccom -Xinllev -- prog.c
```

Inlining is applied separately to intrinsic routines, user routines, and routines inside classes.

The ANSI C and C++ standards require intrinsic routines to do some error checking. For example, the math routines are expected to set the variable **errno** on either a domain or a range error, and to provide certain specific values on range overflow or underflow.

The **-Xsafeintr** option can be used to inform the compiler that error checking in library intrinsic functions is not necessary.

Pros of Inlining

- Inlining enables other optimizations that are otherwise limited to a procedure basis.
- Specifying from the command line allows using a makefile to generate several versions from one source.

Cons of Inlining

- Aggressive inlining increases code size.
- Inlining can affect instruction cache performance.

Automatic vs. Definition-Controlled Inlining

- Automatic inlining is the easiest to use.
- Specifying which routines are to be inlined provides control.
- If intrinsic routines are inlined, they can be faster if it is known that these checks are not required.
- In C/C++, the routine **sqrt** can only be inlined if the **safeintr** control variable is specified.

Alias Analysis

This section explains how alias analysis obtains better performance in your program. Alias analysis is concerned with determining whether two references point to the same object. The quality of alias analysis is particularly important for optimization.

There are three ways to control alias analysis in the compiler. You can use the compiler's alias analysis algorithm, you can use restricted pointers, or you can use the **-A[012]** command option for unsafe aliasing.

Alias Analysis Algorithm

The alias analysis algorithm works for variables, arrays of one dimension, and multi-dimensional arrays. The following criteria apply.

- An array reference and a scalar reference do not alias.
- References to distinct arrays, scalars, and restricted pointers do not alias.
- Globals, automatics, and statics do not alias.
- References to distinct constant array indices do not alias.
- **tab[x]** and **tab[x+C]** do not alias, where **C** is a non-zero constant, **x** is a variable, and **tab** is an array or restricted pointer.
- References to distinct structure elements do not alias.
- A scalar does not alias with a pointer reference if the address is not taken.
- Index expressions involving variables are evaluated over all assignments for constant values.
- References that cannot be shown to be different by the above criteria may possibly alias.

Unsafe Alias Analysis

Alias analysis in C is complex because of the use of pointers. Using unsafe alias analysis gets better performance by relaxing some of the language requirements of the compiler.

- The compiler currently has three levels of alias analysis.
- Level zero is perfectly safe (that is, no assumptions are made other than allowed by ANSI C).
- The two higher levels do make assumptions, but are safe in most programs.

Default Behavior

You can specify unsafe alias analysis with the option **-A[012]** to the compiler. The default level is level one. Level one makes the following assumptions:

- It is assumed that a reference to an object points to the whole object
- A pointer does not point to itself. Moreover, when a pointer points to a structure, there is no field in the structure that points to the same place.

The example below illustrates the first point. The access to **p–>next** cannot conflict with an access to **head** because the reference is inside the structure.

The accesses can be reordered because the initialization of p is required to point to the whole structure.

```
struct ptr {
   char c;
   short s;
   struct ptr *next;
} *p;

struct ptr *head;
p->next = 0;
head = 0;
```

The program below illustrates the second point. The accesses can be reordered because **p** is not allowed to point to itself.

```
int **p, **q;
*p = 0;
q = p;
```

Unsafe Behavior

The **-A2** option relaxes the rules for alias analysis. The compiler assumes that the accesses to extern and static variables do not alias with stores through pointers.

- The program below shows a program that initializes four arrays.
- The compiler does not know whether the addresses of the external variables have been assigned to a pointer.
- For example, the loop is executed 32 times.
- However, **qty_first_reg** could point to **max_qty** (possibly).
- If this is true, the loop should only be executed once.
- This adds many redundant load accesses

```
#include <ops/custom_defs.h>
#define NREG    32
typedef struct rtl *rtx;
int *qty_first_reg,  *qty_last_reg, max_qty;
rtx *qty_const, *qty_const_insn;
main(){
   int start, stop;
   max_qty = NREG;
   qty_first_reg  = (int*)malloc(NREG * sizeof(int));
   qty_last_reg   = (int*)malloc(NREG * sizeof(int));
   qty_const      = (rtx*)malloc(NREG * sizeof(rtx));
   qty_const_insn = (rtx*)malloc(NREG * sizeof(rtx));
   start = CYCLES();
   clearregs();
   stop = CYCLES();
   printf("cycles = %d\n", stop-start);
}
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;
    }
}
```

Pros of the -Xalias option

- It is always safe.
- It improves code size and execution time.

Pros of the -A option

- It can be applied on a per-file basis or using a **#pragma (TCS_A0, TCS_A1, TCS_A2)**.
- It improves performance more than the **-Xalias** option alone.

Restricted Pointers

This section explains restricted pointers.

- Restricted pointers are a very easy way to improve the performance of your program.
- **restrict** is a type qualifier like **const** and **volatile**.
- Like **volatile**, **restrict** is intended to affect optimization.
- Like **const**, there are rules to which a program must conform.
- **restrict** is intended to affect optimizations in the opposite direction from **volatile**, by enabling them.
- With **restrict**, an alias-free function call interface can be obtained, as in FORTRAN.
- The concept of restricted pointers is as proposed by the ANSI C Numerical Extensions Group.
- The performance analysis tool **tmprof** can be used to find parts of your program that can benefit.

Semantics of Keyword Restrict

The following points define the semantics of the **restrict** keyword.

- A **restrict** is an assertion that no other variable, pointer, or restricted pointer will alias the object for as long as the restricted pointer is in scope.
- Aliasing a restricted pointer with a normal pointer is still possible. For example,

```
char *restrict fun;
char **pfun;
pfun = &fun;
```

- Declaring a pointer restricted gives it aliasing properties similar to an array or scalar variable.
- A restricted pointer must point to a private non-overlapping memory region.
- The C compiler does not generate ordering constraints between loads or stores to distinct variables.

Scope of Restricted Pointers

The following sections explain the effects of declaration of a restricted pointer, depending on the scope.

Restricted Pointers of File Scope

This section explains restricted pointers of file scope.

- Global variables declared as **restrict** and variables declared as **restrict** in a header file have file scope.
- A restricted pointer of file scope should point to a single array object for the whole program.
- The array may not be referred to through the restricted pointer and its name or another restricted pointer.
- References through the pointer are optimized as if they were references to an array in its declared scope.
- Restricted pointers of file scope are useful for providing access to dynamically allocated arrays.

Restricted Pointers as Function Parameters

This section explains restricted pointers as parameters to a function.

The following example is the source code for the C library **memcpy** function.

```
void *memcpy(void *s1, void *s2, size_t n){
   char *t1 = s1;
   char *t2 = s2;

   while (n-- > 0) *t1++ = *t2++;

   return s1;
}
```

- The compiler cannot know whether pointers overlap.
- This prevents block copying of the arrays on architectures that support this.
- This forces sequential execution of memory accesses.

The following shows the source code with using the qualifier restrict

```
void *memcpy( void * restrict s1, void * restrict s2, size_t n ){
   char * restrict t1 = s1;
   char *t2 = s2;

   while (n-- > 0) *t1++ = *t2++;

   return s1;
}
```

Using the **restrict** qualifier tells the compiler that **s1** and **s2** point to different storage areas

- The semantics are the same as if **s1** and **s2** were arrays as opposed to pointers.
- This corresponds to FORTRAN semantics for call by reference.
- With **restrict** the compiler knows that **s1** and **s2** cannot overlap.
- Types other than pointer types cannot be **restrict**-qualified.

Restricted Pointers of Block and Structure Scope

This section explains restricted pointers having block or structure scope.

- A pointer of block scope makes an aliasing assertion that is limited to the block.
- For structures, the scope of the assertion is the scope of the ordinary identifier used to access the structure.
- For example, in the code below **r.p**, **r.q**, **s.p**, and **s.q** should all point to distinct storage.

```
struct t {
   float *restrict p;
   float *restrict q;
} ;
void f4( struct t r, struct t s ) {
   ...
}
```

Pros

- It is easy to add restricted pointers to a program.
- In many cases, function parameters can be seen not to overlap.

Cons

- Pay attention to compiler warnings. Otherwise the code may be incorrect.

Converting If Statements

Converting If statements eliminates control flow points, eliminates expensive branch instructions and increases instruction level parallelism.

Option	Description
-tccom -if_param <n>	Specifies parameter for conversion
-tccom -max_if_size <n>	**if** conversion, when possible, is applied to **if** statements with fewer than <n> operations. The default value is 4.
-if_convert	Applies polynomial **if** conversion algorithm.
-full_ifconvert	Applies exponential **if** conversion algorithm.

- Only control flow points that are not loop headers, function return points, or targets of table jumps are eligible for **if** conversion.
- **if** conversion works best when profile information is available.
- Full **if** conversion can take a long time to compile.
- Two heuristics are applied to decide which control flow points to convert. **-if_param** *n* sets the ratio between both heuristics ($0 \leq n \leq 100$).
- **if** conversion serves the same purpose as grafting. Applying both at the same time is therefore not recommended when code size is important.
- In general, **if** conversion, unlike grafting, results in only a minor code size increase.

Mapping from Optimization Level to Optimizations

This section explains the optimization mapping. Rows in Table 3-1 correspond to the different individual optimization control variables. Columns in the table correspond to the different optimization levels.

Table 2 Optimization Mapping

		Optimization Level					
Optimization	**Variable**	**–O0**	**–O1**	**–O2**	**–O3**	**–O4**	**–O5**
Alias Analysis	-Xalias	0	1	3	3	4	4
Call Modification Analysis	-Xcallmod	0	1	1	1	2	2
Constant Propagation	-Xconstp	0	0	2	2	2	2
Copy Propagation	-Xcopyp	0	0	2	2	2	2
Loop Forward Code Motion	-Xfcm	0	0	1	2	2	2
Control Flow Analysis	Xflow	0	0	1	1	1	1
Inlining Level	-Xinllev	0	0	0	0	1	1
"Main" Optimizations	-Xmopt	0	1	3	4	4	4
Register Allocation	-Xreg	0	0	1	1	3	3
Loop Unrolling	-Xunroll	0	0	1	1	1	1
"Extra" Optimizations	-Xxopt	0	0	2	2	3	5
Expression Reordering	-Xzone	0	0	1	1	1	1

Entries in Table 2 are the value of the control variable for that optimization level. For example, the default optimization level (**-O3**) corresponds to: **-Xalias=3 -Xcallmod=1 -Xconstp=2 -Xcopyp=2 -Xfcm=2 -Xflow=1 -Xinllev=0 -Xmopt=4 -Xmopt=4 -Xreg=1 -Xsched=2 -Xunroll=1 -Xxopt=2** and **-Xzone=1**.

Rows with 0 or 1 values correspond to control variables with on/off settings.

Summary of individual optimizations

This section provides a summary of individual optimizations. Rows in Table 3 correspond to a control variable and the corresponding optimization. Columns correspond to different settings for the variable.

Table 3 Individual Optimizations

		Settings					
Optimization	**Variable**	**0**	**1**	**2**	**3**	**4**	**5**
Alias Analysis	-Xalias	—	scalar	arrays element	arrays + flow	arrays + flow	*
Call Modification Analysis	-Xcallmod	—	global	intra	*	*	*
Constant Propagation	-Xconstp	—	local	intra	*	*	*
Copy Propagation	-Xcopyp	—	local	intra	*	*	*
Loop Forward Code Motion	-Xfcm	—	without 'if's	all loops	*	*	*
Control Flow Optimization	-Xflow	—	on	*	*	*	*
Inlining Level	-Xinllev	—	for $1 \le n \le 5$, increasing aggressiveness and optimization				
"Main" Optimizations	-Xmopt	—	local	intra	intra + flow	*	*
Register Allocation	-Xreg	—	intra	inter	*	*	*
Loop Unrolling	-Xunroll	—	automatic (for $n > 1$, use level given)				
"Extra" Optimizations	-Xxopt	—	for $1 \le n \le 5$, increasing aggressiveness and optimization				
Expression Reordering	-Xzone	—	"C" Statement	Non-ANSI	*	*	*

Entries "—" and "*" correspond to no optimization and an illegal setting, respectively.

"Local" means that an optimization is applied on the basic-block level. "Intra" means complete intra-procedure analysis is performed. "Inter" means that inter-procedure analysis is applied to the entire compilation unit or program. "Flow" means that flow analysis is performed to determine the scope of the optimization.

For example, valid settings for the **-Xalias** control variable are from 0 to 4. Specifying **-Xalias=0** corresponds to no alias analysis. Specifying **-Xalias=4** sets alias analysis for scalar variables, array elements with constant subscripts, using flow analysis.

Control flow optimization eliminates dead code and branches to branches and performs basic block collapsing. “Main” optimizations include common subexpression elimination, backwards code hoisting, and strength reduction and reassociation. “Extra” optimizations are applicable to only a minority of programs.

Chapter 4

Using the Instruction Scheduler

Introduction

This chapter is divided into three main sections. The first section explains Instruction Scheduler options, the second explains Instruction Scheduler reports, and the third explains the decision tree syntax.

Below are some general introductory observations.

- Some scheduler optimizations may need to be disabled because of hardware or other considerations.
- Disabling the optimization in a single file limits performance impact.
- It is important to understand the scheduler and assembler to get the best performance.

Instruction Scheduler Options

Main Options

The main options of the Instruction Scheduler are shown in Table 4.

Table 4 Main Options

Option	Description
-eb	Default endianness is big-endian.
-el	Default endianness is little-endian.
-h	Prints a brief help message with options.
-dynamic	Generates dynamic disambiguation code. This option generates code potentially to reorder critical and dependent store → load pairs at runtime. The option can be useful when the static analysis in the compiler cannot determine whether the store → load pair can be reordered.
-o=file	Uses the explicitly scheduled output file. The output file (**-o**) should include the .s extension.
-O1	Selects scheduler optimization level 1.
-O2 **-O3**	Selects scheduler optimization level 2. Selects scheduler optimization level 3. Higher levels of optimization invoke more algorithms.
-reportlog=file	Uses the explicitly specified file for the reports.

Table 4 Main Options (Continued)

Option	Description
-V	Prints tmsched version information.
-v	Enables verbose output.
-w	Suppresses warning messages.

These options must be bracketed by **-tmsched <option> ––.**

- The default optimization level is **-01**.
- Endianness specification is obligatory.

The time taken to generate a schedule depends on the number of operations. Some decision trees contain hundreds or even thousands of operations, so the scheduling phase of compilation can be very long. For long trees, use **#pragma TCS_break_dtree** to force a break. The **tmcc** option **-v** can be used to find which file is being compiled.

Pros

- More algorithms mean better schedules.
- For long trees, breaks reduce compile time.
- Breaks reduce interrupt latency.

Cons

- Breaks add overhead for jumps.

Control Options

The options shown in Table 5 control scheduler operation.

Table 5 Control Options

Option	Description
-nocode	Suppresses assembly code generation.
-serial=**treename,treename**	Serializes memory operations on the trees named by the comma-separated list of tree names (first label if there is more than one label for the tree). The effects of this option are cumulative.
-d=<**tree**>	Generates graphs.dot file for tree. GraphViz, the tool that can display the graph file, can be found at http://www.research.att.com/sw/tools/graphviz/
-noverify	Disables verification of schedule.

Cons

- Use of **-serial** is limited to the debugging of problems related to alias analysis.
- Use of **-nocode** option is limited to the generating of statistics.

The -bc (Avoid Bank Conflicts) Option

This section explains the **-bc** option. This option tells the scheduler to delay a memory operation if it is likely to cause a bank conflict.

The TriMedia data cache is set into eight banks. Two memory operations can proceed in parallel as long as bits 2-4 of their addresses are different. For example, in the following program, both the first and second pair of assignments can proceed in parallel.

```
int a[32];
a[0] = 0;       a[1] = 0;
a[8] = 0;       a[9] = 0;
```

However, if the assignments are interchanged, this parallelism would not be possible:.

```
a[0] = 0;       a[8] = 0;
a[1] = 0;       a[9] = 0;
```

Use **tmsim -statfile** and **tmprof** to find out where bank conflicts exist.

Pros

- Avoiding bank conflicts gains a cycle per instruction.

Cons

- Indexed accesses cannot be optimized.
- Most programs do not benefit from the optimizing of bank conflicts.

Speculative Execution Options

Instructions can depend on a guard if they are under the control of an **if** statement. For best performance, the Instruction Scheduler executes loads and floating-point operations before the guard is ready. This is called *speculative execution*. In some situations, speculative execution may not be possible (if floating-point exceptions must be precise, for example). The speculative execution options of the Instruction Scheduler are shown below. Note that disabling speculation does reduce performance.

Speculative execution can also cause problems in a system where a read to a memory location has side effects. This is true, for example, in Pentium II systems using the Intel 440LX chipset. Standard TriMedia PCI drivers now conservatively disable PCI access

unless specifically required. However, code that is expected to run in this sort of environment might be well-advised to turn off speculative execution.

Table 6 Speculative Execution Options

Option	Description
-nofloatspec	Disallows speculation of floating-point operations. Use this option if you need precise floating-point exceptions.
-noloadspec	Disallows speculation of all memory load operations.

Debugging and Exception Support Options

Debugging support on TriMedia is in both hardware and software. This section explains the scheduler options and their effects on debugging.

- Instruction and data breakpoints are modeled as interrupts.
- Floating-point instructions have latency for exceptions.
- The **-fuzzy_bp** option is recommended for code optimized for speed.

The following options add extra instructions.

Table 7 Debugging and Exception Support Options

Option	Description
-precise_bp	Requires precise data interrupts. Use this option if you require that data breakpoints be caught in the same tree.
-precise_fp	Requires precise floating-point exceptions. Use this option if you require the same treatment for floating-point exceptions. By default, floating-point exceptions are imprecise.
-fuzzy_bp	Does not require precise data interrupts. Use this option if you do not use data breakpoints. You will need to use the **-nofloatspec** option as well.
-g	Enables scheduler behavior required by the debugger. This option implies **-precise_bp**.

Cons

- The **-precise -bp** and **-precise -fp** options adds instructions for latency.

Instruction Scheduler Reports

The scheduler can generate a fairly wide range of reports, which can be very useful when you need to understand how your code is being mapped to the TriMedia hardware.The table below gives a list of report options with the corresponding information. For further information, refer to Chapter 5, *Performance Analysis Overview*.

Option	Description
-report=summary	Overall statistics. Issues/cycle, scheduling factor, total cycles, slot utilization.
-report=procstat	Same as "summary" on a per-procedure basis.
-report=schedule1	Functional units used per instruction (by category).
-report=schedule2	Operations used per instruction, with probability.
-report=schedule4	Operations used per instruction.
-report=schedule8	Functional units used per instruction (shows all 27 units).
-report=schedule16	Per tree statistics.
-report=treestat	Per tree statistics. Compares with code generated for an architecture with infinite resources.
-report=reportprof	One line per tree of statistics.
-report=module	Execution time at different clock speeds.

Report types are illustrated for several of these options, beginning on 69. Shown below is the program for which the sample reports have been generated. The program calculates a convolution.

```
#define NINPUTS 400
void direct_convolution(
   char * restrict a, char * restrict b, int * restrict c )
){
   int k;
   for (k=0; k<NINPUTS; 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;
      b += 4;
   }
}
```


Report 1—Issue Slots

Scheduler Report 1 reports on issue slots. Figure 3 shows the sample report.

```
#tree __direct_convolution_DT_0
  0:         const      branch         alu         ---         ---
  1:           ---         ---         ---         ---         ---
  2:           ---         ---         ---         ---         ---
  3:           ---         ---         ---         ---         ---
#-----------------------------------------------------------------
#tree __direct_convolution_DT_1
  0:           alu         ---         ---        dmem        dmem
  1:           alu         alu         ---        dmem        dmem
  2:           alu         ---         ---        dmem        dmem
  3:           ---       ifmul         ---        dmem        dmem
  4:           ---       ifmul       ifmul        dmem        dmem
  5:           ---       ifmul       ifmul        dmem        dmem
  6:           ---       ifmul       ifmul        dmem        dmem
  7:           ---       ifmul       ifmul        dmem        dmem
  8:           ---       ifmul       ifmul        dmem        dmem
  9:           ---       ifmul       ifmul        dmem        dmem
 10:           alu       ifmul       ifmul        dmem        dmem
 11:           alu       ifmul       ifmul        dmem        dmem
 12:           alu       ifmul       ifmul        dmem        dmem
 13:           alu       ifmul       ifmul        dmem        dmem
 14:           alu       ifmul       ifmul        dmem        dmem
 15:           alu       ifmul       ifmul        dmem         ---
 16:           alu       ifmul       ifmul         ---         ---
 17:           alu       ifmul       ifmul         alu         ---
 18:           alu       ifmul       ifmul         alu         alu
 19:           alu       ifmul       ifmul         alu         alu
 20:           alu       ifmul       ifmul         alu         alu
 21:           alu       ifmul       ifmul         alu         ---
 22:           alu       ifmul       ifmul         alu         alu
 23:           alu       ifmul       ifmul         alu         ---
 24:           alu       ifmul       ifmul         alu         ---
 25:           alu       ifmul       ifmul         alu         ---
 26:           alu       ifmul       ifmul         alu         ---
 27:           alu       ifmul       ifmul         alu         alu
 28:           alu       ifmul       ifmul         alu         alu
 29:           alu       ifmul       ifmul         alu         alu
 30:           alu       ifmul       ifmul         alu        dmem
 31:           alu       ifmul       ifmul         alu         alu
 32:           alu       ifmul       ifmul         alu         alu
 33:           alu       ifmul       ifmul         alu         alu
 34:           alu       ifmul       ifmul         alu        dmem
 35:           alu       ifmul         alu         alu         ---
 36:           alu         alu         alu         ---         ---
 37:           alu         alu         alu         ---         ---
 38:           alu         ---         ---        dmem         ---
 39:           alu      branch      branch         ---         ---
 40:           alu         ---         ---         ---         ---
 41:           alu         ---         ---         ---         ---
 42:           ---         ---         ---        dmem         ---
#-----------------------------------------------------------------
```

Figure 3 Report File "schedule1.txt"

You can obtain the report shown in Figure 3 by entering the following:.

