Book 4—Software Tools

Part B:

Program Development Tools

Version 2.1

Table of Contents

Chapter 10 TriMedia Assembler

Introduction ... **10**

The Assembler ... **11**
- Assembly Code Checks ... 11
- Main Options ... 11
- The -handcode option ... 11

Assembly Syntax ... **12**
- Example ... 12
- Machine Constraints ... 13
- Other Constraints ... 13
  - Pseudo-Operations ... 13
  - Control-Flow Fall-Through ... 14
  - Branches are Mutually Exclusive ... 14
- Assembly Expressions ... 14
- Expression Syntax ... 14
- Use of Symbols in Assembly Expressions ... 15
- Assembler Directives ... 15
- Assembler Directives at a Glance ... 16
- .text, .data, .data1: Switch Sections ... 16
- .ascii: Character Data ... 17
- .byte, .half, .word: Generating Data ... 17
- .common: Declare a Common Symbol ... 18
- .reserve: Define a Symbol with Alignment ... 18
- .global: Define a Global Symbol ... 19
- .align: Align the Current Address ... 19
- Use of Directives and Program Layout ... 19

TM-1*x*00 Constraints ... **20**
- Instruction Format and TM-1*x*00 Constraints ... 21
  - Instruction Format ... 21
  - Code Placement Constraints ... 22
- Special Register Semantics ... 25
  - PCSW Writes and Reads ... 25
  - Changes to DPC ... 26

MMIO Location Updates 27
Forward Compatibility 28
Crossover of Register Writes 28
Assembly Program Checklist 29

Interfacing C with Assembly Language Programs 30
Memory Layout and C Calling Conventions 30
Using the C Preprocessor 30
Calling Assembly Code from C Code 30
C Variables and Corresponding Assembly Directives 39

Opcodes 41
Functional Unit Types 42
ALU 42
BRANCH 42
CONST 42
DMEM 43
DMEMSPEC 43
DSPALU 43
DSPMUL 43
FALU 43
FCOMP 44
FTOUGH 44
IFMUL 44
SHIFTER 44
TriMedia Opcodes 44

Chapter 11 Linking TriMedia Object Modules

Introduction 68

Overview 68
Object Files 68
Object File Structure 69
Object Manipulation Tools 73

Object File Contents 73
Sections 74
Example 75
Program Unit Attributes 76
Section Attributes 77
System Sections and Sections Introduced by tmld 78
Symbols 78
References 80

 ©1999 Philips Semiconductors 10/08/99

Static Linking **80**

Dynamic Linking **82**

Why Dynamic Linking is Valuable 82
Concepts of Dynamic Loading 83
Difference Between Static- and Dynamic Linking 83
Code Segments 84
Simple Examples 87
Dynamic Library 87
Runtime Library Update 88
Application Shell 90
Dynamic Loader Shell 91
Responsibilities of the Dynamic Loader 93
More on Dynamic Libraries 95
Dynamic Library Roles 95
Dynamic Library Search Path 95
Exported Symbols 96
Control Over Implicit Dynamic Loading 98
Function Stubs 101
Compatibility Across Versions of Dynamic Libraries 102
Binding Code Segments 103
TriMedia Dynamic Loader Architecture 106
Notes and Caveats of Dynamic Loading 107
Carefully Consider Transitive Errors 107
Real Time Issue 107
Carefully Consider Exporting Internal State 107
Function Stub is Part of Referring Segment 108
Executables are Generally Larger 108
Code Segments and PIC 109
Compiler Options for Dynamic Loading 109
More Examples 110
Dynamic Loading from Flash 110
Memory Manager Customization 111
Implicit Loading Error Handling 112

Section Renaming **113**

Sections Produced by tmccom 114
Other Sections Produced by SDE Tools 115

Link Optimizations **116**

Multiprocessor Support **117**

Shared Memory 118

SDRAM Memory Images vs Load Images 119

Constructing Load Images Using tmld 122

Download Symbols 123
Reserved Download Symbols 123
Other Download Symbols Used by the TriMedia SDE 125

tmld Options 126

List Construction by tmld 130
Example 131
Other Uses of Chain Symbols 132
Reserved Chain Symbols 132