```
tmcc -K -tmsched -report=schedule1 -reportlog=schedule1.txt -- filter.c
```

Below are some observations regarding this report.

- Columns correspond to issue slots, rows to cycles.
- Four cycles are necessary for the first tree, 42 for the second branches to the loop.
- The first tree just branches, the second corresponds to the loop.
- Four distinct phases can be isolated in the second tree.
- The first phase reads data. **dmem** corresponds to reading from “a” and “b” in C.
- The second phase multiplies elements. **ifmul** corresponds to multiplications in C.
- The third phase adds elements together. **alu** corresponds to additions in C.
- The fourth phase stores the data. The second **dmem** corresponds to the stores in C.
- The length of the loop is determined by the number of multiplies.
- The compiler and scheduler did a reasonable job.

Report 4—Operations

Scheduler Report 4 reports on operation usage. This provides a better understanding of the critical path and operation latency. Figure 4 shows the sample report.

```
#tree __direct_convolution_DT_1
  0:         iaddi          ---          ---        ild8d        ild8d
  1:          iles         igeq          ---        ild8d        ild8d
  2:         wrreg          ---          ---        ild8d        ild8d
  3:           ---         imul          ---        ild8d        ild8d
  4:           ---         imul         imul        ild8d        ild8d
  5:           ---         imul         imul        ild8d        ild8d
  6:           ---         imul         imul        ild8d        ild8d
  7:           ---         imul         imul        ild8d        ild8d
  8:           ---         imul         imul        ild8d        ild8d
  9:           ---         imul         imul        ild8d        ild8d
 10:          iadd         imul         imul        ild8d        ild8d
 11:          iadd         imul         imul        ild8d        ild8d
 12:          iadd         imul         imul        ild8d        ld32d
 13:          iadd         imul         imul        ild8d        ld32d
 14:          iadd         imul         imul        ild8d        ld32d
 15:          iadd         imul         imul        ld32d          ---
 16:          iadd         imul         imul          ---          ---
 17:          iadd         imul         imul         iadd          ---
 18:          iadd         imul         imul         iadd         iadd
 19:          iadd         imul         imul         iadd         iadd
 20:          iadd         imul         imul         iadd        iaddi
 21:          iadd         imul         imul        iaddi          ---
 22:          iadd         imul         imul         iadd         iadd
 23:          iadd         imul         imul         iadd          ---
 24:          iadd         imul         imul         iadd          ---
 25:          iadd         imul         imul         iadd          ---
 26:          iadd         imul         imul         iadd          ---
 27:          iadd         imul         imul         iadd         iadd
 28:          iadd         imul         imul         iadd         iadd
 29:          iadd         imul         imul         iadd         iadd
 30:          iadd         imul         imul         iadd        st32d
 31:          iadd         imul         imul         iadd         iadd
 32:          iadd         imul         imul         iadd         iadd
 33:          iadd         imul         imul         iadd         iadd
 34:          iadd         imul         imul         iadd        st32d
 35:          iadd         imul         iadd         iadd          ---
 36:          iadd         iadd         iadd          ---          ---
 37:          iadd         iadd         iadd          ---          ---
 38:          iadd          ---          ---        st32d          ---
 39:          iadd        ijmpi        ijmpt          ---          ---
 40:          iadd          ---          ---          ---          ---
 41:          iadd          ---          ---          ---          ---
 42:           ---          ---          ---        st32d          ---
```

Figure 4 Report File "schedule4.txt"

You can obtain the report shown in Figure 4 by entering the following:

```
tmcc -K -tmsched -report=schedule4 -reportlog=schedule4.txt -- filter.c
```

Below are some observations regarding this report.

- Table entries are operations. See the appropriate TriMedia data book for more information.

- Two 8-bit loads can be executed on every cycle (**ild8d** operation, cycle 0).
- The multiplication (**imul**) is in cycle 3 because of latency (3 cycles).
- One cycle later, there is an ILP (Instruction Level Parallelism) of four operations per cycle (cycle 4).
- Six cycles later, there is an ILP of five operations per cycle (cycle 10).
- Starting at cycle 18, everything is in registers and the additions (**iadd**) replace the loads in slots 4 and 5.
- Parallelism slows down again at the end.
- The **wrregs** are register-to-register moves.

Report 16—Statistics

Scheduler Report 16 produces reports on tree statistics. This gives you an overall view of the trees in your program. Figure 5 shows an edited version of the report for tree 1.

```
#tree __direct_convolution_DT_1
        scheduling algorithm: if-conversion
        instruction count: 43
        operations count: 171
        spilled values: 0
        renumbered values: 0
        tree internal jumps: 0
        average number of cycles (CPI=1): 43.000000
        ideal number of cycles: 12.000000
        scheduling factor: 27.906977%
        invocation count: 0.000000
        contribution to cycle count (CPI=1): 0.000000
        code size (excl. padding): 677 bytes
        live global registers: {r0-r7, r9-r34}
        live local registers: {r8, r35-r115}
```

Figure 5 Report File "schedule16.txt"

You can obtain the report shown in Figure 5 by entering the following:.

```
tmcc -K -tmsched -report=schedule16 -reportlog=schedule16.txt -- filter.c
```

Below are some observations regarding this report.

- On average, about 4 operations are executed per cycle (171 divided by 43).
- There are 43 cycles necessary, as opposed to 12 on a machine with infinite parallelism.
- This corresponds to 27 percent of the average ILP.
- No registers need to be spilled to memory.
- This decision tree does not use tree-internal jumps.
- This decision tree used the **if** conversion algorithm for this tree.

treestat Reports

The **-report=treestat** option also produces reports on tree statistics. Figure 6 shows the sample report.

```
#tree __direct_convolution_DT_1
#report treestat
maximum register live = 82
number of register spills = 0
average cycles on finite machine = 43.000000
ideal cycles on infinite machine = 12.000000
scheduling factor = 27.91%
tree issues/cycle = 3.976744
tree useful issues/cycle = 3.951163
tree slot utilization = 79.53%
number of operations issued dynamically = 0.000000
number of operations issued dynamically per tree = 171.000000
number of operations issued statically = 171.000000
number of useful operations issued dynamically = 0.000000
number of useful operations issued statically = 169.900000
number of static instructions = 43.000000
number of dynamic instructions = 0.000000
executed 0.000000 times (estimated)
contributes 0.000000 cycles to the total program execution
number of removed rdregs = 6 (of 6)
number of removed wrregs = 2 (of 3)
```

Figure 6 "treestat.txt" Report

You can obtain the report shown in Figure 6 by entering the following.

```
tmcc -K -tmsched -report=treestat -reportlog=treestat.txt -- filter.c
```

Below are some observations regarding this report.

- There are 82 registers used for the scheduler.
- The scheduling factor is 27.9 percent (12 divided by 43).
- Most issues are useful per cycle (3.95 out of 3.97).
- Dynamic operations correspond to execution probabilities.
- Static operations correspond to actual instructions.
- No instructions were generated for **rdreg** pseudo operations (0 out of 6).
- One instruction was generated for **wrregs** (1 out of 3).

Reading Scheduler Reports

The following is useful when reading scheduler reports.

- Using more registers than the 128 available can cause a significance performance degradation. This is reported by **-report=treestat** and can be caused by optimizations such as inlining, unrolling, and grafting
- To detect hot spots, look for “maximum register live” in **-report=treestat**.

- Use the scheduling factor as an indicator of the parallelism of the algorithm. This is defined as the ratio of cycle time to cycle time on a machine with infinite resources.
- Use issues-per-cycle as an indicator of the parallelism achieved.
- Use numbers reported as "static" to determine performance for a single decision tree execution. Use numbers reported as "dynamic" for the whole program. To be significant, these require compiling with **tmcc -r** (and a previous run).
- Use **-report=treestat** to verify the effectiveness of elimination of **rdreg** and **wrreg** operations.

Decision Tree Syntax

This section explains the decision tree syntax and describes what is entailed in optimization of intermediate code. Decision trees are an intermediate representation between C and assembler. They are as efficient as assembler, but because the scheduler is used to enforce pipeline constraints, trees code allows for easier programming and maintenance. To understand this, it helps to compile a small file with the **-t** option.

What is a Decision Tree?

A decision tree (DT) is the work unit for the Instruction Scheduler. Some characteristics of decision trees are described below.

- A decision tree starts wherever control flows to one point from more than one point.
- A decision tree can contain constructs such as **if**···**else** and **while**.
- A merge in control after the construct ends the tree. This is called a *join*.
- A call to a function ends the decision tree.
- A decision tree can have multiple exits.
- The size of a decision tree is chosen by the compiler.
- Information from profiling is used when available.

A single decision tree can be used to represent the code below:.

```
if (cond1)
   if (cond2)
      if (cond3)
         a = 1;
      else
         a = 2;
   else
      a = 3;
else
   a = 5;
```

A join is the corresponding place back to which control flows after branching (after the last statement above, for example).

For more information about guarded decision trees, see *Guarded Execution* on page 80.

Control Flow

Consider the following example.

```
int prod( int a, int b, int c ){
   do{
      b *= c;
   }while( a-- );
   return b;
}
```

The input to the Instruction Scheduler can be obtained as follows (file prog.t).

```
tmcc -t prog.c
```

The content of a decision tree is a basic block. This terminates in a control divergence (**if-then-else** or **select-or)**, which specifies more basic blocks belonging to the current decision tree, or in a control transfer out of the decision tree (**gotree**, **nigotree**, **cgoto**, **nicgoto**).

- The basic block statements are always executed at the start.
- A control divergence is a branch selection between possible conditions coming together after a branch.
- A control transfer occurs because of a join.

The file prog.t has two decision trees, the first of which contains just a control transfer. This decision tree just goes to the **__prod_DT_1** (the main body of the function).

```
   .global _prod
_prod:
entree (0)
   gotree {__prod_DT_1} (* BB:2 *)
endtree (*__prod_DT_0*)
```

Below are some observations regarding this example.

- The **entree** construct corresponds to the beginning of a function.
- The **endtree** construct corresponds to the end of the tree.
- The value inside the parentheses of **entree** is the number of times it was executed (if measured by profiling).
- The text marked in **(* ... *)** are comments for debugging.

Possible control flow constructs are shown in Table 8.

Table 8 Control Flow Constructs

Construct	Description
gotree SymbolName **nigotree SymbolName**	Control transfers to the DT named by **SymbolName**. **nigotree** is the non-interruptible variant of **gotree**.
cgoto OperationName **nicgoto OperationName**	Control transfers to the address produced by **OperationName**. **nicgoto** is the non-interruptible variant of **cgoto**.
if Condition [**(Probability)**] **then DTree1** **else (Condition) DTree2** **end (Condition)**	If the **Condition** (operation name) has the Boolean value true, **DTree1** is executed, else **Dtree2** is executed. The optional probability specifies the probability that **DTree1** is executed.
Select **(Cond1)** [**(Probability)**] **: DT** **or** **(Cond2)** [**(Probability)**] **: DT** **...** **default** [**(Probability)**] **: DT** **endselect**	If **Cond1** (operation name) has the Boolean value true, the DT associated with that case is executed, or if **Cond2** is true, the DT associated with that case is executed, and so on. If none of the conditions are true, the default case (presence required) is executed. The optional probabilities specify the probability of execution for each case. The conditions (**Cond1**, **Cond2**, **Cond***n*) must be mutually exclusive. The probabilities should sum to 1.0.

Operations

Code for the second decision tree is shown below. This tree has two branches. The first branch loops back; the second corresponds to the return.

```
tree (0)
        1 rdreg (2);
        2 rdreg (5);
        3 rdreg (7);
        4 rdreg (6);
        5 imul 4 3 ;
        6 isubi (1) 2 ;
        7 uneqi (0) 2 ;
        8 wrreg (6) 5   after 4;
        if 7 (0.900000) then
                10 wrreg (5) 6  after 2;
                gotree {__prod_DT_1}
        else (7)
                11 wrreg (5) 5  after 2;
                cgoto 1
        end (7)
endtree
```

Below are some observations regarding this example.

- **tree** is like **entree** except that this is not an entry point of a function.
- The value in parentheses of an **if** corresponds to the probability of **if** being taken.
- The value in parentheses of **else** and **end** is used for matching.

There is an operation number for each individual assignment to a variable. There is an operation number for every temporary value in an expression. Redundant values are eliminated by the compiler.

For documentation on an individual operation, refer to Appendix A of the appropriate TriMedia data book.

Operation Syntax

This section explains the syntax of an individual operation. Consider the example given below.

```
12 if 11 st32d(4) 2 3 after 10;
```

This operation can be decomposed into fields as follows.

Field	Value	Required	Description
Op#	12	Y	Identifies the operation.
Guard	11	N	Guard register. Preceded by **if**.
Opcode	st32d	1	Machine operation.
Displacement	4	N	Immediate value. Between parentheses.
Arguments	2 3	N	Operations for arguments.
After constraints	10	N	Operations for ordering constraints. Preceded by **after** keyword.

- Displacements are for operations with an immediate value and for **rdreg**/**wrreg** operations.
- Arguments correspond to previously defined expressions.
- A semicolon terminates the operation.
- Guarded operations are generated by the **-if -convert** option.

Pseudo-Operations

The **rdreg** and **wrreg** operations do not map to machine operations. Instead, they establish a correspondence between operation numbers and hardware registers. The correspondence for the example shown is given in Table 9.

Table 9 Pseudo-Operations Correspondence

Op#	Hardware Register	Value
1	r2	Return address.
2	r5	Parameter **a**.
3	r7	Parameter **c**.
4	r6	Parameter **b**.

- **rdreg(n)** maps the hardware register **n** to the input of an operation.
- **wrreg (n) m** maps the output of operation **m** to hardware register **n**.
- The operation number for a **wrreg** operation can be referred to for ordering.
- Register-to-register moves are generated in some cases.

Table 10 shows the register usage convention.

Table 10 Register Usage Convention

Register	Definition
r1	**r1** is predefined as **1**.
rp (r2)	**rp** is the return pointer. On entry to a function, it contains the return address. For framed functions, **rp** is not saved. For frameless functions, **rp** can be saved in a register to speed up the function return.
fp (r3)	**fp** is the frame pointer. It points at the base of the current stack frame. **fp** is not always updated and is, strictly speaking, not part of the calling convention. For programs compiled with **-g**, the frame pointer is always updated.
sp (r4)	**sp** is the stack pointer. It points at the last word in use by the current stack frame.
rv (r5)	**rv** is the return value register. If the return value is a scalar, it is returned in **rv**. The return of **struct** or **union** values is via copy on exit to an address supplied as a hidden incoming argument.

Table 10 Register Usage Convention

Register	Definition
r5 .. r8	Registers **r5** through **r8** inclusive are argument registers. The first four function arguments of basic type are placed in the argument registers. Note that **r5** is the return value register as well as the first argument register.
r9 .. rn-1	Registers **r9** through **rn-1** inclusive comprise the global register pool. These registers are used by the compiler for global register allocation. In this pool, registers **r9** up to and including **r32** are callee save registers, registers **r33** up to and including **r63** are caller save registers. Note that **n** is currently 64.
rn .. r127	Registers **rn** through **r127** inclusive comprise the decision tree local register pool. Note that **n** is currently 64.

After Constraints

Operation ordering is expressed by the keyword **after**, followed by a list of operations. The keyword imposes a partial order between pairs of operations. The meaning is that the operation specified can be issued, at the earliest, one cycle after all listed operations.

For the purposes of ordering, operations can be divided into the following 5 categories.

Reg	rdreg or wrreg pseudo operation.
Mem	load or store operation.
Jump	gotree, nigotree, cgoto, nicgoto.
Join	join pseudo operation.
Other	Other Databook Operations.

Below are three uses for **after** constraints.

- Specifying that a load from memory must take place after a store (for pointers, for example).
- Specifying that a read from a register must take place after a write to that register.
- Specifying general purpose constraints (delay).

Not all pairs of operations permit **after** constraints. The following table defines allowed pairs of operations.

Source Operation	**Target Operation**				
	Reg	**Mem**	**Jump**	**Join**	**Other**
Reg	YES	YES	YES	NO	NO
Mem	YES	YES	YES	YES	YES

Source Operation	Target Operation				
	Reg	**Mem**	**Jump**	**Join**	**Other**
Jump	NO	NO	NO	NO	NO
Join	NO	NO	YES	NO	YES
Other	NO	NO	YES	NO	YES

- An operation cannot have a jump on the list of its incoming constraints.
- A join operation is not allowed to have a list of constraints

Guarded Execution

TriMedia Compilation System v.2.0 supports guarded execution in the decision tree format, as shown in the example below:.

```
20 if 10 st32 11 12;
```

The **store** operation is executed only if the **lsb** of value **10** is true. Furthermore, the pseudo operation named **join** is supported to merge values. For example.

```
20 if 10 iaddi (4) 11;
21 if 12 isubi (4) 13;
22 join 20 21;
```

If the **lsb** of value **10** is true, value **22** will be equal to value **20** (**val 11+4**). In case **lsb** of value **12** is true, value **22** will be equal to value **21** (**val 13-4**). The **lsbs** of **10** and **12** should not be true at the same time. In the case both are false, the value of **22** is undefined.

In general, **joins** can be used to merge two or more values. These values should be guarded such that at most one of the guards is true when the values are merged.

Guarded execution is used when the **-if_convert** and **-if_param** flags are given to **tmccom**.

Debug Information

In both the trees code and the assembler, the following information is placed by the compiler:

- **.fileinfo stabs** and **stabn** information is global to the module for debugging.
- **.funcinfo stabs** and **.funcinfo stabn** are local to the function.
- **.treeinfo** information is local to the decision tree.
- **.treeinfo regmask** defines the registers used by the compiler. This means the others out of the 128 available are free for the scheduler.
- **.treeinfo label** defines the label of this decision tree.

Embedded Assembler Directives

In addition to debug information, the trees code can contain embedded assembler directives, as shown in the table below.

Assembly Directive	Description
.align n	Advance current address to the next **n** bytes aligned address in the current segment.
.ascii "string"	Generate the ASCII equivalent of the string in the current segment, allowing all the Standard C escape sequences in the string.
.byte list-of-expressions	Generates initialized 1-byte values in the current segment, given the comma-separated list of assembly expressions.
.common symbol, size ["segment" [, alignment]]	Declares the symbol to be a FORTRAN-style common area with the given **size** in bytes. If **symbol** is given, it must be **bss**. In any case, the symbol is defined in **bss**. The **symbol** can have an optional alignment that is used by the linker during its final linking pass (if the symbol did not resolve to a definition).
.data	Switches current segment to the **data** segment.
.data1	Switches current segment to the **data1** segment.
.global list-of-symbols	Declares the comma separated list of symbols to be global.
.half list-of-expressions	Generates initialized 2-byte values in the current segment, given the comma-separated list of assembly expressions.
.reserve symbol, size [,"segment" [, alignment]]	Defines **symbol** in current or optionally given **segment**, reserves size bytes. Align it if given optional **alignment** size (in bytes).
.skip n	Skip the next **n** bytes (advance current address by **n**) in the current segment (not allowed in the **text** segment).
.text	Switches current segment to the **text** segment.
.word list-of-expressions	Generates initialized 4-byte values in the current segment, given the comma-separated list of assembly expressions.
.zero n	Zero the next **n** bytes (and advance current address by **n**) in the current segment (not allowed in the **text** segment).

Segments

The assembler has a concept of "current segment" as it parses the assembly source code. The current segment is initially set to the **text** segment at the beginning of a module. The current segment can be switched to other segments via the assembler directives. The assembler currently supports the following segments: the **text** segment for instructions, the **data1** segments for read-only initialized data, the **data** segment for read-and-write initialized data, and the **bss** segment for uninitialized data.

Labels/Symbols

Labels define a symbol to be at the current address in the current segment. Labels are specified by the symbol name followed by a colon. For segment directives, see *Embedded Assembler Directives* on page 81.

When using labels, keep the following in mind.

- Labels must be followed by instructions.
- Code label arithmetic is not recommended.
- Do not use labels as markers, since the linker is free to reorder dtrees.

Chapter 5

Performance Analysis Overview

Introduction

This chapter explains the program analysis **tmprof**. This tool produces a breakdown on where your program is spending its time.

- You can use **tmprof** in combination with **tmsim -statfile** for simulation.
- You can use **tmprof** in combination with **tmcc -ptm** for actual execution.
- You can use **tmprof** on stand-alone systems.

The next sections of this chapter discuss the command syntax and **tmprof** options, followed by discussions on how to use **tmprof** with the simulator and with a host processor.

Important Guidelines for Making Measurements

Using **tmprof** is straightforward. However, as always, use care when making measurements. It is easy to misinterpret results. Here are some important guidelines:

1. Use a fast board (e.g., a 143-Mhz TM-1300 with a 1:1 SDRAM clock ratio) for measurements.
2. Make sure that the speculative load fix option is disabled when measuring.
3. The major change in the profile consists of the introduction of statistical sampling. The default is to sample the entire program.
4. Full measurements slow the program down by a factor of about 2.5.
5. Stripped library addresses are indicated as “unknown address” lines.
6. Use the profileStart and profileStop functions on context switch for task-based sampling.
7. The overhead is under your control with statistical sampling.
8. Good values are -profdly 20000 -profcnt 20.
9. With statistical sampling, make at least 30 second runs.
10. Decision tree and instruction cycles are exact on the simulator and hardware.
11. Because of measurement interference, cache overhead can be overestimated on the hardware.
12. Use **tmsim** with the **-ns** (no startup) option when comparing.
13. Reported instruction and data cache cycles may be higher than in reality.
14. Idle time can increase with statistical sampling because of overhead reduction.
15. Two timers are required for measurements and are allocated using the API.
16. Applications that use hard-coded timer addresses or more than two timers will not work.

17. You must quit the program or use profileFlush to get any data.
18. You might need to adjust the size of the profiling buffer.
19. Separately compute overhead using DP and the cycles instruction.

Command Syntax

- Options are described in the following section.
- **tracefile** can be a trace file from **tmsim** or "mon.out."
- The executable is the name of your program.

tmprof Options

There are several classes of options for **tmprof**, as described in the sections below.

Formatting Options

Output can be produced in two basic formats, *standard* and *detailed*. The detailed format is invoked by the keyword **-detail**. When no option is given, **tmprof** produces a report in standard format.

Standard Format

The standard format reports the number of executions, total cycles, and numbers of instruction and data cache cycles, as illustrated in Table 11. Cycle totals are in thousands, and include cache overhead and copybacks.

Table 11 Standard tmprof Reporting Format

Treename	Executions	Total Cycles (%)		Inst. Cache Cycles	Data Cache Cycles
_fib	13529	68	40.11	0	0
_fib_DT_1	6764	47	28.06	0	0
_fib_DT_2	6764	27	16.06	0	0
_start	1	2	1.25	2	0
Total/Average	28047	169	100.00	18	2

The threshold is set to 0.0100000. Exact total machine cycles is 168,721 cycles.

Below are some observations regarding this report.

- The first column is the name of a decision tree or function.
- The second column is the number of executions.

- The remaining columns correspond to total, instruction, and data cache cycles. These do not include time spent in called functions.
- Adding the **-wide** option uses 39-character names.

Detailed Format

The detailed format prints statistics on issues and the CPI (Cycles Per Instruction). This is specified by using the option **-detail**.
Table 12 illustrates the **tmprof** reporting format that results from the **-detail** option.

Table 12 Detailed tmprof Reporting Format

	(Instructions + Stall)			Instructions		Inst. Cache Stalls	
Treename	**Total Cycles**	**MCS (%)**	**CPI**	**Cycles**	**MCS (%)**	**Cycles**	**CPI**
_fib	67680	47.58	1.00	67645	47.61	29	0.00
__fib_DT_1	47348	33.28	1.00	47348	33.33	0	0.00
__fib_DT_2	27089	19.04	1.00	27056	19.04	33	0.00
_main	62	0.04	15.50	4	0.00	58	14.50
_main_DT_1	62	0.04	15.50	4	0.00	58	14.50
Total/Average	142252	100.00	1.00	142068		178	0.00

Data Cache Stalls			(Number per Instruction)		
Cycles	**CPI**	**Cycles**	**Copybacks**	**Issues**	**(useful)**
6	0.00	0	0	2.60	(1.90)
0	0.00	0	0	0.86	(0.85)
0	0.00	0	0	1.00	(1.00)
0	0.00	0	0	1.75	(1.75)
0	0.00	0	0	0.50	(0.50)
6	0.00	0	0	1.71	1.38

Following are some observations regarding this report.

- With no cache overhead, CPI = 1.
- CPI figures across a row total.
- The final columns indicate operations and useful issues.
- Bank conflicts are reported.
- Adding the **-wide** option uses 39-character names.

Scaling Options (-scale and -threshold)

This section explains the **-scale** and **-threshold** options. These can be used to scale down the data to manageable amounts. Table 13 lists the **tmprof** scaling options.

Table 13 tmprof Scaling Options

Option	Description
-scale nn	Sets the execution time scale to **nn**. The parameter **nn** is a number by which to divide cycle counts. Default is **-scale 1000**.
-threshold ff	Sets the report threshold. The parameter **ff** is a floating point number. Entries that contribute less are not shown. Default is **-threshold 0.0001**.

Grouping Options (-func and -fcs)

This section explains the **-fcs** and **-func** options. These can be used to group the output by function. Table 14 lists the **tmprof** grouping options.

Table 14 tmprof Grouping Options

Option	Description
-func	Reports functions instead of trees. Totals cycle counts and executions per function.
-fcs <name>	Limits report to a single function called **name**.
-fcs ··· -fcs ···	Limits reports to a range. **-fcs low_address -fcs high_address** limits the output to an address range.

Run-Time Options (-ptm)

This section explains run-time options that are interpreted when you run. The **tmprof** run-time library uses statistical sampling. The size of the buffer, the frequency between samples, and the number of decision trees captured can be specified as shown. These options apply to an executable compiled with **tmcc -ptm**. Table 15 lists the tmprof run-time options.

Table 15 tmprof Run-Time Options

Option	Description
-profsz size	Trace buffer is size bytes.
-profdly ccount	Sample frequency is ccount cycles.
-profcnt dcount	Sample is dcount decision trees in length.

Below are some observations regarding these options.

- The default trace buffer size is approximately 16 percent of the code size.
- The trace buffer size is rounded up to a power of 2.

- Use **-profdly 100 -profcnt 1** to sample everything.
- Use the **tmprof -dump-header** option to find out occupancy. See *Miscellaneous Options* on page 88).
- 32 bytes per decision tree are required in the trace buffer.

Pros

- Lower sampling frequencies mean lower overhead.
- A smaller cache buffer means less memory and cache overhead.

Cons

- Occupancy greater than 50 percent is bad because of hashing.
- Significance depends on sufficient sampling frequency.

Caveats

The following overhead and measurement errors pertain to the use of the **-ptm** option.

- Profiling overhead increases run time by 150 percent of trees sampled. For this reason, cycle counts correspond only to a percentage of executions.
- By default, profiling overhead increases memory overhead by about 16 percent of the text size.
- Profiling overhead can artificially increase instruction and data cache cycles.

Miscellaneous Options

Table 16 lists the tmprof miscellaneous options.

Table 16 tmprof Miscellaneous Options

Option	Description
-h	Prints a help message.
-V	Prints version information.
-genstat	Converts from mon.out to **tmsim -statfile** format. **-genstat** translates the mon.out file into **tmsim -statfile** format. This option can be used for post-processing.
-clockspeed <n>	Specifies clock speed for MCS. Used to specify the processor speed.
-dump-header	Prints trace header.
-group	Group addresses in one entry.
-mcs	Reports an MCS factor, not a percentage. (See below.)

The following explains the handling of unknown addresses and the **-group** option:

- Application libraries are stripped to prevent disassembly.
- Addresses that have been stripped are reported numerically.
- The **-group** option reports these as a single item.
- This can be used to estimate OS overhead if only **pSOS** in being used.

MCS Factor

tmprof includes an option to calculate the so-called "MCS" factor for a component.

- This is meaningful when running with real-time input and output data.
- An operating system must be used to provide blocking.
- For non real-time tasks, the MCS factor corresponds to percentage utilization at 100 MHz.

The MCS factor is computed as:

$$\frac{\text{Cycles for Entry}}{\text{Total Cycles}} \times \text{Clock Frequency}$$

For example, suppose a data measurement is being taken on a 125 MHz processor corresponding to 150 million cycles. Assume the measured time in MPEG component is 60 million cycles, in AC-3 40 million cycles, and in a third component 50 million cycles.

Given these assumptions, then

- The real time for the sample is 1.25 seconds (150 divided by 125).
- The MCS factor is 50 for MPEG, 33 for AC-3, and 42 for the third component.
- Two of the three can be run on a 100 MHz processor.

Using tmprof with the Simulator

This section shows you how to use **tmprof** with the simulator **tmsim**. A small example program is shown below.

```
main(int argc, char **argv){
   printf("%d\n", fib(atoi(argv[1])));
}
fib( n ){
   if(n <= 2 ) return 1;
   return fib(n-1) + fib(n-2);
}
```

This program can be compiled and run with **tmsim** as follows to generate the output in Table 11 on page 85.

```
tmcc prog.c
tmsim -statfile foo.stat a.out 20
tmprof -threshold 0.01 foo.stat
```

Caveats

The points below explain differences between cycle counts reported by **tmprof** and by **tmsim** with the **-v** option.

- Using **tmprof**, the 30 cycles that are required to fill the pipeline on program startup are not counted.
- System calls are counted as zero cycles by **tmprof** and as one cycle by **tmsim**.
- When an instruction cache stall and data cache stall occur simultaneously, two stalls are reported by **tmprof** for one cycle.
- Cycle count discrepancies can also be observed as a result of differences in behavior because of redirection.

Using tmprof with a Host Processor

The program above can be compiled and run with **tmsim** as shown below:.

```
tmcc -host Win95 -ptm prog.c
tmrun -b a.out
```

- Copyback cycles, data conflicts, and operations are not taken into account.
- The mon.out file has information similar to the statfile.
- The mon.out file is in binary format.

The following command displays the file:.

```
tmprof mon.out a.out
```

Depending on the information available, **tmprof** is capable of generating a variety of reports. The information from **tmcc -ptm** is based on statistical sampling

Pros

- Executing on the host is several thousand times faster than on the simulator.
- Using **tmcc -ptm**, you can obtain data about an application using peripherals.

Cons

- The **-g** and **-ptm** options are incompatible.
- The **-ptm** option perturbs the cache behavior, which may affect results in some cases.
- The simulator output includes operation statistics.

Standalone Programming Using the tmprof API

This section explains standalone programming using the TriMedia profiler API. The information here is useful if you do any of the following:

- Explicitly activate or deactivate profiling.
- Use **tmprof** in a standalone environment.
- Use **tmprof** in a multiprocessor environment
- Write a communications driver.
- Have already used standalone profiling and upgrade from the 1.1 release.

Explicit Activation and Deactivation of Profiling

You normally invoke **tmprof** using the **-ptm** option. The **-ptm** option has two effects:

1. The **-lprof** option is added to the **tmld** command line. The profiling library is in libprof.a.
2. Special code is linked that automatically activates and deactivates profiling. This code is in tmprof.o.

Explicit activation and deactivation of profiling requires linking in with **-lprof** but without tmprof.o. To do profiling requires the definition of some parameters and some changes to source code.

Defining Profiling Parameters

Parameters to initialize profiling must be defined. Parameters are set by defining fields in the structure **profileCaps**. The definition of the structure from tmlib/profile_api.h is shown here:

```
#define TMPROF_VERSION "TMPROF_V2.0\0\0\0\0"
#define PROFILE_ARGSIZ 64
struct profileCaps {
   char      version[16];
   char      args[PROFILE_ARGSIZ];
   int       nbytes;
   unsigned  nsamples;
   int       freq;
} ;
```

TMPROF_VERSION is used to check for compatibility between profile_api.h and libprof.a. The version field should be initialized with this string. **PROFILE_ARGSIZ** is the length of the command line. The args field must be initialized using the profileArgs function or using a dummy value.

The command line can be visualized with the **tmprof -dump-headers** option.

A buffer for profiling is allocated by tmprofInit using malloc.

If non-zero, the **nbytes** field specifies the buffer size. Otherwise, it is 16 percent of the text segment size. The term **freq** corresponds to the interval between samples. The term **nsamples** is the number of trees to capture.

These parameters correspond to the **-profdly**, **-profcnt**, and **-profsz** runtime command line options.

Source Code Changes

Three points must be considered for profiling.

- Code must be added to initialize profiling on startup.
- A callback function must be written to output data.
- Code can be added optionally to turn profiling on and off during execution.

The sample code shown below initializes profiling.

```
#include <tmlib/profile_api.h>
profileBegin(){
   struct profileCaps caps;
   int  handle;

   memset( &caps, 0, sizeof(caps) );

   caps.freq     = 10000;
   caps.nsamples = 20;
   strcpy(caps.version, TMPROF_VERSION  );
   strcpy(caps.args   , "<command line>");
   handle = open("mon.out",O_RDWR|O_CREAT|O_TRUNC|O_BINARY,0666);
   profileInit(&caps, write_callback, handle);
}
```

freq is set to generate a timer interrupt every 10,000 clock cycles.

nsamples is set to measure the behavior of 20 decision trees every sample.

Removing these two assignments means profiling the whole program. The arguments to profileInit are the profiling parameters, a pointer to the callback function, and an open file descriptor.

How to Write Output Data

Use a callback function to write the data. Call the profileFlush function when data is needed or at the end of the program.

Sample code for the callback function of the previous example is given below.

```
static int
write_callback( int handle, void *array, int nbytes ){
   if( nbytes == 0 )
      return close( handle );
   else
      return write( handle, array, nbytes );
}
```


The parameters are the same as the POSIX write system call. The first parameter identifies the file; it is a file descriptor from open in the previous example. The second and third parameters represent an array of characters.

All the data must be written to a file on the host system, in the order specified, in binary format, without conversion or insertion. (The profiling library uses a 1K buffer.)

A byte count of zero means the profiled output is flushed to the host system. This can be done while the program is running.

Starting and Stopping Profiling

The profileStart function enables profiling (by generating a debugger interrupt). The debugger interrupt activates the profiling mechanism. To deal with the case where profiling is running already, the TFE (trace) bit is cleared and the timer is stopped.

The profileStop function disables profiling by stopping the timer and by clearing the TFE (trace) bit. The program below gives a simple example:

```
main() {
   profileBegin();
   profileStart();
   fib(20);
   profileStop();
   profileFlush();
}
fib( n ){
   if( n < 3 )
      return 1;
   else
      return fib(n-1) + fib(n-2);
}
```

Clearing the Profiling Buffer

Use the following function to clear the profiling buffer. Doing so allows you to take several samples during the execution of a program.

void profileClear (float percentage)

The single argument represents the maximum occupancy of the profiling table. The table is implemented as a hash buffer and occupancies greater than 50 percent result in increased overhead.

Only samples corresponding to decision trees in the table are recorded if the table overflows.

Adjusting the Size of the Profiling Buffer

The **profileDtrees** function allows you to adjust the size of the profiling buffer. It is called to estimate the default value for **nbytes**. The following provides an example that reduces the buffer to save space.

```
caps.nbytes = profileDtrees()*0.5;
...
profileInit( &caps, write_callback, handle );
```

Pros

Memory overhead is reduced.

Cache overhead is reduced.

Cons

If the buffer is too small, the operation is incorrect.

Command Line Processing

Command line processing can optionally be performed to initialize the profiling parameters. This section explains how to use the profileArgs function.

int profileArgs(struct profileCaps *pcaps, char **argv, int argc)

The first argument points to the profiling parameters, which are initialized to default values. The second and third arguments correspond to the vector of command line arguments, as commonly implemented in C. The function returns the argument count.

Task-Based Profiling

Profiling information can be captured on the task level or on the component level. The following example shows how to perform task-based profiling.

```
void root(void) {
   printf("Hello, world\n");
   t_create( "aaaa",   4, 10000, 10000, 0, &task1 );
   t_create( "catc", 100, 10000, 10000, 0, &task2 );
   profileBegin();
   ...
}
void SwitchHandler(int to, void *totcb, int from, int *fromtcb) {
   int clocknow = CYCLES();

   if (from==task1) TOTAL += clocknow - NOW;
   if (to==task1) NOW = clocknow;
   if (to==task1)
      profileStart();
   else
      profileStop();
}
```


The function **root** is the pSOS main task/function. In addition to **root**, this code creates two tasks, **task1** and **task2**. The root calls profileStart when switching context to **task1** and profileStop when switching context from **task1**. SwitchHandler is a pSOS callback function.

Summary

The following functions comprise the profiling API and use the structure **profileCaps**, previously described.

- **int profileArgs(struct profileCaps *pcaps, char **argv, int argc)**
- **void profileInit(struct profileCaps *pcaps, int (*writefunc)(), int handle)**
- **void profileStart(void)**
- **void profileStop(void)**
- **void profileClear(float percentage)**
- **void profileFlush(void)**

Caveats Regarding Profiling

Be aware of the following points:

- Standalone applications might need to be run with statistical sampling.
- For correct operation under pSOS, use either full or statistical sampling.
- A buffer that is too small can result in wrong behavior.
- With stripped libraries, the output is incorrect.

Chapter 6

Systems Programming

Introduction

This chapter explains how to use the systems programming features of the TriMedia compilers and environment. It uses C examples, source code for which is provided with this release. Refer to source code if you have any problems understanding what is being described.

See Book 2, *Cookbook*, for examples of video and audio drivers.

See Books 5–9 for information about how to use APIs.

Systems Program Debugging

The DP feature of the TriMedia C library allows debugging **printf**s (DPs). This section explains how to use DP.

- DPs are written to RAM.
- DPs and the normal dialog are not mixed up.
- DPs can be used in a stand-alone environment.
- **printf** cannot be used for interrupt and exception handlers, while DP can.
- DPs are much faster.

To use debugging **printf**s, a special SDRAM area must be reserved as a circular buffer. The program below shows an example of the feature.

```
#include <stdio.h>
#include <tmlib/dprintf.h>

main(){
   DP_START(1024,0);
   printf("Hello World\n");
   DP(("Bonjour tout le monde\n"))
   exit(0);
}
```

- To compile, enter the following command line.

```
tmcc -host Win95 program.c -o program.out
```

- To run, use **tmgmon**.
- You should see “Hello World” in the console window and “Bonjour tout le monde” when you click “Dump DP.”
- Both DP and **printf** use parentheses, but DP uses an extra pair.
- The size of the DP buffer is 1024.

Assertions

The **tmAssert** macro is approximately equivalent to the UNIX **assert**. Assertions are quite important for embedded development.

In the following example, the assertion checks for a null pointer. The second parameter is an error code.

```
#include <tmlib/tmassert.h>
UInt32 aoGetNumberOfUnits( UInt32 *pNumberOfUnits ){
   ...
   tmAssert( pNumberOfUnits != NULL, TMLIBDEV_ERR_NULL_PARAM )
   ...
}
```

The assertion, when triggered, prints filename, line number and error code. Error messages are written to stderr using DP. The assertion causes the program to abort.

Interrupt Handlers

This section explains how to program interrupt handlers. Code for interrupt management can be divided into main and interrupt level.

- A C function for the interrupt handler must be written.
- The main function has to set the vector to the handler.
- Single decision trees have minimal overhead.
- Interrupts must be enabled in the main program.
- An atomic function uses scheduler registers so the overhead is lower.
- Interrupt handlers can be interruptible or non-interruptible.
- Overhead is controllable.

Writing an Interrupt Handler

This section shows an example of an interrupt handler.

```
#include <tmlib/dprintf.h>

intInstanceSetup_t isetup;

volatile int count;

void handler(void){

#pragma TCS_handler
   DP(("count = %d\n", count));
   count++;
}
```

- The parameter list of the handler must be declared **void**.

- The result type of the handler must be declared **void**.
- You must use volatile variables to communicate between interrupt handlers and the main loop.
- Do not call the C library for I/O inside a handler.
- Library utility functions (for example, **memcpy**, **strlen**, **sin**) can still be used.

Initializing an Interrupt Vector

This section explains how to program an interrupt vector. The code below sets up a timer interrupt.

```
#include "tm1/tmTimers.h"
#include "tm1/tmInterrupts.h"

main(){
   timInstanceSetup_t setup;
   DP_START(1024,0);
   setup.source = timCLOCK;
   setup.prescale = 1;
   setup.modulus = 1000000;
   setup.handler = Handler;
   setup.priority = intPRIO_4;
   setup.running = 1;
   (void) timOpen(&timer);
   (void) timInstanceSetup(timer, &setup);

   count = 0;

   while (count < 1000){}
}
```

- This code is the main level for the example above.
- The device library is used.
- **timInstanceSetup** calls **intInstanceSetup** to set up the interrupt.

The code below shows how to set up a interrupt using **intInstanceSetup**.

- This uses the debug interrupt (level 30).
- The code for **gen_interrupt** is shown below .

```
#include "tm1/mmio.h"
#define SOFTWARE_INT 30
#include "tm1/tmInterrupts.h"
#include "tmlib/dprintf.h"

intInstanceSetup_t isetup;

main(){
   int i, here;

   isetup.enabled = 1;
   isetup.handler = handler;
   isetup.level_triggered = 0;
   isetup.priority = intPRIO_6;
```



```
    intOpen(PROFILE_INT);
    intInstanceSetup(SOFTWARE_INT, &isetup);

    for (i = 0; i<10; i++) {
        here = count;
        gen_interrupt(SOFTWARE_INT);
    }
}
```

Generating a Software Interrupt

Software interrupts can be used for scheduling asynchronous events. The code example below illustrates how to generate a software interrupt. For information on interrupt MMIO registers, refer to Chapter 3 of the appropriate TriMedia data book.

```
gen_interrupt( int x ){
    intAckClear(x);
    intAckPending(x);
    /* wait for interrupt to go away */
    while (intCheckPending (x)){}
}
```

Reducing Interrupt Overhead

This section explains how to reduce interrupt overhead. Minimum overhead is achieved if the handler is programmed as a single decision tree. An example is shown below.

```
void handler( void ){
#pragma TCS_handler
    buf[count++] = ix;
}
```

- There is no need to save registers in one decision tree.
- There is no need to turn off interrupts since decision trees are non-interruptible.
- The difference from a normal function is the address to return to.

You can also reduce overhead using the compiler pragma **TCS_atomic**. This allows you to program more than one decision tree inside a handler with less overhead.

```
void handler( void ){

#pragma TCS_handler
#pragma TCS_atomic
    ...
}
```

- You need to use the option **-tmccom -allow_atomic_calls** –– to call other functions.
- These functions must also be declared as atomic. (Be careful because the compiler does not check).

Pros

- Code size is reduced.

- Interrupt response time is reduced also.

Cons

- Interrupt latency is increased.

Interruptible Handlers

An interruptible handler is like a normal handler, except that it is declared using **#pragma TCS_interruptible_handler.**

Pros

- Latency is improved because the handler is interruptible.

Cons

- The handler has to be programmed to be re-entrant.

Exception Handlers

This section explains how to program an exception handler. An example is shown below.