Chapter 12 TriMedia Execution Host Utilities

Summary 136
The TriMedia Manager 136
TMMan Setup & Removal 136

TriMedia Commands 137

tmgmon 138
Running the Software 138
Processor 138
Code Download 139
Memory 139
Trace 140

TMRun 140

TMmpRun 141

Chapter 13 The TriMedia Simulator

Introduction to Machine-Level Simulation of TriMedia Processors 144

The Simulated Architecture 146

Command Syntax 149
Command Line Options 150
Interactive debug commands 154
Setup Commands 154
Simulation and Debugging Commands 155
Information-Printing Commands 156
Batch Mode and Source Files 158
Trace Mode and Performance Statistics 159

 ©1999 Philips Semiconductors 10/08/99

Interrupt and Exception Handlers 160
Data Layout 161
Program Debugging 161
Operating System Emulation 162
Performance Analysis Support: The tmsim Statfile 162
Video-Out Support 163

Chapter 14 Using the TriMedia CodeWarrior Plugins

Overview 166
Installing the TriMedia CodeWarrior Plugins 166
Win95/98/NT 166
MacOS 166
Known Problems 166
Implementation Notes 167
Speeding Up Compilation 167
Browser Catalog 167
File Names and Search Paths 168
Specifying TriMedia CodeWarrior Settings 168
Target Settings 169
Access Paths 169
File Mappings 170
TriMedia Target 171
C Language 174
TriMedia Assembler 175
TriMedia Compiler 176
TriMedia Scheduler 178
TriMedia Linker 180

 © 1999 Philips Semiconductors 10/08/99

Chapter 10

TriMedia Assembler

Topic	Page
Introduction	10
The Assembler	11
Assembly Syntax	12
TM-1x00 Constraints	20
Interfacing C with Assembly Language Programs	30
Opcodes	41

Introduction

This chapter describes the TriMedia assembler **tmas**. It consists of five main sections, which contain in this order:

1. An overview of the options to the Assembler.
2. A description of the syntax of the TriMedia Assembly Language.
3. The constraints on the assembly as imposed by the TM-1X00 processors.
4. A guide to interfacing Assembly with C.
5. A table of available opcodes.

The TriMedia Assembly Language has been designed as an intermediate language between the instruction scheduler and the assembler, and was never meant to be used as a programming language. Due to the architecture, there are many constraints on the type of instruction that can be used, and the order in which they can be used. Using the assembler to write programs is therefore not recommended. This chapter is meant as a guide for those people who need that extra cycle that assembly programming can bring.

 ©1999 Philips Semiconductors 10/08/99

The Assembler

Assembly Code Checks

The assembler reads in the assembly file (typically a **.s** file) and writes an object file (typically a **.o** file). Several checks are run on the source file to determine possible run-time conflicts. Note that not all run-time conflicts can be detected at compile time. The user should therefore be very careful with writes and branches. Currently, the following checks are performed:

- Correct issue slot. The assembler checks whether all operations are in slots that support the operation.
- Maximum five results per cycle. The assembler tries to check whether only five result-producing operations complete at each cycle. It is not always possible to check this in the presence of guards and jumps.
- Conflicting writes. The assembler checks whether no two operations complete and write to the same register simultaneously. Again, under the presence of guards this will not always be possible.
- Guarded immediates. Immediate instructions cannot be guarded, so the assembler checks on this.

Main Options

Table 1 contains the main options of the Assembler **tmas**.

Table 1 Main Options

Option	Description
-V	Prints version information.
-eb	Default endianness is big-endian.
-el	Default endianness is little-endian.
-h	Prints a brief help message with options.
-o=file	Uses the specified file for output.
-handcode	Prints additional handcode assistance warnings.

Most of these options are straightforward, so only **-handcode** is described here.

The -handcode option

The **-handcode** option reports some extra potential problems with the assembly code. The option influences two major assembly code checks:

- Conflicting writes. All simultaneous writes to a register are reported, instead of just the ones that the assembler considers to be potentially conflicting.
- Uninitialized reads. The assembler reports reads from registers that have not yet been initialized with a value in the given schedule.

Assembly Syntax

The assembly code format is the textual representation of scheduled TriMedia code. It is used as the communication medium between the TriMedia scheduler and the TriMedia assembler. The inherent characteristics of a VLIW processor and the constraints of the TM-1X00 processor make it ill-suited for direct use by the programmer.