```
#include "tm1/mmio.h"
#include "ops/custom_defs.h"
#include "tmlib/dprintf.h"

void handler( unsigned spc ){
   #pragma TCS_exception_handler
   writepcsw(0x00000000, 0x8000);
   DP(("During exception, spc = %x\n", spc));
}

main(){"
   int a[2], i, p;
   DP_START(1024,0);
   MMIO(EXCVEC) = (int)handler;
   writepcsw(0x80000000, 0x80000000);
   DP(("Before exception\n"));
   for( i=0; i<1000; i++ ){}

   p = (int) a;
   *(int *)(p + 1) = 0;
   for( i=0; i<1000; i++ ){}

   DP(("After exception\n"));
}
```

- This is an example of the unaligned access exception.
- The store through **p+1** is to an illegal odd address.
- Bit 31 in the PCSW validates the exception.

- Bit 15 in the PCSW registers the exception
- These are turned on and off in main and handler, respectively.
- The main function sets bit 31 to trap on this exception.
- Refer to Chapter 3 of the appropriate TriMedia data book for more information.

Critical Sections

A *critical section* is code that, when it is executed, must be executed completely and cannot be interrupted. The following program provides an example.

```
#include <tmlib/dprintf.h>
#include "tm1/tmTimers.h"

volatile int count, loops, interrupts;

critical_section(){
   int old = count;
#pragma TCS_break_dtree
   count = old + 1;
}
void handler(void){
#pragma TCS_handler
   critical_section();
   ++interrupts;
}
main(){
   timInstanceSetup_t setup;
   int timer, i;

   DP_START(1024,0);
   setup.source = timCLOCK;
   setup.prescale = 1;
   setup.modulus = 100;
   setup.handler = handler;
   setup.priority = intPRIO_4;
   setup.running = 1;
   (void) timOpen(&timer);
   (void) timInstanceSetup(timer, &setup);

   for (i=0; i<50; i++) {
      ++loops;
      critical_section();
   }
   DP(("LOOPS      +  INTERRUPTS    = TOTAL COUNT\n"));
   DP(("%3d        +  %3d           = %3d\n", loops, interrupts,count));
}
```

This produces the following output.

```
LOOPS      +  INTERRUPTS    = TOTAL COUNT
50         +  28            = 73
```

- A **#pragma TCS_break_dtree** is used to artificially introduce an interruptible jump.
- Five increments of **count** are not accounted for.
- The number corresponds to interruptions because of the jump.

A single decision tree is a natural critical section. In the example, this corresponds to removing the jump.

```
critical_section(){
   int old = count;
   count = old + 1;
}
```

The output numbers then sum up as shown below.

```
LOOPS      +  INTERRUPTS    = TOTAL COUNT
50         +  30            = 80
```

Using an Atomic Function

An *atomic* function is a function in which interrupts do not occur. The following shows an example.

```
critical_section(){

#pragma TCS_atomic
   int old = count;
#pragma TCS_break_dtree
   count = old + 1;
}
```

- With **TCS_atomic**, non-interruptible jumps are used inside the function and the output totals up.
- The compiler uses scheduler registers, which reduces overhead.

Pros

- Overhead is reduced by the use of scheduler registers.

Cons

- Interrupt latency is increased.

Atomic Functions and Procedure Calls

It may be necessary to call a function inside a critical section, as shown in the example below.

```
critical_section(){
   #pragma TCS_atomic
   sub_func();
}
sub_func(){
   #pragma TCS_atomic
   ++count;
}
```


Compiling this produces the following error message.

```
"ex3d.c", line 7: Atomic function critical_section may not perform calls
```

This program can be compiled as follows.

```
tmcc -tmccom -allow_atomic_calls_ -- ex3d.c
```

Decision Tree Breaks

This section explains how to use decision tree breaks to control timing with hardware. These can be used to insert delay cycles for an interrupt acknowledgment. The following example is taken from the interrupt handler for the "vivot" example.

```
votestISR(){
   #pragma TCS_handler
   ...

   voAckBFR1_ACK();
   #pragma TCS_break_dtree
   return;
}
```

- For a level triggered interrupt the interrupt acknowledge must be issued at least two cycles before the ijump which ends the handler.
- The #pragma TCS_break_dtree adds a delay of four cycles.
- For more information, see section 3.4.3.3 of the databook.

The pragma can also be used for scheduling interrupts, as shown below.

```
{
   (long sequence of instructions)
   #pragma TCS_break_dtree
   (long sequence of instructions)
}
```

- Interrupts only occur on decision tree jumps, which the pragma forces.
- In this example, if the pragma is inserted in the middle, interrupt latency is halved.
- The pragma can also be reduced to reduce spills.
- For more information, see Chapter 4, *Using the Instruction Scheduler*.

Caller Save Registers

This section explains caller save registers.

- If a function or handler is a leaf function, it uses caller save registers.
- Since there are no function calls, these do not have to be saved at all.
- This can be overwritten using a pragma **TCS_no_caller_save**.
- The pragma needs to be used inside a function, otherwise it will not work.

Software Cache Support

This section tells you how to use custom operations that manipulate the cache.

Cache Copyback

TriMedia uses a write-back cache, which means that updates from the processor are not made immediately, but only when the cache line is needed. This poses a problem that can be illustrated in a PC development system using the IREF board.

- On the PC the SDRAM from the IREF board is memory mapped.
- The memory dump utility of **tmgmon** can be used to examine an address.

An example program is shown below.

```
#include <tmlib/dprintf.h>
#include <ops/custom_ops.h>
#define SZBUF 16

main( int argc, char **argv ){
   int i, wd1, wd2, flags;
   int *buf, *after;
   DP_START(1024,0);
   sscanf(argv[1], "%x", &fill);
   sscanf(argv[3], "%x", &flag);
   buf = (int *)_cache_malloc(SZBUF*sizeof(int));
   after = (int *)_cache_malloc(SZBUF*sizeof(int));
   for (i=0; i<SZBUF; i++) after[i] = fill;
   dump("           MEMORY ARRAY BEFORE (%08x)\n", buf);
   dump("           MEMORY ARRAY AFTER\n", after);
   for (i=0; i<SZBUF; i++) buf[i] = after[i];
   if (flag) copyback(buf, 4);

   for(;;){}
}
dump( char *fmt, int *buf ){
   int i;
   DP((fmt, buf));
   for( i = 0; i<SZBUF; i+= 4 ){
     DP(("buf[%2d]: %08x %08x %08x %08x\n", i, buf[i], buf[i+1],
        buf[i+2], buf[i+3]));
   }
}
```

This program illustrates the effect of the cache copyback operation.

- To compile, use **tmcc -host Win95 ex4b.c -o ex4b.out**.
- To run, use **tmgmon**.
- The program prints the buffer address.
- To view **buf** in SDRAM, specify **DWORD** with the address in **tmgmon**.
- To view using the cache, use **dump DP** in **tmgmon**.

The command-line arguments must be specified. The first argument corresponds to a fill pattern. The second argument is a flag indicating the use of copyback. Running the pro-

gram without copyback shows the SDRAM as being equal to the old value. Running the program with copyback shows the memory as one would expect it to be.

Cache Invalidate

TriMedia does not see the effect of writes outside the CPU to the cache. This means that cache contents must be invalidated after a DMA operation, or when getting data from another processor.

- On the PC, the SDRAM from the IREF board is memory mapped.
- The memory dump utility of **tmgmon** can be used to patch an address.

An example program is shown below.

```
#include <tmlib/dprintf.h>
#include <ops/custom_ops.h>
#define SZBUF 16
#define CLOCK_SPEED     100000000

main( int argc, char **argv ){
   int i, fill, flag;
   int *buf, *after, secs;
   DP_START(1024,0);
   sscanf(argv[1], "%x", &fill);
   sscanf(argv[2], "%d", &secs);
   sscanf(argv[3], "%x", &flag);
   buf = (int *)_cache_malloc(SZBUF*sizeof(int));
   after = (int *)_cache_malloc(SZBUF*sizeof(int));
   for(i=0; i<SZBUF; i++ ) buf[i] = fill;
   copyback(buf, 4);
   dump("           MEMORY ARRAY BEFORE (%08x)\n", buf);
   pause(secs*CLOCK_SPEED);
   if( flag ) invalidate(buf, 4);
   for( i=0; i<SZBUF; i++ ) after[i] = buf[i];
   dump("           MEMORY ARRAY AFTER (%08x)\n", after);
   for(;;){}
}
dump(char *fmt, int *buf){
   int i;
   DP((fmt, buf));
   for (i = 0; i<SZBUF; i+= 4) {
      DP(("buf[%2d]: %08x %08x %08x %08x\n", i, buf[i], buf[i+1],
         buf[i+2], buf[i+3]));
  }
}
pause(unsigned clicks){
   unsigned end = cycles() + clicks;

   while( cycles() < end ){}
}
```

This program illustrates the effect of the cache invalidate operation.

- To compile, use **tmcc -host Win95 ex4c.c -o ex4c.out**.
- To run, use **tmgmon**.
- To view using the cache, use **dump DP**.

- The program prints the buffer address, then pauses.
- To patch **buf** in SDRAM, specify **DWORD** with the address of the buffer.
- The buffer can be patched during the pause by typing over the old data.
- To view the new contents after the pause, use **dump DP**.

The command-line arguments must be specified. The first argument corresponds to a fill pattern. The second argument is the time to wait before the update (the number of seconds at 100 MHz). The third argument is whether to invalidate the cache. Running the program without invalidate shows the SDRAM as being equal to the old value. Running the program with invalidate shows the memory as one would expect it to be.

Miscellaneous Issues

Code Checksumming

TriMedia architecture does not protect the code segment and it is possible for a store to a wrong address to destroy part of your program. This occurrence is extremely difficult to detect. Such behavior is also possible because of a DMA write by a peripheral to a wrong address. In a PC environment, the board is memory mapped and destruction can be caused by your host program

The program below shows how to checksum your code. **_tmprof_init** is a stub that is linked at the end of the **tmld** command line. You can adapt this code to do checksumming from the host.

```
int fib(n){
   if (n <= 3) return 1;
   return fib(n-1) + fib(n-2);
}
int checksum( unsigned char *from, unsigned char *to ){
   int sum = 0;

   while (from != to) sum += *from++;
   return sum;
}

extern unsigned char _start[];
extern unsigned char _tmprof_init[];

main(){
   printf("checksum = %x\n", checksum(_start, _tmprof_init));
   fib(20);
   printf("checksum = %x\n", checksum(_start, _tmprof_init));
}
```

Uninitialized Variables

Automatic variables are not initialized automatically. The content of the variable depends on the previous stack contents.

Use of an uninitialized variable can cause strange and unpredictable behavior. One symptom of this is if your program is failing, but then works when you add a **printf**.

The compiler gives warnings about uninitialized automatic variables. It is important to pay attention to these warnings.

Race Conditions

A *race condition* is a condition that depends on timing, because of an interrupt or a peripheral. It corresponds to a bad use of a critical section. Race conditions are most frequently encountered in combination with a multitasking system such as pSOS

One symptom is if your program works when you compile with **-g**, then stops working at a normal level of optimization. Another symptom is if your program fails when compiled with a different level of optimization.

Race conditions are more likely in a multiprocessor systems and can be aggravated in combination with incorrect cache programming. The best way to avoid race conditions is to carefully program your critical sections

Chapter 7

Using Custom Operations

Introduction

This chapter explains how to use custom operations. By adding custom operations to your C program, you can take advantage of the highly parallel TriMedia implementation.

- Custom operations permit DSP efficient programming at the C level.
- Custom operations permit efficient programming of multimedia algorithms.
- Custom operations can be used for vector computation and operate on data in parallel (1, 2, or 4 elements).
- In some other compilers, custom operations are called *intrinsic operations*.

For more information, see Chapter 4, *Custom Operations for MultiMedia,* in the appropriate TriMedia data book. For examples of the use of custom operations, refer to Chapter 12, *Case Studies*, in Book 2, the *Cookbook*.

Syntax

Below are some observations regarding syntax for custom operations ("custom ops" for short).

- A custom operation is defined by a prototype like a C function, using the keyword "**custom_op**."
- The definition defines the type of the function and its operands.
- The operation should be from Appendix A of the appropriate TriMedia data book.
- The set of definitions is in the include file <ops/custom_defs.h>.
- Some databook operations map to more than one custom op because of type.
- The TM-1100 has additional custom ops.

An example of the definition of **dspiadd** taken from <ops/custom_defs.h> is shown below. The **dspiadd** custom op is a signed 32-bit saturating add.

```
custom_op long   dspiadd(long a, long b);
```


Classes of Custom Operations

Custom operations can be divided into the following classes, which are described in more detail on the following pages.

Note
See Appendix A of the appropriate TriMedia data book for the exact semantics of each operation.

In the following tables, the "Result Type" refers to a single element, and the "#operands" refers to the definition of the custom op.

Operations on Vectors of Four Elements

The operations shown below operate on two 32-bit words treated as two four-element vectors (8 bits).

Result Type	Operation	#Operands	Function
unsigned char	dspuquadaddui	2	Quad clipped add.
unsigned	quadumulmsb	2	Quad multiply.
unsigned	quadavg	2	Quad byte average.

- **dspuquaddaddui** adds two vectors of four bytes.
- **quadumulmsb** multiplies two vectors of four bytes.

- **quadavg** computes the mean vector between two vectors of four bytes.
- **dspuquaddaddui** is useful for adding an I-frame to a P-frame in MPEG. For this reason, the first argument is unsigned and the second is signed.
- **quadavg** can be used for superimposing two video images.
- **quadumulmsb** can be used prior to **quadavg** for alpha blending.
- The output of these instructions is clipped to an unsigned byte (see below).

Operations on Vectors of Two Elements

The operations shown below operate on two 32-bit words treated as two two-element vectors (16 bits).

Result Type	Operation	#Operands	Function
signed short	dualiclipi	2	Dual 16 bit signed clip.
unsigned short	dualuclipi	2	Dual 16 bit unsigned clip.
signed short	dspidualabs	2	Dual signed absolute value 16 bits.
signed short	dspidualadd	2	Dual signed add 16 bits.
signed short	dspidualsub	2	Dual signed subtract 16 bits.
signed short	dspidualmul	2	Dual signed multiply 16 bits.

- Audio data is often encoded in 16 bits.
- **dspidualadd** adds two vectors together of 16 bits.
- **dspidualsub** subtracts two vectors together of 16 bits.
- The result of these instructions is clipped (see below).
- These are useful for DSP applications, the Discrete Cosine Transform (DCT), and complex arithmetic.
- The clip instructions limit the output to the range of the second operand.
- This is useful for the DCT and for saturated arithmetic in DCT applications.
- **dspidualabs** computes the absolute value and clips.
- **dspidualmul** multiplies two vectors together of 16 bits.
- The result is clipped to a signed range (see below) and corresponds to the low-order 16 bits.
- **dualiclipi** and **dualuclipi** are not available on the TM-1000.

Vector-to-Scalar Computation

The operations shown below perform vector-to-scalar computation.

Result Type	Operation	#Operands	Function
signed short	ifir16	2	Dual signed scalar product.
unsigned short	ufir16	2	Dual unsigned scalar product.
signed char	ifir8ii	2	Quad scalar product.
signed char	ifir8ui	2	Quad scalar product.
signed char	ufir8uu	2	Quad scalar product.
signed char	ume8ii	2	Quad sum of absolute value of signed differences).
unsigned char	ume8uu	2	Quad sum of absolute value of unsigned differences.

- These instructions operate on a vector and produce a scalar result.
- **ume8uu** computes the distance vector between the inputs and sums up.
- The "fir" instructions compute a sum of products for two or four elements.
- The result is represented in full precision (32 bits).
- The **ifir16** and **ufir16** instructions are useful for complex arithmetic, for the DCT, for the Fast Fourier Transform, and in vector processing and DSP applications.
- The **ume8ii** and **ume8uu** instructions are useful for motion estimation in video applications.
- The **fir** instructions on 8-bit data operate on twice as much data, but with half the precision at the same speed.

Multiple Precision Arithmetic

The operations shown below are useful in multiple precision arithmetic.

Result Type	Operation	#Operands	Function
unsigned	carry	2	Carry from unsigned add.
unsigned	borrow	2	Carry from unsigned subtract.
signed long	imulm	2	Multiply high-order 32 bits.
unsigned long	umulm	2	Multiply high-order 32 bits.

- The carry and borrow instructions can be used to add vectors of long numbers.
- The **imulm** and **umulm** instructions can be used for DSP fractional arithmetic in combination with a multiply.

- These instructions are useful for cryptographic and other applications.

Clipped Computation

The operations shown below perform clipped computation.

Result Type	Operation	#Operands	Function
signed long	dspiadd	2	Clipped 32-bit add.
unsigned long	dspuadd	2	Clipped 32-bit add.
signed long	dspimul	2	Clipped 32-bit multiply.
unsigned long	dspumul	2	Clipped 32-bit multiply.
signed long	dspisub	2	Clipped 32-bit subtract.
signed long	iclipi	2	32-bit clipping.
unsigned long	uclipi	2	32-bit clipping.
unsigned long	uclipu	2	32-bit clipping.
unsigned long	iavgeoneop	2	Average of two operands.

- These instructions are useful for DSP applications.
- **dspiadd** adds two numbers but saturates in case of positive or negative overflow.
- **dspiadd** is the same for unsigned numbers.
- **dspisub** subtracts two numbers.
- The clip instructions are useful to limit a value to a range.
- Unsigned clipping is the same as a minimum computation.
- Signed clipping limits the result to [**-M-1** to **M**].

Floating Point

The operations shown below perform operations on floating-point values.

Result Type	Operation	#Operands	Function
float	fsqrt	1	Floating square root.
float	fabsval	1	Floating absolute value.
float	fsign	1	Sign of floating point value.
float	ifloatrz	1	Convert integer to floating.
float	ufloatrz	1	Convert integer to floating.
signed long	ifixieee	1	Convert floating to integer.

Result Type	Operation	#Operands	Function
unsigned long	ufixieee	1	Convert floating to integer.
signed long	ifixrz	1	Convert floating to integer.
signed long	ufixrz	1	Convert floating to integer.

- These instructions are useful for getting better performance with floating point.
- The **fabsval** and **fsign** instructions eliminate branches that impact performance on VLIW.
- The **fsqrt** routine computes a square root in 17 cycles.
- For **fsign**, the result is -1, 0. or 1 depending on the sign of the argument.
- The **ifixrz** and **ufixrz** instructions can be used to perform rounding to zero as specified by C.
- The **ifixieee** and **ufixieee** use the four rounding modes specified by the IEEE-754 standard.

Vector Data Packing and Rearrangement

The operations shown below operate on vectors of two or four elements. They perform data packing and rearrangement.

Result Type	Operation	#Operands	Function
short	pack16lsb	2	Pack 16 least significant bits.
short	pack16msb	2	Pack 16 most significant bits.
char	packbytes	2	Pack least significant bytes.
char	mergemsb	2	Merge most significant bits.
char	mergelsb	2	Merge least significant bytes.
long	funshift1	2	Funnel shift.
long	funshift2	2	Funnel shift.
long	funshift3	2	Funnel shift.
unsigned char	ubytesel	2	Byte select.
signed char	ibytesel	2	Byte select.

- Rearrangement of data is frequently necessary in signal processing and vector computation.
- The funnel shifts are useful for filters, for the fixed-point FFT, and for motion estimation.
- **pack16msb** and **pack16lsb** are useful for the DCT and its inverse.

- **mergemsb** and **mergelsb** are useful for the DCT in MPEG.
- Funnel shifts can be used to perform unaligned data accesses.
- For more information, refer to Appendix A of the appropriate TriMedia data book.

Minimum, Maximum, and Absolute Value

The operations shown below perform minimum, maximum, and absolute value computations.

Result Type	Operation	#Operands	Function
signed long	imin	2	Integer minimum.
signed long	imax	2	Integer maximum.
unsigned	umin	2	Integer minimum.
unsigned	uclipi	2	Integer maximum.
float	fmin	2	Floating minimum.
float	fmax	2	Floating maximum.
unsigned char	quadumin	2	Quad minimum.
unsigned char	quadumax	2	Quad maximum.
signed long	iabs	2	Absolute value.
signed long	dspiabs	2	Clipped absolute value.

- Minimum and maximum instructions replace branches that impact performance on VLIW architectures.
- The **quadumin** and **quadumax** operate on four elements at a time. These operations are not available on the TM-1000.
- These greatly increase the performance of median filtering.
- This is a key algorithm in interlaced-to-progressive scan conversion and image processing.
- **iabs** computes the absolute value in one cycle.
- **dspiabs** computes the absolute value in two cycles and clips.

Shift and Rotate

The operations shown below perform shift and rotate computations.

Result Type	Operation	#Operands	Function
signed short	dualasr	2	Dual arithmetic shift right.
long	rol	2	Rotate left.
long	roli	2	Rotate left immediate.

- The rotate instructions are useful for compression and decompression of multimedia bit streams.
- The **rol** instruction takes a variable count and **roli** takes a fixed count.
- The **dualasr** instructions are useful for implementation of 9-bit conformant MPEG video.
- Normal C syntax can be used for left and right shifts (signed and unsigned).
- **dualasr** is not available on the TM-1000.

Processor Control

The operations shown below perform processor control.

Result Type	Operation	#Operands	Function
long	readpcsw	0	Read processor control and status word.
void	writepcsw	2	Write processor control and status word.
unsigned long	cycles	0	Read clock cycle counter, lsb.
unsigned long	hicycles	0	Read clock cycle counter, msb.
unsigned long	curcyles	0	Read current clock cycle counter, lsb.
unsigned long	readspc	0	Read source program counter.
unsigned long	readdpc	0	Read destination program counter.
void	writedpc	1	Write destination program counter.
void	writespc	1	Write source program counter.

- These instructions are useful for performance analysis and signal processing.
- Using the cycles counter, the exact time in a code segment can be bracketed.
- This is also useful for timestamps in multimedia streams.
- The processor order is programmable using **readpcsw**.

- The second argument is a mask.
- The last four instructions are used to program interrupt and exception handlers for the IEEG rounding mode.

Cache Control

The operations shown below are useful for cache control. A combination of hardware and software is used for cache control on TriMedia.

Result Type	Operation	#Operands	Function
void	copyback	2	Update memory from cache.
void	invalidate	2	Discard memory in cache.
void	prefetch	2	Preload memory to cache.
void	allocate	2	Discard contents.

- Cache control uses a combination of hardware and software on TriMedia.
- The **copyback** and **invalidate** instructions are useful for programming with DMA peripherals, when using TriMedia in a PC system, and for multiprocessor TriMedia systems.
- The **prefetch** instruction is useful for improving cache performance for video-conferencing.
- The **allocate** instruction is useful for reserving a stack frame for compilers.
- The starting and ending addresses for these instructions should be cache-aligned (64 bytes).
- Incorrect use will result in unpredictable results that are difficult to debug.

Conditional Computation

The operations shown below are useful for conditional computation.

Result Type	Operation	#Operands	Function
signed long	inonzero	2	If non-zero, select zero.
float	fnonzero	2	If non-zero, select zero.
signed long	iflip	2	If non-zero, negate.
signed long	izero	2	If zero, select zero.
float	fzero	2	If zero, select zero.
long	mux	3	Select between two arguments.
float	fmux	3	Select between two arguments.

- These instructions eliminate branches that impact performance on VLIW architectures.
- **mux** and **fmux** replace a C conditional operator (**?:**) with a guard.
- **mux** and **fmux** are scheduler operations.
- The "zero" instructions replace a conditional operator where one of the results is zero.
- The V2.0 compiler can generate code automatically for these operations in most cases.

Chapter 8

TriMedia C/C++ Languages

Introduction

This chapter defines the C programming language as implemented for the 32-bit variants of the TriMedia family of microprocessors. This chapter only describes the differences between the TriMedia implementation of the C programming language, the ISO C language standard, and various C dialects. It is not intended as a complete language reference.

The following sections describe standards and compatibility, extensions to ISO C, and implementation issues.

Standards and Compatibility

Relevant Standards

This implementation of the C programming language is based on the following standards.

- *American National Standard for Programming Languages—C*, ANS X3.159—1989
- ISO/IEC 9899:1990
- *Technical Corrigendum 1 (1994) to* ISO/IEC 9899:1990
- *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE 754-1985

Additionally, the compiler supports the concept of *restricted pointers*, as proposed by the *Numerical C extensions group* in

- X3J11/95-049, WG 14/N448,

available from **ftp://ftp.dmk.com/DMK/sc22wg14/c9x/aliasing/**.

Note that *"ANS X3.159—1989"* is the basis for the later ISO standard "*ISO/IEC 9899:1990*," which includes the prior standard in its entirety, with only minor editorial changes, notably section renumbering. All prior references to "ANSI C" or "Standard C" are equivalent to, and replaced by, references to ISO C, since ISO working group WG14 now controls the common C standard.

The ISO C standard has also been adopted as Federal Information Processing Standard (FIPS) 160.

Compatibility Considerations

The TriMedia C compiler is primarily an ANSI C compiler. However, some extensions have been added. These extensions are ones either likely to be approved in future standard revisions, or ones in common use in other ANSI C compilers. Note that language

extensions in conflict with the ISO C language standard are implemented under control of compile time switches, allowing for verification of compliance with the standard.

See Chapter 2, *Using the C Compiler*, for details regarding compiler options.

Additional Reading

For specific details on the TriMedia processor, refer to the appropriate TriMedia data book.

For a complete definition of the ISO C language or programming in C, consult the following resources:

The C Programming Language (Second Edition)
Kernighan & Ritchie
Prentice-Hall
ISBN 0-13-110362-8

The Annotated ANSI-C Standard, ANSI/ISO 9899-1990
Annotated by Herbert Schild
Osborne/McGraw-Hill
ISBN 0-07-881952-0

C, A Reference Manual (Fourth Edition)
Samuel P. Harbison & Guy L. Steele Jr.
Prentice-Hall
ISBN 0-13-326224-3

Language Extensions

Alternate Extended Reserved Words

The following are extended reserved words. These reserved words are not defined or allowed by the ISO C standard.

- **custom_op**
- **pragma**
- **restrict**
- **inline**

The compiler provides the following alternate reserved words as replacements when ISO C compliance requires that recognition of the extended reserved words be disabled.

- **__custom_op__**
- **__pragma__**
- **__restrict__**
- **__inline__**

To avoid potential conflicts with otherwise standard conforming programs, we recommend you use the alternate reserved word form. You can easily redefine the alternate reserved words as the extended reserved words.

Future implementations may not support the extended reserved word form.

Custom Operators

Custom operators are provided to allow direct access to all machine-level operations from the C programming level. For the most commonly used operators there is an include file custom_defs.h that provides prototypes. This file must be included with