Example

The following is a simple example assembly code routine for the TriMedia chip:

```
(* cycle 0 *)
    IF r1   iadd r5 r6 -> r125 ,            (* alu/Op10 *)
    IF r1   igtr r5 r6 -> r126 ,            (* alu/Op11 *)
    IF r1   igeq r6 r5 -> r127 ,            (* alu/Op20 *)
    IF r1   ld32d(16) r4 -> r125 ,          (* dmem/Op3 *)
    IF r1   nop ;

(* cycle 1 *)
    IF r1   asri(0x2) r125 -> r125 ,        (* shifter/Op12 *)
    IF r1   ijmpt r126 r2 ,                 (* branch/Op18  *)
    IF r1   ijmpf r126 r2 ,                 (* branch/Op19  *)
    IF r1   nop ,
    IF r1   nop ;

(* cycle 2 *)
    IF r1   nop ,
    IF r1   nop ,
    IF r1   nop ,
    IF r1   h_st32d(0) r125 r7 ,            (* dmem/Op13 *)
    IF r1   nop ;

(* cycle 3 *)
    IF r1   nop ,
    IF r1   nop ,
    IF r1   nop ,
    IF r126   h_st32d(0) r6 r8 ,            (* dmem/Op14 *)
    IF r127   h_st32d(0) r5 r8 ;            (* dmem/Op16 *)

(* cycle 4 *)
    IF r1   nop ,
    IF r1   nop ,
    IF r1   nop ,
    IF r126   h_st32d(0) r5 r125 ,          (* dmem/Op15 *)
    IF r127   h_st32d(0) r6 r125 ;          (* dmem/Op17 *)
```

TriMedia has five issue slots, so five operations are specified in each cycle. The **nops** are necessary to specify the operation slot position to the hardware. For instance, in cycle four, the two stores are scheduled into issue slots four and five, because those are the

 ©1999 Philips Semiconductors 10/08/99

only two issue slots from which stores can be issued. Each operation has an optional guard (**IF rx**), the opcode, an optional parenthesized modifier field (if appropriate for the opcode), followed by a list of the argument registers, and an optional destination register (if appropriate for the opcode). The most general form of the operation is written:

```
IF ra opcode(modifier) rb rc -> rd,
```

where **ra** is the guard register, **modifier** is the opcode modifier value (integer constant), **rb** and **rc** are the argument registers, and **rd** is the destination register.

Machine Constraints

When writing an assembly program, you must consider all the constraints of the target machine. In particular, the following must be satisfied as shown in Table 2. The next section describes these constraints in more detail.

Table 2 Machine Constraints in Assembly Code

Name	Description
Register index	The register named must be in the valid range. Hardware registers **r0** and **r1** must not be written to.
Issue slots	There must be exactly *n* operations in each instruction, where *n* is the number of issue slots of the target machine.
Operation property	Use of operations must match the properties declared in the machine description file, such as arity, modifier signedness, range and step, destination register.
Input crossbar	There is a restriction on which operations may issue in which issue slots.
Writeback buses	There must not be more than *n* results produced in a given cycle, where *n* is the number of writeback buses on the target machine.
Functional unit availability	The functional unit that the operation issues on must be available, as governed by the RECOVERY attribute of the functional unit type in the machine description file.

Other Constraints

Some additional restrictions on the input must be obeyed to produce a correct program.

Pseudo-Operations

Pseudo-operations are not allowed in the assembly source code, since they are not implemented in the actual hardware. During compilation, the instruction scheduler maps the pseudo-operations to the real hardware operations.

Control-Flow Fall-Through

The assembly program execution correctness should not rely on control-flow fall-through. For example, any execution of an instruction at a label (branch target) should be a result of a control transfer (FLOW operation). This is required by the instruction compression scheme. The assembler attempts to detect control-flow fall-throughs and flags errors/warnings as appropriate. The assembler cannot detect all such cases due to dynamic behavior of certain assembly code.

Branches are Mutually Exclusive

It is possible to schedule two (or more) branches in one operation, but they have to be mutually exclusive, i.e. only ONE branch can be taken in each instruction. Care should be taken when scheduling more than one branch in an instruction that only one guard evaluates to true. If more than one branch is taken, the outcome is undefined.

Assembly Expressions

The **tmas** assembler allows a limited set of expressions in assembly programs. The sections in this section define the types of expressions recognized by **tmas**.

Expression Syntax

The only operators allowed in an assembly expression are

- Unary plus
- Unary minus
- Binary plus
- Binary minus

An assembly *expression* can be a numeric constant (an integer constant or a floating-point constant), a symbol, or a combination of these using the four operators mentioned previously. Unary operators have higher precedence over binary operators. Parentheses can be used to group subexpressions. Symbols are limited in the way they can be used, as explained in the next section. The precise syntax of the expressions is given by the following rules:

```
expression            ::=  FloatConst | additive_expression .
additive_expression   ::=  unary_expression
                           | additive_expression "+" unary_expression
                           | additive_expression "-" unary_expression.
unary_expression      ::=  primary_expression
                           | "+" unary_expression
                           | "-" unary_expression .
primary_expression    ::=  SymbolName | IntConst
                           | "(" additive_expression ")" .
```


©1999 Philips Semiconductors 10/08/99

Use of Symbols in Assembly Expressions

An assembly expression must be either *absolute* or *relocatable*. An expression is absolute if its value is known at compile time. An expression is relocatable if it contains a symbol. There are some restrictions on the way symbols can be used in assembly expressions. A symbol is allowed in an expression only if the resulting expression is relocatable or absolute. Thus, the addition of a symbol and an integer constant is valid, and so is the difference of two symbols that belong to the same section. When the two symbols whose difference is taken are in the **text** section, the expression computes to the number of instructions between the two instructions.

The following are examples to illustrate these statements. Assume that the two symbols **_sym1** and **_sym2** have been defined previously.

Symbol	Description
_sym1	A simple relocatable item
_sym1 + 10	A simple relocatable expression
10 – _sym2	Not allowed (result cannot be relocated)
_sym1 + _sym2	Not allowed (result cannot be relocated)
_sym1 – _sym2	Allowed if and only if they are of the same segment and the offset is known at assembly time (if text segment, the result is number of instructions starting at **_sym2** and ending at **_sym1**, not including the instruction at **_sym1**)
(_sym1 – _sym2) + _sym3	Valid (equivalent to adding a constant to a symbol)

Assembler Directives

Assembler directives direct the actions of the **tmas** assembler to initialize data, reserve space, export symbols to other files, and so on. The assembler recognizes the following four sections: **text**, **data1**, **data**, and **bss**. The programmer can instruct the assembler to switch to different sections and take appropriate actions within these sections by using assembler directives.

The next section, *Assembler Directives at a Glance,* provides a summary of all the assembler directives. Further discussions elaborate on some of the directives in detail.

Assembler Directives at a Glance

The TriMedia assembler supports the assembly directives shown in Table 3.

Table 3 Assembly Directives

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*[A]	Generate initialized 1-byte values in the current segment, given the comma-separated list of assembly expressions.
.common *symbol*, *size* [, "*segment*" [, *alignment*]]	Declare the symbol to be a FORTRAN-style common area with the given size in bytes. It will be defined either in the optionally given segment or in bss if none was given. The symbol can have an optional alignment, which will be used by the linker during its final linking pass (if the symbol did not resolve to a definition).
.data	Switch current segment to data segment.
.data1	Switch current segment to data1 segment.
.global *list-of-symbols*	Declare the comma-separated list of symbols to be global
.half *list-of-expressions*	Generate initialized 2-byte values in the current segment, given the comma separated list of assembly expressions.
.reserve *symbol*, *size* [, "*segment*" [, *alignment*]]	Define *symbol* in current or optionally given *segment*, reserve *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 text segment.
.text	Switch current segment to text segment.
.word *list-of-expressions*	Generate 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 text segment.

A. *List-of-expressions* denotes a comma-separated list of assembly expressions and list-*of-symbols* is a comma-separated list of symbols.