```
#include <ops/custom_defs.h>
```

This file defines the custom operations in capitals, which allows the user to compile and run the program on the TriMedia system, as well as on the native host.[1]

During TCS compilation, the definitions in custom_defs.h are mapped to the definitions that reside in the file custom_ops.h, which describes the most commonly used operators in a syntax that the TriMedia compiler can understand. Only the operations described in **custom_ops.h** are officially supported at the current time. Although other machine oper-

1. Compilation and execution on the native host system are unsupported features of the compilation system. Include files may be found in the directory include/<native host>/custom_ops (for example, include/SunOS/custom_ops). This directory has to be added to the include file search path with the -I option to the native compiler. Furthermore, accompanying object files need to be linked in to the application. These may be found in the directory lib/<native host>/custom_ops (for example, lib/SunOS/custom_ops).

ations will be accepted by the compiler, they have not received the extensive testing that the ones in **custom_ops.h** have. In any case, the compiler knows what the prototype should look like based on the description of the machine. Any prototype declared differently from this internal description will be flagged as in error.

Custom operations are declared as ISO C prototype function declarations using the storage class specifier **custom_op**. The use or declaration of a custom operator has exactly the same semantics as a prototyped ISO C function declaration or call, with the following additional constraints.

- A custom operation is not an addressable object.
- The **custom_op** specifier only applies to custom operator declarations.
- The declarative and executable forms of parameterized and unparameterized custom operations must match.
- The parameter argument to a parameterized custom operation must be an integer constant expression and is not allowed in the declarative form.

If custom operations are memory-accessing operations (such as loads or stores) they are assumed **volatile** and the compiler generates dependency constraints.

Parameterized custom operators are those hardware operations that require an integer modifier. For instance, the first parameter in an **iaddi(long immediate_value, long value)** instruction is an **opcode** modifier and should be a constant or a constant expression.

Refer to the appropriate TriMedia data book for more information on machine operations.

The compiler generates all appropriate conversions as for a prototyped function call and returns the integer type result of the application of the machine operation **dspiadd** to j and 1, as shown in Figure 7.

```
int i,j;
custom_op long dspiadd(long, long) ;
i = 2 + dspiadd(j,1) + 3 ;
```

Figure 7 Definition and Use of an Unparameterized custom_op

Figure 8 shows the use of a parameterized custom operation **iaddi**. The parameter "2" is passed through to be generated as the opcode modifier of the emitted operation.

```
int i,j;
custom_op long iaddi(unsigned long, long) ;
i = 2 + iaddi(2, j) + 3;
```

Figure 8 Definition and Use of a Parameterized custom_op

The V2.0 TriMedia Compilation System, like the V1.1 TriMedia Compilation System, supports custom ops in C++.

The Pragma Statement

A pragma statement has been added to the language. The compiler ignores all undefined syntactically correct pragmas, and produces a warning for those.

The syntax of the pragma statement is

```
pragma <expr>;
#pragma <expr>
```

where **<expr>** is any syntactically well-formed expression in the ISO C language. The expression is the pragma directive and, in general, is a name, a list of names, or some simple expression form. An ill-formed expression will cause an unavoidable syntax error.

For a list of supported pragmas, see *C Language Pragmas* in Chapter 2.

Note
Because pragmas are treated as statements, you must enclose a block of statements that includes a pragma in braces. The **else** section below gives an example:

```
if( a )
   b: ...
else {
   #pragma break_dtree
   c: ...
}
```

The asm Statement

The compiler does not support the **asm** statement. Specific machine operations are explicitly available at the "C" level.

Restrict

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 (WG14/N448, X3J11/95-049) may be found at **ftp://ftp.dmk.com/DMK/sc22wg14/c9x/aliasing**.

The **restrict** reserved word is added syntactically to the set of type qualifiers. It can be applied only to pointer types. Declaring a pointer to be restricted is an assertion by the programmer that no variable and no other restricted pointer will be used as alias for the object that the pointer references for as long as the restricted pointer is in scope. However, aliasing via a "normal" pointer is still allowed; that is, a nonrestricted pointer in the same scope as the restricted pointer may point to the same object.

Declaring a pointer to be restricted gives it aliasing properties similar to those of variables. In C, variables define private, non-overlapping memory regions. The C compiler need not generate ordering constraints between loads or stores, either from or to distinct variables, because it knows the memory references must refer to distinct objects.

Aliasing problems for the C compiler arise from the use of pointers. Without extensive analysis, the compiler must assume that each global variable and each local variable for which the address has been taken may alias with any pointer. Therefore, the compiler must be conservative and generate ordering constraints between the memory accesses of such variables and the memory accesses via pointers.

In many cases, a pointer argument to a function references a private memory region that the function accesses exclusively through the pointer. Often, the region is a variable that is not even within the scope of the function. An example of this is shown in Figure 9, where function **foo** passes references to local array variables **lsample** and **left_time** to function **subbandsyn**. In this case, the programmer knows these pointers will never alias with each other, nor with any variable in scope, but since this information is not available to the compiler in function **subbandsyn**, the compiler must assume that these pointers might alias with each other or with any global variable.

```
void subbandsyn(float *ip, int *op){
   ...
}
foo( void ){
   float lsample[96];
   int left_time[96];
   subbandsyn(lsample, left_time );
   ...
}
```

Figure 9 Passing References to Variables

The **restrict** qualifier lets you tell the compiler that a pointer reference does not alias, so it allows the compiler to generate more efficient code. Figure 10 shows how function **subbandsyn** in the example could be modified to assert that the function arguments **ip** and **op** point to private nonoverlapping memory regions.

IMPORTANT
It is the responsibility of the programmer to guarantee the correctness of the assertions implied by the use of **restrict**.

```
void subbandsyn( float * restrict ip, int * restrict op ){
...
}
```

Figure 10 Making Pointers Restricted

Restricted pointers can improve the performance of a program. However, when improperly used, the compiler may generate code for the program that does not operate as expected.

As a type qualifier, **restrict** only has meaning for **lvalues**. In other words, only locations containing pointers can be restricted, and the assertion applies to the pointer contained by the location. There is no further restriction on the use of **restrict**: any variable of pointer type may be restricted, including local variables, global variables, function arguments, structure members, and array elements. Figure 11 shows some examples of

restricted pointer declarations. In the function **foo**, all of the pointers in scope are declared restricted. The compiler may assume that each pointer must reference a distinct object. However, the compiler may not be able to determine whether two address expressions refer to the same restricted pointer (for example, **restricted_ptrs[*a]** and **restricted_ptrs[*c]**). In this situation, the pointers, although restricted, are assumed to alias.

```
typedef struct {
   int * restrict p
} MyStruct, *MyStructPtr;

typedef int * restrict Rintp;

Rintp   restrict_ints[345];
MyStructPtr   restrict_ptrs[345];

extern Rintp restrict_int;

foo( Rintp a, MyStruct b ){
    Rintp c;

/* All pointers currently in scope are restricted and therefore must point to
 * distinct objects. */
}
```

Figure 11 Examples of Restricted Pointers

Long Float

The **long float** type specifier is a synonym for **double**.

This extension is minimally useful and is provided for compatibility reasons only.

Constants

The overflow of an integer, float, wide character, or string escape constant generates a warning, rather than an error. The TriMedia C compiler allows for turning off all warning-level diagnostics.

The ISO C standard requires a diagnostic be issued on constant overflow.

Bitfields

The TriMedia C compiler allows **signed** and **unsigned** bitfields of **char**, **short**, or **long** types, as well as **int** type.

The ISO C standard allows only **signed** or **unsigned** bitfields of **int** type.

The maximum number of bits in a bitfield is equivalent to the number of bits in the specified type.

In little-endian mode, the bitfield allocation is compatible with the Microsoft Version 2.2 C compiler.

Implementation

The information in this section is intended for programmers who require detailed knowledge of the implementation to write system-level interface routines, library support, or hand-optimized code.

This section defines machine-specific aspects of the C programming language as implemented for the TriMedia processor series. It includes sections on basic data representation, alignment, parameter passing, function call, function return and stack, register, and memory usage conventions.

Data Representation

The following table summarizes the sizes of the basic data types in the TriMedia C language.

Type	Size (in Bytes)
char	1
short	2
int	4
long	4
float	4
double	4
long double	4 or 8
pointer	4

The integer data types **char**, **short**, **int**, and **long** are, by default, signed types.

The integer data types are expressed as two's complement 8-, 16- or 32-bit, signed or unsigned values. The default byte-ordering is big-endian.

Without the **-uselongdouble64** option, all floating-point data types are represented as 32-bit, single-precision values. A float value consists of a sign bit, followed by an 8-bit biased exponent, followed by a 23-bit mantissa (not including the hidden bit). Values of type **float**, **double**, and **long double** are stored in IEEE Floating Point Standard P754, single-precision representation, but denormalized values are not supported by this implementation. The floating-point representation is discussed in detail later in this chapter.

With the **-uselongdouble64** option, floating-point data types **float** and **double** are represented as 32-bit, single-precision values, as described above. Floating-point data type

long double is represented as a 64-bit, double-precision value. A long double value consists of a sign bit, followed by an 11-bit biased exponent, followed by a 52-bit mantissa (not including the hidden bit). Values of type **long double** are stored in IEEE Floating Point Standard P754 double-precision representation. Long double floating point operations are performed in software, not with TriMedia hardware operations.

Alignment Requirements

This table summarizes the alignments of the basic data types in the TriMedia C language.

Type	Alignment
char	1
short	2
int	4
long	4
float	4
double	4
long double	4
pointer	4

All variables must be aligned to 1-, 2- or 4-byte boundaries, depending on their types, as follows:

- Type **char** variables may be aligned on any byte boundary.
- Type **short** variables must be aligned on any half-word, or 2-byte, boundary.
- All other basic data types require full-word, or 4-byte, boundary alignment.
- *Arrays* require the same alignment as their element type, unless you use **-Xalign**.
- *Structures* and *unions* are aligned according to the alignment requirements of the largest member. They are padded internally to preserve the alignment requirements of their members and padded at the end to ensure that each element of an array of such a struct or union type would have proper alignment. In little-endian mode, the bit-field alignment is compatible with the Microsoft Version 2.2 C compiler.
- All pointer values are required to be properly aligned for the object type of the pointer.

Loading or storing a value using a pointer containing an improperly aligned address causes an undefined result. A common poor programming practice is to assume that all

data types are byte-addressable. The problem is compounded by its intractability, since such uses cannot be detected at compile time. The following is an example.

```
int foo( char *cp ){
   int *ip;
   ip = (int *) cp;           /* cp is improperly aligned for an int type */
   return *ip;                /* causing the result to be undefined */
}
```

Naming Conventions

For function and variable names, the compiler prepends an underscore (_) to the name. The compiler internally generates names of the form **_l.nn** (where *nn* represents an integer value) as labels for initializer expressions. Local **static** variables are renamed **_name.LSnn**.

Internal C library functions and system-support functions reserve the use of function and variable names that begin with an underscore to avoid clashes with user-defined names.

All implementation-defined extensions and macro names must have prepended and appended double underscores, (__), to the name, for example, **__name__**.

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. At least do not prefix any identifiers with the special string "**__0**".

Memory Layout

Memory can be categorized as dynamically or statically allocated. Statically allocated memory includes the compiler-generated **text**, **data1**, **data**, and **bss** segments. Dynamically allocated memory includes the **stack**, **heap**, and **available** memory areas.

Figure 12 shows the relative position of dynamic and static memory areas.

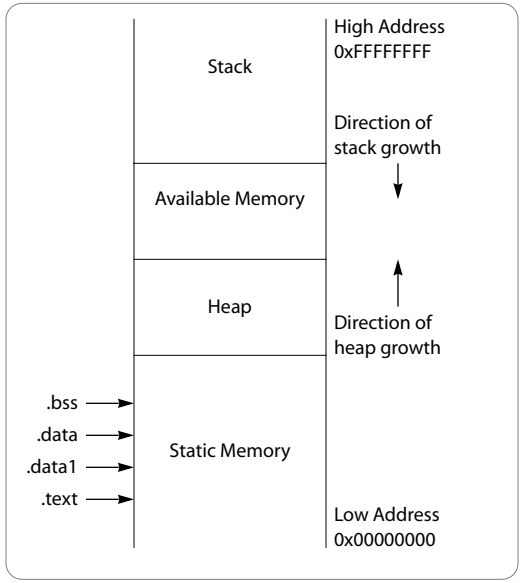

Figure 12 Memory Layout

Statically Allocated Memory

The **.data1** and **.text** segments are read-only initialized static memory. The compiler places all initialized data in the **.data** segment. The **.data** segment is writable.

The compiler places all uninitialized static or global variables in the **.bss** segment. The system guarantees that **.bss** is initialized to all zeros. The **.bss** segment is writable.

Dynamically Allocated Memory

The stack frame is bounded by the frame pointer and stack pointer registers. At program start, the stack pointer is initialized to the top of available memory. The stack grows down, toward the heap.

The heap is used by **malloc** via the underlying system call **_sbrk**. The heap grows up, toward the stack.

In a free-standing or embedded system environment, the programmer is responsible for ensuring that the stack and heap never collide.

Direct allocation of stack memory is achieved via the built-in compiler function **alloca**.

Register Usage Conventions

Table 17 defines the current C compiler usage conventions for the TriMedia processor register set. It is intended as generic reference, to provide background for later discussions on calling conventions.

Future compiler and scheduler register usage conventions may change as necessary.

Table 17 Register Usage Conventions

Register	Definition
r1	**r1** is predefined as 1.
rp (**r2**)	**rp** is the return pointer. On entry to a function, it contains the return address. For framed functions, **rp** is not saved. For frameless functions, **rp** can be saved in a register to speed up the function return.
fp (**r3**)	**fp** is the frame pointer. It points at the base of the current stack frame. **fp** is not always updated and is, strictly speaking, not part of the calling convention. For programs compiled with **-g**, the frame pointer is always updated.
sp (**r4**)	**sp** is the stack pointer. It points at the last word in use by the current stack frame.
rv (**r5**)	**rv** is the return value register. If the return value is a scalar, it is returned in **rv**. The return of struct or union values is through a copy, on exit, to an address supplied as a hidden incoming argument.
r5 to **r8**	Registers **r5** through **r8** inclusive are argument registers. The first four function arguments of basic type are placed in the argument registers. Note that **r5** is the return value register as well as the first argument register.
r9 to r*n*–1	Registers **r9** through r*n*–1 inclusive comprise the global register pool. These registers are used by the compiler for global register allocation. In this pool, registers **r9** up to and including **r32** are callee save registers, registers **r33** up to and including r63 are caller save registers. Note that *n* is currently 64.
r*n* .. **r127**	Registers r*n* through **r127** inclusive comprise the decision tree local register pool. Note that *n* is currently 64.

Callee vs. Caller Saved Registers

Except for very simple functions, all compiled C functions need a *register working set*, that is, a certain number of global registers for holding values during inter-decision tree jumps. Because any global register might already be allocated for use by the calling function, some convention between callers and callees must exist to ensure that none of the caller's values are lost during the call. Two well-known conventions for freeing such register sets are *callee saving* and *caller saving*. The TriMedia compiler uses both of these conventions.

Callee saving refers to the convention of saving the contents of each global register used in a function by the function itself. Typically, such registers are saved and restored in the

function's prologue and epilogue, which is the case for the TriMedia compiler. In this convention, the caller can trust that its register values will not be overwritten by the functions it calls. The disadvantage is that the callee generally does not know whether the registers it saves/restores actually contain important values (they might not be used by the caller).

Caller saving refers to the convention of letting the contents of the caller's registers be saved and restored by the caller itself. In this scheme, the callee can trust all the registers are available for use without the obligation of saving/restoring the original contents. The advantage here is that the caller can often let the saving and restoring be shared by several consecutive calls, thereby reducing the number of saves/restores during execution. The disadvantage, as in callee saving, is that the caller generally does not know whether the registers it frees are actually used by the callee. The caller saving convention can altogether prevent register saving/restoring for most leaf functions.

The TriMedia compiler implements callee and caller saving as follows: Registers 9 through 32 are defined as callee-saved; Registers 33 through 63 are defined as caller-saved. Caller-saved registers currently are only used by leaf functions, and are available for local use (by the scheduler) when not in use. Note with this current restriction the term "caller saving" is somewhat of a misnomer, since the caller-saved registers are only used in situations in which saving is unnecessary.

Calling Conventions

This section describes the function calling sequence. In general, function call overhead is almost entirely placed on function entry and exit, the assumption being that the compilation system will use procedure inlining whenever possible to eliminate expensive call sites.

There are several steps involved in executing a function call. At the call site, the arguments must be loaded prior to executing the call. At function entry, the new stack frame must be set up and the values of the callee-saved registers used by the function must be saved. At function exit, the old stack frame and the values of the callee-saved registers must be restored and the return value must be loaded.

Argument Passing

In standard C, all arguments are passed by value. Prior to the actual call operation, all argument expressions are evaluated. The resulting values are placed on the stack in the current frame's outgoing argument area. The evaluation of the argument expressions and the loading of the actual argument values are two discrete operations and generally cannot be interspersed.

The actual arguments are always promoted via the *default argument promotion* rules, regardless of whether the actual function called is defined with a prototype parameter list.

Excluding arguments placed in registers, the actual arguments are placed in ascending address order based on the stack pointer (**sp**). The implementation uses **sp** as the outgoing argument base register. The frame pointer is only used for large stack frames or for functions calling the compiler-supported **alloca** routine.

Up to four arguments are selected and passed in argument registers **r5** through **r8**. Register arguments are selected by scanning the argument list from left to right and taking the first four non-structure and non-union parameters. If a structure or union parameter is encountered during the scan, it is simply skipped and the scan continues. Parameters that live in registers do not have stack space allocated unless:

- The function is a **stdarg** function
- The program is compiled with the **-varargs** option;
- When parameters that could have been stored in registers would exceed the four registers used.

ANSI **stdargs** functions follow a slightly different calling convention. The fixed arguments (the ones before the ellipsis, if any) are passed using the normal calling convention rules outlined earlier. Starting at the ellipsis argument, all remaining arguments are unconditionally placed on the stack regardless of the number of argument registers still available.

Note
For a call to be considered a **stdargs** call, the called function's prototype must be in scope. A non-prototyped call to a **stdargs** function will not work correctly.

Old C **varargs** functions are unsupported. Currently, they work with some limitations. The first limitation is that no structures or unions may be passed to a **varargs** function. The second limitation is that the function definition must properly use the macros in varargs.h. The arguments are passed to a **varargs** function using the standard calling convention. The third limitation is that functions having old-style variable number of arguments and all functions calling these functions should be compiled with the **-varargs** option[1].

Structure-returning functions have a hidden first argument that identifies the address into which the returned structure value is stored. The compiler allocates a temporary (or uses a program variable if allowable) and generates the push of its address as part of the argument passing sequence for such a function.

Function Call

The return address is loaded in the return pointer register (**rp**). The branch to the target function is executed.

1. This is because of stackframe optimizations, which are, by default, on. The compiler must reserve stackspace at the caller of a varargs function to enable the varargs function to save the register parameters on the stack. Since for varargs functions there is not always a prototype in scope, it would mean that, without having the **-varargs** option, the compiler always had to reserve the stackspace.

At Entry

The old frame pointer value is saved when the frame pointer is used by this function. In that case, the frame pointer is set to the base of the current frame (old frame's **sp**). The stack pointer is set to the end of the current frame. The return pointer (**rp**) is saved, as are all function-level callee-saved registers. If necessary, the contents of the incoming argument registers are stored. For **varargs** functions, the arguments that contain variable parameters are always saved to their corresponding stack locations.

If the function is an *interrupt service routine* (or *handler*), the argument registers are saved, as well as the processor's destination program counter (DPC). This DPC contains the return address for handlers. For *non-interruptible* handlers, the interrupt enable bit (IEN) in the processor status word is cleared after a save of the old status word for later restoration of this bit.

If the function is an *exception handler* (see Chapter 11 of Book 2, the *Cookbook*), the argument registers are saved and DPC is stored to **r5**, the first incoming parameter of the exception handler.

At Exit

Upon function exit, the return value register (**rv**) is loaded (if used). The frame pointer (when used) and stack pointer are reset to their old values, all callee-saved registers are restored, and the return pointer is reloaded. Further actions depend on the type of function.

- In the case of a normal function, this reloaded return pointer is used as the return address.
- In case of an interrupt service routine (handler), the argument registers are restored and the saved DPC is reloaded and used as a return address.
- In the case of a noninterruptible handler, the saved processor status word is reloaded and used to restore the IEN.
- If the function is structure returning, the returned structure value is copied to the address supplied as the hidden structure return address argument.

After Return

After return from the called function, it is sometimes necessary to save the contents of the return value register (**rv***)* if the value of **rv** is required past intervening calls.

Atomic Functions

Atomic functions can be created with the **TCS_atomic** pragma. Atomic functions execute outside of the calling convention just described. The compiler will not use any of the computer's global registers. It uses registers from the scheduler's pool. The scheduler might choose to use a caller-saved global register.

Function and Handler Entry/Exit Optimizations

The TCS implements the following function and handler entry and exit optimizations.

- If the function/handler is a leaf function/handler (that is, if the function/handler has no outgoing calls), the return pointer register (**rp**) is not saved. **rp** can also be saved to a register to speed up the exit code.
- The frame pointer (**fp**) is not saved for most functions. Only functions with large stackframes of unknown size have a frame pointer.
- If the function/handler is a leaf function/handler it uses *caller save* registers. Since there are no function calls, these do not have to be saved at all. It is possible to overwrite this on either the command line or with a pragma **TCS_no_caller_save**.
- If the function/handler entry extended basic block has exit edges, those edges can use local register values rather than reading saved registers from memory.
- If there are no local variables or outgoing arguments, the entire entry/exit setup can be deleted. The function/handler then is a *frameless* function/handler.
- In case of a leaf handler, the argument registers are not saved.
- If the handler consists of a single extended basic block, neither the processor status word nor the DPC is saved.
- Atomic functions do not save or restore any of the global registers. They execute in the scheduler's register set.

Stack Conventions

The stack pointer (**sp**) is initialized at the start of the program to point to the top of available memory. As illustrated in Figure 13, the stack grows down from the top of memory to the end of the heap.

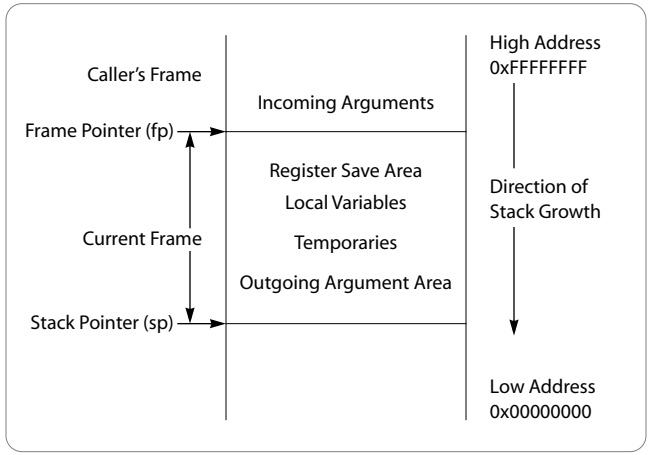

Figure 13 Stack Frame Organization

Stack Calculation

The C run-time system maintains a stack used as the system stack. But when pSOS is used, each pSOS task has its own stack. pSOS stacks can be monitored using the function **t_taskinfo**. A prototype is given in $TCS/OS/pSOS/pSOSystem/include/psos.h. The **t_taskinfo** function can be used to monitor the current size of the stack, and it also records a high watermark.

Incoming Arguments

Stack-based incoming arguments are located in the caller's frame. Incoming arguments are addressed in ascending order, based on the frame pointer (**fp**). The first four arguments of basic type are passed in registers and saved to the incoming argument area (if required) on function entry. As illustrated in Figure 14, the incoming argument area is

assumed to be allocated by the called function regardless of whether any arguments are stored there.

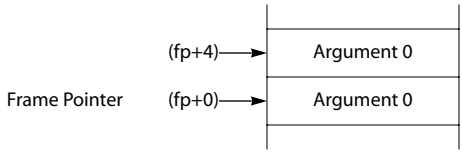

Figure 14 Stack-Based Incoming Arguments

Stack-based incoming parameters can be any of the following.

- Structures or unions.
- Parameters to a **stdargs** or **varargs** function.
- Parameters that exceed the number of parameter registers.

Register Save Area

The register save area is located in the current frame. Space is reserved only for save-on-entry global register uses. If **rp** and/or **fp** are saved, they are stored first in the frame, in that particular order. Other saved registers are saved in ascending order after **rp** and **fp**.

Outgoing Argument Area

Outgoing arguments are loaded in ascending address order based on the stack pointer (**sp**). Space is reserved for outgoing arguments beginning at (**sp** + 0). Since the first four words of arguments get passed in argument registers, the first argument actually placed on the stack is at (**sp** + 16).

The size of the outgoing argument area is the maximum size of all outgoing argument lists or 16 bytes minimum for the four argument registers for **stdarg** and **vararg** functions.

Note
A function using a variable argument list does not know the actual number of arguments passed and must assume that all argument registers are in use.

Implementation-Defined Behavior

The ISO C standard requires each implementation to document its behavior in each of the following areas. The behaviors are implementation-defined.

Note
The letters and numbers in parentheses refer to the relevant sections of the C language standard.

Environment (G.3.2)

Specific Area	Behavior
The semantics of the arguments to **main** (5.1.2.2.1)	The **main** function is passed two arguments, **argc**, which is the number of command-line arguments, and **argv**[ ] which is an array of pointers to the command-line arguments. At least one argument, the name of the program, is always passed to **main**. Currently, the simulator **tmsim** stores **argv**[ ] and the command line arguments at the bottom of the heap.
What constitutes an interactive device (5.1.2.3)	An interactive device is one of the following: an asynchronous terminal, a paired display and keyboard, or an interprogram connection.

Figure 15 illustrates passing arguments to **main**.

```
int main ( int argc, char *argv[] ){
...
}
```

Figure 15 Arguments to Main

Identifiers (G.3.3)

Specific Area	Behavior
The number of significant initial characters (beyond 31) in an identifier without external linkage (5.1.2)	The number of significant initial characters in an identifier without external linkage is over 2,048 characters.
The number of significant initial characters (beyond six) in an identifier with external linkage (5.1.2)	The number of significant initial characters in an identifier with external linkage is over 2,048 characters.
Whether case distinctions are significant in an identifier with external linkage. (5.1.2)	Case distinctions are significant in all identifiers, with or without external linkage.

Characters (G.3.4)

Specific Area	Behavior
To what do members of the source and execution character sets, except as explicitly specified in the standard, conform. (5.2.1)	The members of the source and execution character sets conform to the ASCII character set.
The shift states used for the encoding of multibyte characters (5.2.1.2)	There is no extended character set nor shift states for encoding of multibyte characters.
The number of bits in a character in the execution character set (5.2.4.2.1)	A character in the execution character set occupies eight bits.
The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.1.3.4)	The mapping of source and execution character sets is identical.

Specific Area	Behavior
The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (6.1.3.4)	Integer character constants are truncated so that only the first character or escape sequence is represented. Further, the value of an integer character constant is limited to the range of values representable by a **char** type object. Wide character constants are truncated so that only the first **wchar_t** size characters or escape sequences are represented. Further, the value of a wide character constant is limited to the range of values representable by a **wchar_t** type object.
The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (6.1.3.4)	The current locale is "C."
Whether a "plain" character has the same range of values as a signed character or unsigned character (6.2.1.1)	The default char type is signed and has the same range of values as type signed char.

Integers (G.3.5)

Specific Area	Behavior
The representations and sets of values of the various types of integers (6.1.2.5)	See Table 18 following.
The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2)	Converting to a shorter integer from a longer integer truncates the result value. Converting **unsigned** to **signed** of equal length does not alter the bit pattern of the result. In either case, the result may be negative.
The results of bitwise operations on signed integers (6.3)	Bitwise operations yield their results based on the bitwise evaluation of the operands, including any sign bits.
The sign of the remainder on integer division (6.3.5)	The result has the same sign as the dividend.
The result of a right shift of a negative-valued signed integral type (6.3.7)	The shift is signed. The sign of the operand is preserved.

Note
Integer division is implemented with a call to the floating-point division hardware. This may raise the sticky "INX" exception bit. Integer division neither uses nor clears this bit. This issue must be addressed by any developer wanting to use exceptions.

Table 18 Integer Type Representations and Values

Type	Bits	Minimum	Maximum
char	8	–128	127
signed char	8	–128	127
unsigned char	8	0	255
short	16	–32768	32767
signed short	16	–32768	32767
unsigned short	16	0	65535
int	32	–2147483648	2147483647
signed int	32	–2147483648	2147483647
unsigned int	32	0	4294967295
long	32	–2147483648	2147483647
signed long	32	–2147483648	2147483647
unsigned long	32	0	4294967295

Floating Point (G.3.6)

Specific Area	Behavior
The representations and sets of values of the various types of floating-point numbers (6.1.2.5)	Without the **-uselongdouble64** option, the representation of **float**, **double**, and **long double** values is the single-precision format specified in ANSI/IEEE Standard 754-1985, except that denormals are not supported by this implementation. The IEEE standard single-precision format consists of one sign bit, 8 exponent bits, and 23 significand bits. The exponent bits represent a binary exponent with bias 127, and an implicit hidden 1 bit followed by a binary point appears to the left of the most significant significand bit. Exponent 255 with nonzero significand represents NaN (quiet if the most significant significand bit is 1, signaling if it is 0); exponent 255 with zero significand represents +Infinity or -Infinity; exponent 0 with nonzero significand represents a denormalized value (not supported by this implementation); and exponent 0 with zero significand represents zero. All other values represent normalized values. Because types double and long double are represented as 32-bit single precision values, they do not satisfy some numerical limits requirements of the ANSI/ISO C Standard 9899-1990. In particular, section 5.2.4.2.2 Characteristics of floating types <float.h> mandates: DBL_DIG 10 LDBL_DIG 10 DBL_EPSILON 1E-9 LDBL_EPSILON 1E-9 but 32-bit single precision floating point does not satisfy these requirements. See the following table. With the **-uselongdouble64** option, the representation of **float** and **double** values is the IEEE single-precision format, as noted above, but the representation of **long double** values is the IEEE double-precision format. The IEEE standard double-precision format consists of one sign bit, 11 exponent bits, and 52 significand bits. The exponent bits represent a binary exponent with bias 1023, and an implicit hidden 1 bit followed by a binary point appears to the left of the most significant significand bit. Exponent 2047 with nonzero significand represents NaN (quiet if the most significant significand bit is 1, signaling if it is 0; exponent 2047 with zero significand indicates +infinity or –infinity; exponent 0 with nonzero significand represents a denormalized value; and exponent 0 with zero significand represents zero. All other values represent normalized values.

Specific Area	Behavior
The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3)	The direction of rounding is to the nearest representable value.
The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4)	The direction of rounding is to the nearest representable value.

Table 19 Floating Point Type Representation and Values without **-uselongdouble64**

Type	Bits	Minimum	Maximum
float	32	1.17549435e-38	3.40282347e+38
double	32	1.17549435e-38	3.40282347e+38
long double	32	1.17549435e-38	3.40282347e+38

Table 20 Floating Point Type Representation and Values with **-uselongdouble64**

Type	Bits	Minimum	Maximum
float	32	1.17549435e-38	3.40282347e+38
double	32	1.17549435e-38	3.40282347e+38
long double	64	2.2250738585072014E–308	1.7976931348623157E+308

Arrays and Pointers (G.3.7)

Specific Area	Behavior
The type of integer required to hold the maximum size of an array—that is, the type of the sizeof operator **size_t** (6.3.3.4, 7.1.1)	The type of **size_t** is defined in <common/_size_t.h> as **unsigned int**.
The result of casting a pointer to an integer or vice versa (6.3.4)	The value does not change, unless casting to a smaller integer type, which will truncate the result.
The type of integer required to hold the difference between two pointers to elements of the same array, **ptrdiff_t** (6.3.6, 7.1.1)	The type of **ptrdiff_t** is **int**.

Registers (G.3.8)

Specific Area	Behavior
The extent to which objects can actually be placed in registers by use of the register storage-class specifier (8.5.1)	Use of the **register** storage-class specifier has no effect on the placement of objects in registers.

Structures, Unions, Enumerations, and Bit-Fields (G.3.9)

Specific Area	Behavior
A member of a union object is accessed using a member of a different type (8.3.2.3)	The object value is accessed and treated according to the accessing member type.
The padding and alignment of members of structures (8.5.2.1)	See *Alignment Requirements* on page 132.
Whether a "plain" integer bitfield is treated as a signed integer bitfield or as an unsigned integer bitfield (8.5.2.1)	Bitfields of **int** type are treated as bitfields of **signed int** type.
The order of allocation of bitfields within an **int** (8.5.2.1)	In big-endian mode, bitfields are allocated in descending bit order within a storage unit. In little-endian mode, bitfield allocation is compatible with the Microsoft V2.2 C compiler.
Whether a bitfield can straddle a storage unit boundary (8.5.2.1)	Bitfields cannot straddle a storage-unit boundary.
The integer type chosen to represent the values of an enumeration type (8.5.2.2)	The values of an enumeration type are always represented by a **signed int**.

Qualifiers (G.3.10)

Specific Area	Behavior
What constitutes an access to an object that has volatile qualified type (8.5.3)	Any reference to the name of an object constitutes an *access* of that object.

Declarators (G.3.11)

Specific Area	Behavior
The maximum number of declarators that may modify an arithmetic, structure, or union type (8.5.4)	The number of declarators that may modify an arithmetic, structure, or union type is unlimited.

Statements (G.3.12)

Specific Area	Behavior
The maximum number of case values in a switch statement (8.6.4.2)	The number of **case** values in a **switch** statement is unlimited.

Preprocessing Directives (G.3.13)

Specific Area	Behavior
Whether the value of a single-byte character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set; whether such a character constant may have a negative value (6.8.1)	The value of a character constant in a preprocessing expression is the same as in any other context. Character constants may have negative values.
The method for locating includable source files (6.8.2)	The preprocessor searches for a file whose name is delimited by brackets (< >), first in directories named by the **-I** option, then in the standard header file directory (directory **include** in the TriMedia Compilation System distribution directory, typically /usr/local/tcs/include). The preprocessor searches for a file whose name is delimited by quotes, first in the directory containing the current source file, then in directories specified by the -I option, then in the standard header file directory.
The support of quoted names for includable source files (6.8.2)	Quoted file names are supported.
The mapping of source file character sequences (6.8.2)	All source file characters conform to their defined ASCII character codes.
The behavior on each recognized **#pragma** directive (6.8.6)	**#pragma** directives are translated into the internal *pragma statement* form. Refer to *The Pragma Statement* on page 128 for more information on the pragma statement language extension and for an enumeration of recognized pragmas.
The definitions of **__DATE__** and **__TIME__** when, respectively, the date and time of translation are not available (6.8.8)	These definitions are always available.

Library Functions (G.3.14)

Specific Area	Behavior
The null pointer constant to which the macro **NULL** expands (7.1.5)	The macro **NULL** expands to **0**.
The diagnostic printed by and the termination behavior of the **assert** function (7.2)	The **assert** function emits a diagnostic of the following form: **Assertion failed: expression, file filename, line linenumber** then exits via the **abort** function.
The sets of characters tested for by the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions (7.3.1)	The character set test functions operate on the ASCII character set. Specific ranges tested by each function are defined in Table 21.
The values returned by the mathematics functions on domain errors (7.5.1)	Refer to Table 22.
Whether the mathematics functions set the integer expression **errno** to the value of the macro **ERANGE** on underflow range errors (7.5.1)	Mathematics functions set **errno** to **ERANGE** on overflow or underflow range errors. Refer to Table 23.
Whether a domain error occurs or zero is returned when the **fmod** function has a second argument of zero (7.5.6.4)	A domain error occurs.
The set of signals for the **signal** function (7.7.1.1)	The **signal** function currently supports the signals specified in signal.h.
The semantics for each signal recognized by the **signal** function (7.7.1.1)	The **signal** function currently supports raising of signals only by **raise**.
The default handling and the handling at program startup for each signal recognized by the **signal** function (7.7.1.1)	**SIGABRT** prints an abort message and terminates program execution. Other signals are ignored by default.
If the equivalent of **signal(sig, SIG_DFL);** is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.7.1.1)	The signal is reset to the default prior to the signal handler being called.
Whether the default handling is reset if the **SIGILL** signal is received by a handler specified to the **signal** function (7.7.1.1)	The **signal** function does not restore the default handler when it receives **SIGILL** as a handler.
Whether the last line of a text stream requires a terminating new-line character (7.9.2)	The last line of a text stream does not need to end in a new line.
Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.9.2)	Space characters immediately preceding a new line appear when the stream is read in.

Specific Area	Behavior
The number of null characters that may be appended to data written to a binary stream (7.9.2)	The implementation does not append null characters to a binary stream.
Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (7.9.3)	The file position indicator is positioned at the end of the file.
Whether a write on a text stream causes the associated file to be truncated beyond that point (7.9.3)	A write on a text stream does not truncate the associated file beyond that point.
The characteristics of file buffering (7.9.3)	Output streams are, by default, block buffered if referring to a file and line buffered if referring to a terminal. The output stream **stderr** is, by default, unbuffered.
Whether a zero-length file actually exists (7.9.3)	Zero-length files do exist.
The rules for composing valid file names (7.9.3)	Valid files names are under 1,024 characters in length and can be composed of any characters excluding the null character and / (slash).
Whether the same file can be opened multiple times (7.9.3)	The same file can be opened multiple times.
The effect of the **remove** function on an open file (7.9.4.1)	The file is marked for deletion following the last close of the file. The file cannot be opened following a **remove** call.
The effect if a file with the new name exists prior to a call to the **rename** function (7.9.4.2)	The existing file is retained and the **rename** call fails.
The output for **%p** conversion in the **fprintf** function (7.9.6.1)	The output for **%p** is equivalent to **%x**.
The input for **%p** conversion in the **fscanf** function (7.9.6.2)	The input for **%p** is equivalent to **%x**.
The interpretation of a "-" character that is neither the first nor the last character in the scanlist for %[conversion in the **fscanf** function (4.9.6.2)	The "-" character, when neither the first nor last character in the scanlist, indicates an inclusive range.
The value to which the macro **errno** is set by the **fgetpos** or **ftell** functions on failure (7.9.9.1, 7.9.9.4)	On error, the function **fgetpos** or **ftell** sets the macro **errno** to **ESPIPE**.
The messages generated by the **perror** function (7.9.10.4)	Refer to Table 23 following.

Specific Area	Behavior
The behavior of the **calloc**, **malloc**, or **realloc** functions if the size requested is zero (7.10.3)	The function **calloc** or **malloc** will allocate and return a pointer to a zero-sized object. The function **realloc** resizes or frees a zero-sized object in the same fashion as any other size object.
The behavior of the **abort** function with regard to open and temporary files (7.10.4.1)	The **abort** function exits immediately, without closing open files or removing temporary files.
The status returned by the exit function if the value of the argument is other than zero, **EXIT_SUCCESS**, or **EXIT_FAILURE** (7.10.4.3)	The **exit** function returns its argument.
The set of environment names and the method for altering the environment list used by the **getenv** function (7.10.4.4)	The set of environment names are those that existed in the program environment at the time program execution was initiated.
The contents and mode of execution of the string by the **system** function (7.10.4.5)	The **system** function with an argument of **NULL** returns 0 to indicate that no command processor is available. The **system** function with a non-**NULL** argument currently never succeeds; it always returns **EXIT_FAILURE**.
The contents of the error message strings returned by the **strerror** function (7.11.6.2)	The **strerror** function returns the same error messages as the **perror** function.
The local time zone and Daylight Saving Time (7.12.1)	The local time zone is set by the environment variable **TZ**.
The **era** for the **clock** function (7.12.2.1)	The **era** originates at the time program execution was initiated.

Table 21 Character Set Test Function Ranges

Function	ASCII Character Set Range
isalnum	A–Z, a–z, and 0–9.
isalpha	A–Z and a–z.
iscntrl	Character values 0–31 and 127.
islower	a–z.
isprint	Character values 32 through 126.
isupper	A–Z.

Table 22 Math Function Domain Error Return Values

Function (Domain Error)	Return Value
acos(|x| > 1)	NaN
asin(|x| > 1)	NaN
atan2(±0, ±0)	NaN
log(x < 0)	NaN
log10(x < 0)	NaN
pow(0, y < 0)	NaN
pow(x < 0, y non-integral)	NaN
sqrt(x < 0)	NaN
fmod(x, y == 0)	NaN

Table 23 perror Error Messages

Number	Message
9	Bad file number.
12	Not enough memory.
22	Invalid argument.
29	Invalid seek.
33	Argument too large.
34	Result too large.

C++ Language Definition

Dialect

TriMedia-C++ has four modes that govern the C++ dialect accepted by the compiler, which is determined by the value of control-variable **c**:

- ARM mode—This is the strict ANSI mode of TriMedia-C++. It will track the standard as it develops. This mode initially accepted and implemented the language as described in *The Annotated C++ Reference Manual*, by Margaret A. Ellis and Bjarne Stroustrup (the ARM), which was the base document of the ANSI C++ standard currently being developed. This mode is invoked by the **-Xc=arm** option for **tm**CC and **tmcc**.
- CP mode—This is same as ARM but with several restrictions relaxed. This mode is the first-default-value for **tmCC** (.c and .C files) and for **tmcc** (.C files). This mode is invoked by the **-Xc=cp** option.

Version 4.0 of the C++ compiler is nearly current with the standardization process. It supports exceptions, RTTI (runtime type identification), templates, namespaces, and libraries including STL (the Standard Template Library). It also recognizes the keywords for **bool**, and **wchar_t**.

Several of the newer features are on by default, but can be selectively disabled by altering the value of control-variable **c**. All possible values are discussed in the reference above, but additional details are given in the next few paragraphs.

Boolean Type (bool)

Whether **bool** is recognized as a keyword or not is controlled by the presence or absence of the value **bool** in control-variable **c**. Beyond the obvious effect of preventing you from using **bool** for something besides the new type, there are additional ramifications of the change. The issue is similar to the first issue of **wchar_t** described in the next paragraph, and the names of the related predefined preprocessor macros are similar as well.

Wide Characters (wchar_t)

There are two fundamental issues related to **wchar_t**. The first issue is whether **wchar_t** is a built-in type distinct from any integer type (this is controlled by the presence or absence of the value **wchar_t** in control-variable **c**); the second issue is exactly what type is used to store variables declared to be of type (or typedef) **wchar_t** (this is controlled by control-variable **wchart**).

The first issue is more straightforward, but also has a larger impact in existing code. Recently the symbol **wchar_t** has been changed from a typedef of an integral type to a truly distinct, built-in type like **char**, **short**, or **float** always have been. This change in

behavior has two fundamental effects that are visible to existing code: trying to specify a typedef for **wchar_t** is now illegal, so any existing typedef declarations must be removed; and name mangling rules are different for **wchar_t** as a typedef than for **wchar_t** as a distinct type.

Before **wchar_t** became a type unto itself, an explicit typedef defining that symbol was required to be provided in one or more include files. In fact, the nature of the requirements usually led to it being defined identically in several include files, with each definition protected by a preprocessor macro to avoid making multiple definitions visible if multiple include files (each of which needed to define it) were included in one source file compilation. The preprocessor macro name varies on different target computer environments. Basically there was code in header files which looked something like.