.text, .data, .data1: Switch Sections

The **tmas** assembler recognizes the following four sections: **text**, **data1**, **data**, and **bss**. The **text** section consists of the assembly instructions to be executed on TriMedia. The **data1** section consists of initialized data that can not be modified at run time. The **data**

 © 1999 Philips Semiconductors 10/08/99

section has initialized data that can be modified during run time, and the **bss** section has all uninitialized data (that is, statically allocated space). The **bss** section is initialized to 0 at run time.

The **tmas** assembler has the concept of *current section*. By default, **tmas** assumes that an assembly file starts with the **text** section as the current section. The assembler directives .**text**, .**data1**, and .**data** specify that the assembler switch the current section to the appropriate section. For example, in the following program, the assembler initializes the first word in the data section to hold the value 234, then generates code in the **text** section, and then initializes the first word in the **data1** section to be 888.

```
.data
.word 234
.text
_label_1:
uimm(22) -> r123, ijmpi(_somewhere_else), nop, nop, nop;
nop,nop,nop,nop,nop;
nop,nop,nop,nop,nop;
nop,nop,nop,nop,nop;
.data1
.word 888
```

.ascii: Character Data

The **. ascii** directive generates the ASCII equivalent of the given string at the current address in the current section. The string can have any of the escape sequences allowed in the ANSI/ISO C Standard. The following are some examples:

```
.ascii   "Hello! How are you? \n\0"
.ascii   "abc\001\023\xf6"
```

Note that a **\0** must be explicitly given at the end of the string if the string is to be null-terminated. In the second example, 6 bytes in memory are initialized (the first 3 bytes with the ASCII values of **a**, **b**, and **c**, followed by two bytes with octal values **001** and **023**, followed by a one byte with hex value **0xF6**).

.byte, .half, .word: Generating Data

The directives .**byte**, .**half**, and .**word** reserve storage locations in the current section and initialize them with the specified values. These directives are not allowed in the **text** section.

The .**byte** directive reserves one byte of space for each expression in the list and initializes the byte with the low-order eight bits of the corresponding expression's value.

The .**half** directive reserves two bytes of space for each expression in the list and initializes the bytes with the low-order 16 bits of the corresponding expression's value.

The .**byte** and .**half** directives do not allow relocatable expressions in their list of expressions. This is because 32-bit addresses will not fit into either the 8-bit or the 16-bit spaces allocated for these directives.

The **.word** directive reserves four bytes of space for each expression in the list and initializes the bytes with the value of the corresponding expression.

The following are some examples:

```
.data
_symname:
.byte 34          (* reserves 1 byte and initializes it with the value 34. *)
.byte 3,45,6      (* reserves 3 bytes initialized with 3,45,and 6.         *)
_sym_2:
.byte 123456      (* reserves 1 byte initialized with 0100 0000 (0x40).    *)
                  (* 123456 (0x1E240) in binary is 1 1110 0010 0100 0000.  *)
.half 123456789   (* reserves 2 bytes initialized with 1100 1101 0001 0101 *)
                  (* (0xCD15). 123456789 (0x75BCD15) in binary is          *)
                  (* 0111 0101 1011 1100 1101 0001 0101.                   *)
.word _symname    (* reserves four bytes initialized with the relocatable  *)
                  (* address of _symname.                                  *)
.half _symname    (* This is illegal. Relocatable expressions are not      *)
                  (* allowed with the .half directive.                     *)
.half (_sym_2 - _symname)    (* This is legal. Two bytes are reserved and  *)
                             (* initialized with the value of the given    *)
                             (* expression (which is 4).                   *)
```

.common: Declare a Common Symbol

The **.common** directive allows the user to declare a symbol to be a FORTRAN-style common area with the given size in bytes. When such a symbol appears in several files, the linker resolves them and eventually the space allocated for the symbol is the maximum of all the sizes given in the files. The format of the **.common** directive is

```
.common symbol, size [, "bss" [ , alignment]]
```

Regardless of whether **bss** is explicitly indicated, the symbol is always taken to be in the **bss** section. When the alignment is not given, its value defaults to one (byte alignment).