```
#ifndef _WCHAR_T
typedef short /* or something */ wchar_t;
#define _WCHAR_T
#endif
```

This would ensure that **wchar_t** was defined once and only once. If you request compilation in a mode such that **wchar_t** is not a distinct built-in type, this behavior remains. If, however, you request compilation in a mode such that **wchar_t** is distinct (which is the default now), the compiler will pre-define **-D_WCHAR_T** (or whatever is appropriate) to eliminate all typedefs. If you have similar code in your sources, you will need to protect that typedef (or remove it entirely) in a similar manner.

The second half of this first issue is that name mangling rules require that the type **wchar_t** must be mangled differently from the typedef **wchar_t**. This has a minor effect if you have properly declared and used all of your functions and objects consistently. However, it has a significant impact in three areas: overloading of functions or operators, mangling of names, and instantiation of templates. For example, you may wish to overload functions for a variety of types.

```
void overloaded_func(          char  *a ) { ..... };
void overloaded_func( unsigned char  *a ) { ..... };
void overloaded_func(   signed char  *a ) { ..... };
void overloaded_func(          short *a ) { ..... };
void overloaded_func( unsigned short *a ) { ..... };
.....
#ifndef __WCHAR_T_IS_KEYWORD
void overloaded_func( wchar_t *a ) { ..... };
#endif
```

In this example, the final function must be supplied if **wchar_t** is a distinct type, and must be eliminated otherwise (to avoid a duplicate definition error, since **wchar_t** was actually equivalent to one of the other types). To support this, TriMedia-C++ will predefine a macro which is present if **wchar_t** is a keyword (and therefore a distinct type) and absent otherwise.

The different name mangling becomes visible if you try to link two modules that were compiled with different values of **wchar_t**. If **wchar_t** is a typedef, mangling is done based on the underlying type. If **wchar_t** is a type unto itself, mangling is done differ-

ently. This will prevent proper linking unless **wchar_t** is always (or never) a typedef across the entire program.

The second issue involving **wchar_t** is exactly what type is used. This issue exists in both C and C++, and it exists regardless of whether **wchar_t** is a type or a typedef. The compiler must have a built-in idea of what type is used for **wchar_t** because it is legal to use a wide-character string (for example, **L"abcd"**) before declaring the typedef.

In C, and in C++ when **wchar_t** is not a keyword, it is generally not a good idea to be changing the type backing **wchar_t** because the default value is set up to match the definitions in the include files. In C++ when **wchar_t** is a keyword, it is also generally not a good idea to be changing the type backing **wchar_t** because library routines have been compiled with the default value. If neither of these is an issue, or if you are supplying the include files and/or libraries, you may change the type by changing the value of control-variable **wchart**.

Special Pragmas

All pragma directives can be used in all four C++ modes. In addition three pragmas have been added to aid in control of template instantiations. The TriMedia-C++ compiler implements templates with a linker feedback mechanism.

```
#pragma instantiate argument
```

The above pragma causes the compiler to instantiate **argument** in this compilation. It can be used with the **-Xtmpl=none** mode to do only the required instantiations by explicitly specifying only the required ones.

```
#pragma do_not_instantiate argument
```

The above pragma causes the compiler to not instantiate **argument** in this compilation. It can be used with other values of the **tmpl** control-variable to suppress certain instantiations, because they are being done in some other source file.

```
#pragma can_instantiate argument
```

The above pragma tells the compiler that the **argument** may be instantiated in the current translation unit if needed.

In each case, **argument** may be a template class name, a member function name, a static data member name, a member function declaration, or a function declaration. When a class name is specified, the directive is applied to all member functions and static data members of the class.

Exception Handling

Exception handling has two major impacts on the compilation system:

1. Recognition of, and code generation for, explicit exception-handling constructs and keywords like **try**, **throw**, and **catch**.

2. Generation of code and/or tables in code that has no exception-handling constructs, but has local variables that need cleanup in case an exception is thrown across this code. This could happen if a stack frame for non-exception-related code is on the portion of the execution stack that is unwound as part of exception handling. The impact on the compiler is that all functions having local variables that require destruction must have their destructors called.

Exceptions controls only whether item 2 (preceding) is done. Item 1 (preceding) is done regardless of the setting. The implementation has a reduced cost where exceptions are not thrown, at the cost of some extra tables and some constant assignments. This cost is only borne in routines with destructors. The default is now 'on' in **cp** and **arm** modes. All code, by default, works properly in the presence of exceptions, with only minor negative impact.

An additional benefit is that you may now (with caution) turn exceptions off in portions of your code that you know are free from exceptions or where calling destructors is not important. This should, of course, be done with extreme care. As evidenced by our changing the default value to 'on', we recommend compiling *all* modules with exceptions enabled if you compile *any* modules that way.

If you want to eliminate completely the impact of exceptions on code that otherwise must be compiled in **arm** or **cp** modes, you can specify options such as **-Xc=arm-exceptions** or **-Xc-=exceptions** on the command line.

Ongoing Standardization Issues

As of SDE v2.0, the following features not in the ARM (but in the X3J16/WG21 Working Paper) are accepted:

- Standard library and include files.
- The dependent statement of an **if**, **while**, **do-while**, or **for** is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an **if**, **while**, **do-while**, or **for**, as the first operand of a **?:** operator, or as an operand of the **&&**, **||**, or **!** operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form **x.::A::B** and **p->::A::B**.
- The precedence of the third operand of the **?:** operator is changed.
- If control reaches the end of the **main()** routine, and **main()** has an integral return type, it is treated as if a **return 0;** statement were executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.

- A functional-notation cast of the form **A()** can be used even if **A** is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to **const volatile** cannot be bound to an **rvalue**.
- Qualification conversions such as conversion from **T**** to **T const * const *** are allowed.
- Digraphs are recognized.
- Operator keywords (for example, **and**, **bitand**, and so on) are recognized.
- Static data member declarations can be used to declare member constants.
- **wchar_t** is recognized as a keyword and a distinct type, by default.
- **bool** is recognized, by default.
- RTTI (runtime type identification), including **dynamic_cast** and the **typeid** operator is implemented, by default.
- Declarations in tested conditions (in **if**, **switch**, **for**, and **while** statements) are supported.
- Array **new** and **delete** are implemented, by default.
- New-style casts (**static_cast**, **reinterpret_cast**, and **const_cast**) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- **mutable** is accepted on nonstatic data member declarations.
- Namespaces are implemented, including **using** declarations and directives. Access declarations are broadened to match the corresponding **using** declarations.
- Explicit instantiation of templates is implemented.
- The **typename** keyword is recognized.
- **explicit** is accepted to declare non-converting constructors.
- The scope of a variable declared in the **for-init-statement** of a **for** loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using **template <>**) is implemented.
- cv-qualifiers are retained on **rvalues** (in particular, on function return values).

The following features not in the ARM (but in the X3J16/WG21 Working Paper) are not accepted:

- Virtual functions in derived classes may not return a type that is the derived-class version of the type returned by the overridden function in the base class.
- **enum** types are not considered to be non-integral types.
- It is not possible to overload operators using functions that take **enum** types and no class types.
- The new lookup rules for member references of the form **x.A::B** and p->**A::B** are not yet implemented.
- Classes are not assumed to always have constructors, and the distinction between trivial and nontrivial constructors is not implemented. (From a practical point of view, this has almost no effect.)
- **enum** types cannot contain values larger than can be contained in an **int**.
- **reinterpret_cast** does not allow casting a pointer to member of one class to a pointer to member of another class if the classes are unrelated.
- Explicit qualification of template functions, of the form **foo<int>(20)** is not implemented.
- Name binding in templates in the style of N0288/93-0081 is not implemented.
- In a reference of the form **f()->g()**, with **g** a static member function, **f()** is not evaluated. This is as required by the ARM. The Working Paper, however, requires that **f()** be evaluated.
- **(p->*pm) = 0** cannot yet be written as **p->*pm = 0** (the syntax still matches the ARM).
- Class name injection is not implemented.
- Overloading of function templates (partial specialization) is not implemented.
- Partial specialization of class templates is not implemented.
- Placement delete is not implemented.
- Putting a **try**/**catch** around the initializers and body of a constructor is not implemented.
- The notation **::template** (and **->template**, etc.) is not implemented.
- Template template parameters are not implemented.
- Certain restrictions are not yet enforced on the use of (pointer-to-) function types that involve exception-specifications.
- **extern inline** functions are not supported.

- Distinct name mangling for template function is not implemented. In Modern C++ a non-template function can not be used as an specialization for a template function.

```
template <class T> void foo(T a) { ..... template body  .... }
void foo(int x)            // say foo-1
{
   // Not a specialization of void(T) for T=int in modern C++
   // Used to be a specialization of void(T) for T=int in old C++.
}
template<> void foo(int x) // say foo-2
{
   // This is a specialization of void(T) for T=int.
}
// ...
foo(30);                   // calls foo-1
foo<int>(3.1)              // calls foo-2 with argument ((int)3.1), which
                           // equals 3. This is not accepted in release 4.0
```

In order to differentiate between these versions, the name mangling rules may have to be modified. That has not been done yet.

Other restrictions with the current C++ implementation

If an iterator template is defined with a -> operator method, then the compiler will issue an error if the type instantiated by the template cannot validly use the –> operation. (e.g., **int**). For this reason, our STL library does not include –> operators for the iterator templates. Please use