The following are examples:

```
.common symbol_1, 10   (* symbol_1 is declared to be in the bss section;   *)
                       (* 10 bytes are reserved for it.*)
.common symbol_2, 12, "bss", 4   (* symbol_2 is declared to be in the bss  *)
                                 (* section; 12 bytes are reserved for it; *)
                                 (* The linker makes sure that symbol_2 is *)
                                 (* aligned to a 4-byte boundary.          *)
```

.reserve: Define a Symbol with Alignment

The **.reserve** directive defines a symbol in the current section (or in the optionally specified section) and reserves the specified number of bytes for the symbol. The optional alignment value is used to ensure that the symbol is aligned to the appropriate byte boundary.

The format of the **.reserve** directive is

```
.reserve symbol, size [,"section" [, alignment]]
```

 ©1999 Philips Semiconductors 10/08/99

The following is an example:

```
.data1
.reserve res_symbol, 12
      (* Reserve 12 bytes for the symbol res_symbol in the data1 section. *)
.reserve another_res, 16, "data", 32
      (* Reserve 16 bytes for the symbol another_res in the data section, *)
      (* aligning it to a 32-byte boundary.                                *)
```

.global: Define a Global Symbol

A program can consist of several *modules*. A symbol defined in one module may be referenced in another. If a symbol defined in one module is to be referenced in another module, the symbol must be made global by using the **.global** directive. If a symbol is not explicitly exported by using the **.global** directive, the symbol does not appear in the global symbol table, and, thus, is not available for use in other modules.

The format of the **.global** directive is:

```
.global list-aof-symbols
```

The following is an example:

```
.data
.global _name
_name:
.word 234      (* _name is a variable available for use in other modules. *)
               (* It is initialized with the value 234.                   *)
.global name2, name3, name4
               (* name2, name3, name4 are also made global. *)
```

.align: Align the Current Address

The **.align** directive advances the current address to the next *n*-byte boundary. For example, this directive could be used to ensure that integers initialized by the **.word** directive start at a word boundary. The **.align** directive is not allowed in the **text** section.

The format of the **.align** directive is:

```
.align size
```

where **size** is an integer that is an integral divisor of the block size of the current section. The linker/loader always ensures that each section when relocated starts at a proper block boundary, where the block size is dependent on the section. This restriction ensures that the alignment is maintained, even when the section is relocated.

Use of Directives and Program Layout

This section includes an example that shows how to switch sections in an assembly program and how to initialize data using assembler directives. Note that the **tmas** assembler assumes that the assembly file starts with the **text** section. Thus, you do not need to start a file with a **.text** directive. However, if you want to define symbols in other sections at

the beginning of the file, then you should start the file with an appropriate directive. There are no restrictions on how many times you can switch sections in an assembly file.

```
.data             (* change from text section to data section. *)
.align 4          (* current address aligned to a word boundary *)
.global aa        (* declare the symbol aa to be global. *)
aa:               (* define the symbol aa in data section. *)
.word 1234        (* initialize the word at aa with value 1234. *)
.byte 3           (* initialize the 5th byte starting from aa *)
.byte 4           (* to 3 and the 6th to 4. *)
.ascii "Hello world\n\0"
                  (* Initialize the next few bytes with ASCII values for *)
                  (* the characters in the string.                       *)
.align 4          (* skip bytes if needed to align the current address   *)
.align 4          (* word boundary.                                      *)

cc:
.half 34, 56, 78, (cc - aa)
.reserve another_data,
                  (* reserve 4 bytes while defining the symbol another_data *)
                  (* the data section. These 4 bytes are filled with zero.  *)
                  (* The symbol another_data gets the current address as    *)
                  (* its value.                                             *)

.text             (* switch to text section. *)
.global _main
_main:

uimm(aa) -> r10, ijmpi(outside), nop, nop,nop;

nop,nop,nop,nop,nop;
nop,nop,nop,nop,nop;
nop,nop,nop,nop,nop;

.data1             (* switch to data1 section. *)
.global bb
.word 0s2.0e30     (* single-precision, floating-point constant *)

.common sym_1, 4, "bss", 4
                   (* declare sym_1 to be in the bss section. *)
```

TM-1*x*00 Constraints

This chapter discusses the **text** section of a TM-1*x*00 assembly program. The **tmas** assembler provides plenty of room for the assembly programmer to exploit parallelism at the instruction level. To utilize this fully, you must be aware of some machine constraints and subtleties. The purpose of this chapter is to present you with such constraints from the point of view of an assembly programmer. The section *Instruction Format and TM-1x00 Constraints* discusses the instruction format and machine constraints that influence the assembly programs. All assembly programmers must know the details presented in this section. Details involving special operations that read/write special registers and MMIO locations are discussed in the section *Special Register Semantics*. Finally, the section *Assembly Program Checklist* discusses a convenient checklist to make sure that an assembly program satisfies all the machine constraints.

 ©1999 Philips Semiconductors 10/08/99

Instruction Format and TM-1*x*00 Constraints

The **text** section contains a sequence of assembly instructions. Each instruction consists of five TM-1*x*00 operations. Instructions are separated by semicolons. A single line can have many instructions, and a single instruction can cross line boundaries.

Instruction Format

The format of an instruction is

```
[<label-field>] [<op_1>,<op_2>,<op_3>,<op_4>,<op_5>;]
```

The label field is optional. If present, it consists of an identifier followed by a colon ":". Any instruction starting with a label is taken to be a branch target. In contrast to most machines, the TriMedia assembler does *not* allow control flow to fall through to a branch target. In other words, the program must be written so that a branch target instruction can be reached only by an explicit jump instruction. (For more discussion of this issue, refer to the section *Control Flow Fall-Through* beginning on page 24.) The identifier used in the label can be used in assembly expressions to denote the address of the instruction.

Five operations are allowed within an instruction. That is, each instruction has five issue slots, and *operation_i* is in issue slot *i*. Each operation (*operation1* through *operation5*) has the following format:

```
[If <guard> ] <opcode> [ (<modifier>) ] [ <operand_1> [ <operand_2> ] ]
[-> destination> ]
```

Note the following about the various fields:

- The *guard* is optional. When the *guard* is present, the operation will be executed if and only if the least-significant bit of the *guard* register is 1. Immediate operations (**uimm** and **iimm**) cannot be guarded.
- The optional *modifier* field can be any assembly expression. Some opcodes do not take a modifier, some allow the modifier value to be anything representable within 32 bits, while some others have more severe restrictions (such as restricting the value to be representable within 7 bits). The table of opcodes in Chapter 6 can be used to find out the modifier ranges allowed for various opcodes.
- The *operand* fields are shown to be optional because some operations do not take any operands while some operations take only one operand. The *destination* field is shown as optional because some operations do not produce any value.
- Since TriMedia is a load/store architecture (that is all memory transactions are via explicit loads and stores), ‹*guard*›, ‹*operand_1*›, ‹*operand_2*›, and ‹*destination*› must be registers. Registers are represented as **rn**, where **n** is the number of the register. Thus, **r5, r98**, and **r121** are valid representations for registers in an operation.
- The fields within an operation must be separated by white-space characters (spaces, tabs, and newlines). Comments are also taken to be white space.

Code Placement Constraints

The TriMedia architecture, while providing a considerable amount of parallelism, imposes certain restrictions on the way operations can be placed within an instruction. This section discusses such constraints. Read this section carefully, because it is very important to understand these constraints before writing assembly programs.

Issue Slot Limitations

Each opcode in the TriMedia assembler belongs to one of the categories listed in Table 4. Each category is implemented by a certain *functional unit* type that determines various properties of the operation (such as latency, recovery and so on). The issue slots in which an operation can be placed depends on its functional unit type. For example, the operations belonging to the functional unit type **BRANCH** can be placed only in the issue slots 2, 3, and 4. Table 4 shows the mapping of the functional unit types to the issue slots. See page 44 for more information about the opcodes.

Table 4 Mapping from Functional Unit Type to Issue Slot

Functional Unit Type	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5
CONST	X	X	X	X	X
ALU	X	X	X	X	X
DMEM				X	X
DMEMSPEC[A]					X[A]
SHIFTER	X	X			
DSPALU	X		X		
DSPMUL		X	X		
BRANCH		X	X	X	
FALU	X			X	
IFMUL		X	X		
FCOMP			X		
FTOUGH		X			

A. When an operation belonging to the functional unit type DMEMSPEC is placed in an instruction, an operation from DMEM is not allowed in that instruction. In addition, an invalidate (dinvalid) or copyback (dcb) operation should not be preceded by any DMEM or DMEMSPEC operation(s) in the previous instruction (HW bug # 21299), and a copyback (dcb) operation should not follow a store operation to the same cache block in the next instruction (HW bug # 21331).