```
(*iterator).function()
```

instead of

```
iterator->function()
```

as a workaround.

Using Templates

This section describes several pragmas and command line options that are useful when working with template classes and functions.

Three situations occur when working with template classes:

1. The template class is used in one file only.
2. The template class is used in multiple compilation units (files) but all of its member functions are declared inline.
3. The template class is used in multiple compilation units and not all member functions can be inline.

The first two situations will pose no problem when trying to compile. However, the third can cause problems for the user.

To illustrate the point, assume a template list class, declared in list.h, with member functions in list.cc. In the first situation, list.h is included in the one file that uses the template list class; **tmCC** will automatically include list.cc upon parsing list.h to get all the

template definitions, and instantiate them if necessary. Thus, list.cc does not have to be compiled separately and linked with the other files!

In the second situation, all the member functions in list.cc are declared inline. **tmCC** automatically includes list.cc in every file that includes list.h, but because all member functions are inline, there will be no 'multiply defined' symbols, and the compilation will proceed as expected. This is largely how the STL is implemented.

The third situation—the tricky one—uses options and pragmas to ensure successful compilation and linking. In this situation, list.cc should contain initializations of static variables and all member functions that are not declared inline. All inline member functions should be included in list.h. First, you must prevent the automatic inclusion of list.cc whenever list.h is included in a file, because this will lead to 'multiply defined' symbols at link time. You can do this two ways: (a) rename list.cc (for instance, to lst.cc), which is the quick-and-dirty way, or (b) use the command line option **-Xtmpl=none+noautoincl** on all files. Second, recompile the list.cc file separately, with the following pragma in the file for each different template use of a class:

#pragma instantiate *name*

where *name* is the name of the instantiated template class (**List<int>**) in our example. This will correct instantiate everything used by **List<int>** too, e.g. **Link<int>**. When all the files are linked together, you should experience no linking errors. There will be only one function template instantiation.

Using <iostream> and <string>

As of release 2.1 of TriMedia C++, certain common template inline functions in <iostream> and <string> have moved to the library libC++.a. This gives an enormous improvement in compilation time for programs that use <iostream>, while only minimally affecting performance.

This move has a slight impact for users that use streams or strings with types other than **char**. For these users, there are two compile-time defines that will allow full template instantiation in the STL. If you need wider strings, use **TCS_FULL_STRING**. If you need wider streams, use **TCS_FULL_IOSTREAM**. These defines can be specified on the **tmcc** command line as **-DTCS_FULL_IOSTREAM** or **-DTCS_FULL_STRING**.

Implementation Specifics

The following points address specific characteristics of the implementation.

Error Message Compiling with -p

The program must be compiled with the same options when producing a dtprof.out file (**-p** option) and for the second compilation run. In particular, the same optimization level will result in both cases.

The dtprof.out file does not contain the necessary information.

Correct behavior when compiling with different optimization levels for the two runs is not guaranteed.

Variable Addressing

The compiler can generate code using either based addressing (from a register) or absolute addressing for each individual variable. The default is absolute addressing. Make the choice by passing **-generate data_units** to the compiler.

When compiling with the **-t** or **-S** options, the default is changed. To examine code generated by the compiler, use the **-tmccom -generate_data_units** –– option.

Compiler Messages

The warning level controls that were available in previous versions of the compiler are no longer available. Some source code that compiled without warnings using the 1.1 compiler's default settings will now have warnings.

Messages produced by the compiler are subject to change in future versions.

Performance Impact of -compact Option

Cache performance can be affected by code reordering caused by the **-compact** option. This can be detected using **tmprof**.

To disable code reordering, use the **tmcc -nocompact** option.

Run-Time Exit Code

In order for the run-time system to work, the main function needs to return a value. If no value is returned, or if **main** is declared of **void** type, an undetermined return value is returned to the host environment and correct operation is not guaranteed.

Chapter 9

Library Functions

Introduction

The TriMedia Compilation System supports the standard C library as defined in *American National Standard for Programming Languages—C*, ANSI/ISO 9899-1990, commonly known as the ANSI C Standard. Section 7 of the Standard specifies standard header files (cf. 7.1.2), which define macros, types, functions, and globals.

Headers

Table 24 lists the 15 header files required by the Standard, together with the section number of the Standard that describes the contents of each header and its subject.

Table 24 Headers

Header	Section	Subject
assert.h	7.2	Diagnostics.
ctype.h	7.3	Character handling.
errno.h	7.1.4	Errors.
float.h	7.1.5	Limits (characteristics of floating types).
limits.h	7.1.5	Limits (sizes of integral types).
locale.h	7.4	Localization.
math.h	7.5	Mathematics.
setjmp.h	7.6	Nonlocal jumps.
signal.h	7.7	Signal handling.
stdarg.h	7.8	Variable arguments.
stddef.h	7.1.6	Common definitions.
stdio.h	7.9	Input/output.
stdlib.h	7.10	General utilities.
string.h	7.11	String handling.
time.h	7.12	Date and time.

Macros

Table 25 lists the 92 macros required by the Standard, the name of the header that defines each macro, and a brief description of the macro. A few macros are defined in more than one header. See the Standard for a detailed description of each macro. See the headers in the TCS distribution for the TCS implementation of each macro.

Note
TCS defines **va_end** as a macro while the Standard allows it to be implemented either as a macro or as a function.

Table 25 Macros

Header	Macro	Description
stdio.h	_IOFBF	Fully buffered flag.
stdio.h	_IOLBF	Line buffered flag.
stdio.h	_IONBF	Unbuffered flag.
stdio.h	BUFSIZ	Input/output buffer size.
limits.h	CHAR_BIT	Bits per character.
limits.h	CHAR_MAX	Maximum char value.
limits.h	CHAR_MIN	Minimum char value.
time.h	CLOCKS_PER_SEC	Clocks per second.
float.h	DBL_DIG	Decimal digits in double significand.
float.h	DBL_EPSILON	Smallest double greater than 1, minus 1.
float.h	DBL_MANT_DIG	Number of digits in double significand.
float.h	DBL_MAX	Maximum representable double.
float.h	DBL_MAX_10_EXP	Maximum double decimal exponent.
float.h	DBL_MAX_EXP	Maximum double exponent.
float.h	DBL_MIN	Minimum representable double.
float.h	DBL_MIN_10_EXP	Minimum double decimal exponent.
float.h	DBL_MIN_EXP	Minimum double exponent.
errno.h	EDOM	Domain error.
stdio.h	EOF	End of file indicator.
errno.h	ERANGE	Range error.
stdlib.h	EXIT_FAILURE	Program failure exit status.
stdlib.h	EXIT_SUCCESS	Program success exit status.
stdio.h	FILENAME_MAX	Longest file name length.

Table 25 Macros

Header	Macro	Description
float.h	FLT_DIG	Decimal digits in float significand.
float.h	FLT_EPSILON	Smallest float greater than 1, minus 1.
float.h	FLT_MANT_DIG	Number of digits in float significand.
float.h	FLT_MAX	Maximum float value.
float.h	FLT_MAX_10_EXP	Maximum float decimal exponent.
float.h	FLT_MAX_EXP	Maximum float exponent.
float.h	FLT_MIN	Minimum float value.
float.h	FLT_MIN_10_EXP	Minimum float decimal exponent.
float.h	FLT_MIN_EXP	Minimum float exponent.
float.h	FLT_RADIX	Radix of floating point exponent representation.
float.h	FLT_ROUNDS	Floating point rounding mode.
stdio.h	FOPEN_MAX	Maximum number of simultaneously open streams.
math.h	HUGE_VAL	Numeric overflow value.
limits.h	INT_MAX	Maximum int value.
limits.h	INT_MIN	Mimimum int value.
locale.h	LC_ALL	All locale functions.
locale.h	LC_COLLATE	Locale collating functions.
locale.h	LC_CTYPE	Locale character handling functions.
locale.h	LC_MONETARY	Locale monetary formatting functions.
locale.h	LC_NUMERIC	Locale numeric formatting functions.
locale.h	LC_TIME	Locale time formatting functions.
float.h	LDBL_DIG	Decimal digits in long double significand.
float.h	LDBL_EPSILON	Smallest long double greater than 1, minus 1.
float.h	LDBL_MANT_DIG	Number of digits in long double significand.
float.h	LDBL_MAX	Maximum long double value.
float.h	LDBL_MAX_10_EXP	Maximum long double decimal exponent.
float.h	LDBL_MAX_EXP	Maximum long double exponent.
float.h	LDBL_MIN	Minimum long double value.

Table 25 Macros

Header	Macro	Description
float.h	LDBL_MIN_10_EXP	Minimum long double decimal exponent.
float.h	LDBL_MIN_EXP	Minimum long double exponent.
limits.h	LONG_MAX	Maximum long value.
limits.h	LONG_MIN	Minimum long value.
stdio.h	L_tmpnam	Temporary file name length.
stdlib.h	MB_CUR_MAX	Largest size of a multibyte character in current locale.
limits.h	MB_LEN_MAX	Maximum number of bytes in multibyte character.
assert.h	NDEBUG	Suppress assertion processing.
locale.h, stddef.h, stdio.h, stdlib.h, string.h, time.h	NULL	Null pointer constant.
stdlib.h	RAND_MAX	Largest size of a pseudo-random number.
limits.h	SCHAR_MAX	Maximum signed char value.
limits.h	SCHAR_MIN	Minimum signed char value.
stdio.h	SEEK_CUR	Seek from current position.
stdio.h	SEEK_END	Seek from end of file.
stdio.h	SEEK_SET	Seek from beginning of file.
limits.h	SHRT_MAX	Maximum short value.
limits.h	SHRT_MIN	Minimum short value.
signal.h	SIGABRT	Abnormal termination signal.
signal.h	SIGFPE	Arithmetic exception signal.
signal.h	SIGILL	Illegal instruction signal.
signal.h	SIGINT	Interactive attention signal.
signal.h	SIGSEGV	Invalid memory access signal.
signal.h	SIGTERM	Termination request signal.
signal.h	SIG_DFL	Default signal handler.
signal.h	SIG_ERR	Signal handler error return value.

Table 25 Macros

Header	Macro	Description
signal.h	**SIG_IGN**	Signal handler to ignore signal.
stdio.h	**TMP_MAX**	Minimum number of temporary file names.
limits.h	**UCHAR_MAX**	Maximum unsigned char value.
limits.h	**UINT_MAX**	Maximum unsigned int value.
limits.h	**ULONG_MAX**	Maximum unsigned long value.
limits.h	**USHRT_MAX**	Maximum unsigned short value.
assert.h	**assert(p)**	Check assertion at run time.
errno.h	**errno**	Error condition indicator.
stddef.h	**offsetof(type, member)**	Offset of member in type.
setjmp.h	**setjmp(jmp_buf env)**	Save state for non-local goto.
stdio.h	**stderr**	Standard error stream.
stdio.h	**stdin**	Standard input stream.
stdio.h	**stdout**	Standard output stream.
stdarg.h	**va_arg(va_list ap, type)**	Get next argument from argument list.
stdarg.h	**va_end(va_list ap)**	Clean up after traversal of argument list.
stdarg.h	**va_start(va_list ap, parmN)**	Begin traversal of argument list.

Functions

Table 26 lists the prototypes of the 141 functions required by the Standard, the name of the header that declares each function, and a brief description of the function's purpose. See the Standard for a detailed description of each function.

Table 26 Functions

Header	Type	Function	Description
<stdlib.h>	void	abort(void)	End program immediately.
<stdlib.h>	int	abs(int j)	Return the absolute value of an integer.
<math.h>	double	acos(double x)	Compute inverse cosine.
<time.h>	char *	asctime(const struct tm *timeptr)	Convert time structure to string.
<math.h>	double	asin(double x)	Compute inverse sine.
<math.h>	double	atan(double x)	Compute inverse tangent.
<math.h>	double	atan2(double y, double x)	Compute inverse tangent.
<stdlib.h>	int	atexit(void (*func)(void))	Register function to be called when the program exits.
<stdlib.h>	double	atof(const char *nptr)	Convert string to floating point.
<stdlib.h>	int	atoi(const char *nptr)	Convert string to integer.
<stdlib.h>	long	atol(const char *nptr)	Convert string to long integer.
<stdlib.h>	void *	bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar) (const void *, const void *))	Search an array.
<stdlib.h>	void *	calloc(size_t nmemb, size_t size)	Allocate dynamic memory.
<math.h>	double	ceil(double x)	Compute numeric ceiling.
<stdio.h>	void	clearerr(FILE *stream)	Reset stream error status.
<time.h>	clock_t	clock(void)	Get processor time.
<math.h>	double	cos(double x)	Compute cosine.
<math.h>	double	cosh(double x)	Compute hyperbolic cosine.
<time.h>	char *	ctime(const time_t *timer)	Convert system time to string.

Table 26 Functions

Header	Type	Function	Description
<time.h>	double	difftime(time_t t1, time_t t0)	Compute difference between two times.
<stdlib.h>	div_t	div(int numer, int denom)	Perform integer division.
<stdlib.h>	void	exit(int status)	Terminate program gracefully.
<math.h>	double	exp(double x)	Compute exponent.
<math.h>	double	fabs(double x);	Compute absolute value.
<stdio.h>	int	fclose(FILE *stream)	Close stream.
<stdio.h>	int	feof(FILE *stream)	Discover stream end of file status.
<stdio.h>	int	ferror(FILE *stream)	Discover stream status.
<stdio.h>	int	fflush(FILE *stream)	Flush output stream's buffer.
<stdio.h>	int	fgetc(FILE *stream)	Read character from stream.
<stdio.h>	int	fgetpos(FILE *stream, fpos_t *pos)	Get value of file-position indicator.
<stdio.h>	char *	fgets(char *s, int n, FILE *stream)	Read line from stream.
<math.h>	double	floor(double x)	Compute numeric floor.
<math.h>	double	fmod(double x, double y)	Compute floating-point modulus.
<stdio.h>	FILE *	fopen(const char *filename, const char *mode)	Open stream for standard I/O.
<stdio.h>	int	fprintf(FILE *stream, const char *format, ...)	Print formatted output into stream.
<stdio.h>	int	fputc(int c, FILE *stream)	Write character into stream.
<stdio.h>	int	fputs(const char *s, FILE *stream)	Write string into stream.
<stdio.h>	size_t	fread(void *ptr, size_t size, size_t nmemb, FILE *stream)	Read data from stream.
<stdlib.h>	void	free(void *ptr)	Free dynamic memory.
<stdio.h>	FILE *	freopen(const char *filename, const char *mode, FILE *stream)	Reopen stream for standard I/O.
<math.h>	double	frexp(double value, int *exp)	Separate fraction and exponent.
<stdio.h>	int	fscanf(FILE *stream, const char *format, ...)	Format input from stream.

Table 26 Functions

Header	Type	Function	Description
<stdio.h>	int	fseek(FILE *stream, long int offset, int whence)	Seek on stream.
<stdio.h>	int	fsetpos(FILE *stream, const fpos_t *pos)	Set file-position indicator.
<stdio.h>	long	ftell(FILE *stream)	Return current position of file pointer.
<stdio.h>	size_t	fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)	Write to stream.
<stdio.h>	int	getc(FILE *stream)	Read character from stream.
<stdio.h>	int	getchar(void)	Read character from standard input.
<stdlib.h>	char *	getenv(const char *name)	Read environmental variable.
<stdio.h>	char *	gets(char *s)	Read string from standard input.
<time.h>	struct tm *	gmtime(const time_t *timer)	Convert system time to calendar structure.
<ctype.h>	int	isalnum(int c)	Check if character is a number or a letter.
<ctype.h>	int	isalpha(int c)	Check if character is a letter.
<ctype.h>	int	iscntrl(int c)	Check if character is a control character.
<ctype.h>	int	isdigit(int c)	Check if character is a numeral.
<ctype.h>	int	isgraph(int c)	Check if character is printable.
<ctype.h>	int	islower(int c)	Check if character is a lower-case letter.
<ctype.h>	int	isprint(int c)	Check if character is printable.
<ctype.h>	int	ispunct(int c)	Check if character is a punctuation mark.
<ctype.h>	int	isspace(int c)	Check if character prints white space.
<ctype.h>	int	isupper(int c)	Check if character is an uppercase letter.

Table 26 Functions

Header	Type	Function	Description
<ctype.h>	int	isxdigit(int c)	Check if character is a hexadecimal numeral.
<stdlib.h>	long	labs(long j)	Return the absolute value of long integer.
<math.h>	double	ldexp(double x, int exp)	Combine fraction and exponent.
<stdlib.h>	ldiv_t	ldiv(long numer, long denom)	Perform long integer division.
<locale.h>	struct lconv *	localeconv(void)	Set members of current locale.
<time.h>	struct tm *	localtime(const time_t *time)	Convert system time to calendar structure.
<math.h>	double	log(double x)	Compute natural logarithm.
<math.h>	double	log10(double x)	Compute common logarithm.
<setjmp.h>	void	longjmp(jmp_buf env, int val)	Perform non-local goto.
<stdlib.h>	void *	malloc(size_t size)	Allocate dynamic memory.
<stdlib.h>	int	mblen(const char *s, size_t n)	Compute multibyte character length.
<stdlib.h>	size_t	mbstowcs(wchar_t *pwcs, const char *s, size_t n)	Convert multibyte character string to wide character string.
<stdlib.h>	int	mbtowc(wchar_t *pwc, const char *s, size_t n)	Convert multibyte character to wide character.
<string.h>	void *	memchr(const void *s, int c, size_t n)	Search region of memory for character.
<string.h>	int	memcmp(const void *s1, const void *s2, size_t n)	Compare two regions.
<string.h>	void *	memcpy(void *s1, const void *s2, size_t n)	Copy one region of memory into another.
<string.h>	void *	memmove(void *s1, const void *s2, size_t n)	Copy region of memory into area it overlaps.
<string.h>	void *	memset(void *s, int c, size_t n)	Fill an area with character.
<time.h>	time_t	mktime(struct tm *timeptr)	Convert broken-down time into calendar time.

Table 26 Functions

Header	Type	Function	Description
<math.h>	double	modf(double value, double *iptr)	Separate integral part and fraction.
<stdio.h>	void	perror(const char *s)	System call error messages.
<math.h>	double	pow(double x, double y)	Compute power of a number.
<stdio.h>	int	printf(const char *format, ...)	Print formatted text.
<stdio.h>	int	putc(int c, FILE *stream)	Write character into stream.
<stdio.h>	int	putchar(int c)	Write character onto the standard output.
<stdio.h>	int	puts(const char *s)	Write string onto standard output.
<stdlib.h>	void	qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))	Sort arrays in memory.
<signal.h>	int	raise(int sig)	Send signal.
<stdlib.h>	int	rand(void)	Generate pseudo-random numbers.
<stdlib.h>	void *	realloc(void *ptr, size_t size)	Reallocate dynamic memory.
<stdio.h>	int	remove(const char *filename)	Remove file.
<stdio.h>	int	rename(const char *old, const char *new)	Rename file.
<stdio.h>	void	rewind(FILE *stream)	Reset file pointer.
<stdio.h>	int	scanf(const char *format, ...)	Accept and format input.
<stdio.h>	void	setbuf(FILE *stream, char *buf)	Set alternative stream buffer.
<setjmp.h>	int	setjmp(jmp_buf env)	Save machine state for non-local goto.
<locale.h>	char *	setlocale(int category, const char *locale)	Set current locale.
<stdio.h>	int	setvbuf(FILE *stream, char *buf, int mode, size_t size)	Set alternative file-stream buffer.
<signal.h>	void	(*signal(int sig, void (*func)(int))) (int)	Specify action to take upon receipt of a given signal.
<math.h>	double	sin(double x)	Compute sine.

Table 26 Functions

Header	Type	Function	Description
<math.h>	double	sinh(double x)	Compute hyperbolic sine.
<stdio.h>	int	sprintf(char *s, const char *format, ...)	Format output.
<math.h>	double	sqrt(double x)	Compute square root.
<stdlib.h>	void	srand(unsigned int seed)	Seed random number generator.
<stdio.h>	int	sscanf(char *s, const char *format, ...)	Format string.
<string.h>	char *	strcat(char *s1, const char *s2)	Concatenate two strings.
<string.h>	char *	strchr(const char *s, int c)	Find character in a string.
<string.h>	int	strcmp(const char *s1, const char *s2)	Compare two strings.
<string.h>	int	strcoll(const char *s1, const char *s2)	Compare two strings, using locale-specific information.
<string.h>	char *	strcpy(char *s1, const char *s2)	Copy one string into another.
<string.h>	size_t	strcspn(const char *s1, const char *s2)	Return length a string excludes characters in another.
<string.h>	char *	strerror(int errnum)	Translate an error number into a string.
<time.h>	size_t	strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr)	Format locale-specific time.
<string.h>	size_t	strlen(const char *s)	Find length of a string.
<string.h>	char *	strncat(char *s1, const char *s2, size_t n)	Append one string onto another.
<string.h>	int	strncmp(const char *s1, const char *s2, size_t n)	Compare two strings.
<string.h>	char *	strncpy(char *s1, const char *s2, size_t n)	Copy one string into another.
<string.h>	char *	strpbrk(const char *s1, const char *s2)	Find first occurrence of character from another string.
<string.h>	char *	strrchr(const char *s, int c)	Search for rightmost occurrence of a character in a string.

Table 26 Functions

Header	Type	Function	Description
<string.h>	size_t	strspn(const char *s1, const char *s2)	Return length a string includes characters in another.
<string.h>	char *	strstr(const char *s1, const char *s2)	Find one string within another.
<stdlib.h>	double	strtod(const char *nptr, char **endptr)	Convert string to floating-point number.
<string.h>	char *	strtok(char *s1, const char *s2)	Break a string into tokens.
<stdlib.h>	long	strtol(const char *nptr, char **endptr, int base)	Convert string to long integer.
<stdlib.h>	unsigned long	strtoul(const char *nptr, char **endptr, int base)	Convert string to unsigned long integer.
<string.h>	size_t	strxfrm(char *s1, const char *s2, size_t n)	Transform a string.
<stdlib.h>	int	system(const char *string)	Pass a command to the shell for execution.
<math.h>	double	tan(double x)	Compute tangent.
<math.h>	double	tanh(double x)	Compute hyperbolic cosine.
<time.h>	time_t	time(time_t *time)	Get current system time.
<stdio.h>	FILE *	tmpfile(void)	Create a temporary file.
<stdio.h>	char *	tmpnam(char *s)	Generate a unique name for a temporary file.
<ctype.h>	int	tolower(int c);	Convert characters to lower-case.
<ctype.h>	int	toupper(int c);	Convert characters to upper-case.
<stdio.h>	int	ungetc(int c, FILE *stream)	Return character to input stream.
<stdio.h>	int	vfprintf(FILE *stream, const char *format, va_list arg)	Print formatted text into stream.
<stdio.h>	int	vprintf(const char *format, va_list arg)	Print formatted text into standard output stream.

Table 26 Functions

Header	Type	Function	Description
<stdio.h>	int	vsprintf(char *s, const char *format, va_list arg)	Print formatted text into string.
<stdlib.h>	size_t	wcstombs(char *s, const wchar_t *pwcs, size_t n)	Convert wide character string to multibyte character string.
<stdlib.h>	int	wctomb(char *s, wchar_t wchar)	Convert wide character to multibyte character.

Long Double Library Functions

In addition to the standard library functions listed above, TCS provides the following equivalent long double functions for use with the **-uselongdouble64** flag. These functions provide greater precision than their double counterparts, at the expense of slower execution.

Header	Function	Description
<math.h>	_ld_acos(long double)	Compute inverse cosine
<math.h>	_ld_asin(long double)	Compute inverse sine
<math.h>	_ld_atan(long double)	Compute inverse tangent
<math.h>	_ld_atan2(long double, long double)	Compute inverse tangent
<math.h>	_ld_ceil(long double)	Compute numeric ceiling
<math.h>	_ld_cos(long double)	Compute cosine
<math.h>	_ld_cosh(long double)	Compute hyperbolic cosine
<math.h>	_ld_exp(long double)	Compute exponent
<math.h>	_ld_fabs(long double)	Compute absolute value
<math.h>	_ld_floor(long double)	Compute numeric floor
<math.h>	_ld_fmod (long double, long double)	Compute floating-point modulus
<math.h>	_ld_frexp(long double, int *)	Separate fraction and exponent
<math.h>	_ld_ldexp(long double, int)	Combine fraction and exponent
<math.h>	_ld_log(long double)	Compute natural logarithm
<math.h>	_ld_log10(long double)	Compute common logarithm
<math.h>	_ld_modf(long double, long double*)	Separate integral part and fraction
<math.h>	_ld_pow(long double, long double)	Compute power of a number
<math.h>	_ld_sin (long double)	Compute sine
<math.h>	_ld_sinh(long double)	Compute hyperbolic sine
<math.h>	_ld_sqrt(long double)	Compute square root
<math.h>	_ld_tan(long double)	Compute tangent
<math.h>	_ld_tanh(long double)	Compute hyperbolic cosine
<stdlib.h>	_ld_atof(const char *)	Convert string to floating point number
<stdlib.h>	_ld_strtod(const char *, char **)	Convert string to floating-point number

Types

Table 27 lists the 14 types required by the Standard, together with the name of the header that defines each type. A few types are defined in more than one header. See the Standard for a detailed description of each type. See the headers in the TCS distribution for the TCS implementation of each type.

Table 27 Types

Header	Type	Description
time.h	clock_t	Clock tick.
stdlib.h	div_t	Result type for div().
stdio.h	FILE	Input/output stream.
stdio.h	fpos_t	File position indicator.
setjmp.h	jmp_buf	Nonlocal goto calling environment.
locale.h	struct lconv	Locale-specific numeric conversion.
stdlib.h	ldiv_t	Result type for ldiv().
stddef.h	ptrdiff_t	Pointer subtraction result type.
signal.h	sig_atomic_t	Integral atomic object.
stddef.h, stdio.h, stdlib.h, string.h, time.h	size_t	Result type for sizeof().
time.h	time_t	Time.
time.h	struct tm	Calendar time components.
stdarg.h	va_list	Variable argument list.
stddef.h, stdlib.h	wchar_t	Wide character.

System Calls

In addition to functions that are part of the standard C library defined in ANSI/ISO 9899-1990, the TriMedia compilation system library includes a number of other functions, including system calls and functions that are TriMedia-specific. Programs that use these functions are not necessarily portable to other C implementations.

The TCS library includes implementations of several system calls. Most of these are defined in the POSIX.1 standard. Table 28 lists these system calls, together with the header that declares the system call and a brief description of its purpose. For more detailed descriptions, see the POSIX.1 standard. Programs which use POSIX system calls are portable across POSIX-compliant systems.

Table 28 System Calls

Header	Type	Function	Purpose
unistd.h	int	access(const char *path, int amode)	Check file access.
unistd.h	int	close(int fildes)	Close file.
dirent.h	int	closedir(DIR *)	Close directory.
unistd.h	void	_exit(int status)	Terminate a program.
fcntl.h	int	fcntl(int fildes, int cmd, ...)	File control.
sys/stat.h	int	fstat(int fd, struct stat *sbuf)	Find file attributes.
unistd.h	int	fsync(int fildes)	Flush I/O buffers (non-POSIX).
unistd.h	int	isatty(int fildes)	Check if file is a terminal.
unistd.h	int	link(const char *existing, const char *newfile)	Create a link.
unistd.h	off_t	lseek(int fildes, off_t offset, int whence)	Set read/write position.
unistd.h	unsigned int	microsleep(unsigned int microseconds)	Sleep for microseconds (non-POSIX).
unistd.h	int	mkdir(char *path, int mode)	Create directory.
unistd.h	char	*mktemp(char *temp)	Generate a temporary file name (non-POSIX).
fcntl.h	int	open(const char *path, int oflag, ...)	Open file.
dirent.h	DIR *	opendir(const char *)	Open directory.
unistd.h	int	putenv(char *string)	Put string into environment (non-POSIX).
unistd.h	ssize_t	read(int fildes, void *buf, size_t nbyte)	Read from file.

Table 28 System Calls (Continued)

Header	Type	Function	Purpose
dirent.h	struct dirent	*readdir(DIR *)	Read directory.
dirent.h	void	rewinddir(DIR *)	Rewind directory.
unistd.h	int	rmdir(char *path)	Remove directory.
unistd.h	void	*sbrk(int incr)	Increase heap size (non-POSIX).
unistd.h	unsigned int	sleep(unsigned int seconds)	Sleep for seconds.
sys/stat.h	int	stat(const char *path, struct stat *sbuf)	Find file attributes.
unistd.h	int	sync(void)	Flush I/O buffers (non-POSIX).
unistd.	int	unlink(const char *path)	Remove file.
unistd.h	ssize_t	write(int fildes, const void *buf, size_t nbyte)	Write to file.