Latency, Recovery, and Delay

The *latency* of a functional unit type defines the number of cycles between the cycle an operation of that functional unit type is issued and the cycle at which the result of the

operation becomes available. For example, **FALU** has a latency of three. This means that if an **fadd** instruction is issued in cycle i, its result is available for use in cycle i+3.

Recovery defines the number of cycles needed to free a functional unit after an operation of that type is issued. All functional units except the **FTOUGH** unit have a recovery of 1. **FTOUGH** has a recovery of 16. For example, if the guard is **TRUE** and a **fsqrt** is issued in cycle i, no **FTOUGH** operations can be issued in cycles i+1 through i+15. (They are allowed in cycle i+16.) However, if the guard is **FALSE**, the recovery time is 2 cycles (as opposed to the expected single cycle) because of a hardware feature. In other words, two **FTOUGH** operations can never be issued sequentially.

Delay (applicable only to **BRANCH** type operations) defines the number of cycles between the issue of a branch operation and the cycle in which the instruction at the location specified by the branch instruction is executed. Thus, in the following example, instructions in cycles i+1, i+2, and i+3 are executed before the instruction at **new_place** is executed, because the delay of **jmpi** is three cycles:

```
(*cycle i *)
nop,
jmpi(new_place),
nop,
ld32d(0) r10 -> r60,
IF r90 ld32d(4) r10 -> r61;

(* cycle i+1 *)
nop,nop,nop,nop,nop;

(* cycle i+2 *)
uimm(1) -> r50,
uimm(2) -> r51,
uimm(3) -> r52,
uimm(4) -> r53,
uimm(5) -> r54;

(* cycle i+3 *)
nop, nop, nop, nop, nop;

...

new_place:

uimm(mem_loc) -> r100,
nop, nop, nop, nop;
```

Number of Writebacks

The maximum number of writebacks allowed in a cycle is five. This means that, in any given cycle, at most five operations that produce a result can complete. Thus, the following piece of code is illegal:

```
nop,
nop,
nop,
ld32d(0) r10 -> r60,
ld32d(4) r10 -> r61;
```

```
nop,nop,nop,nop,nop;

uimm(1) -> r50,
uimm(2) -> r51,
uimm(3) -> r52,
uimm(4) -> r53,
uimm(5) -> r54;

nop,nop,nop,nop,nop;   (* The two loads and the five immediates complete in *)
                       (* this cycle, resulting in a total of seven write- *)
                       (* backs. (ld32d has a latency of 3 cycles. uimm has *)
                       (* a latency of 1 cycle.) Only five are allowed!    *)
```

It is the responsibility of the programmer to ensure that the number of writebacks does not exceed five in any cycle. Exceeding the limit can cause nondeterministic behavior.

Important
Special care should be taken at points in the program that can be reached from several places. The number of writebacks should not exceed five in each possible execution path.

You can use guards to ensure that the number of writebacks does not exceed the limit. Thus, the following is legal code if the guard registers **r90** and **r91** are mutually exclusive (that is only one of them is **TRUE** when the instruction with the loads is executed):

```
nop,
nop,
nop,
IF r91 ld32d(0) r10 -> r60,
IF r90 ld32d(4) r10 -> r61;

nop,nop,nop,nop,nop;

uimm(1) -> r50,
uimm(2) -> r51,
uimm(3) -> r52,
uimm(4) -> r53,
nop;

nop,nop,nop,nop,nop;   (* When r90 and r91 are mutually exclusive,only one *)
                       (* load and four immediates result in writebacks in *)
                       (* this cycle, resulting in a total of five write-  *)
                       (* backs. Thus this code is legal.                  *)

nop, nop, nop, nop, nop;
```

Control Flow Fall-Through

Any instruction that has an associated label is considered to be a branch target. The variable-length instruction format of TriMedia imposes a restriction on the control flow, forbidding control fall-through to a branch target. Thus, the execution of the instruction at a branch target must be through an explicit **BRANCH** type instruction. The following code is thus illegal:

```
(* cycle 0 *)
uimm(1) -> r50,
uimm(2) -> r51,
uimm(3) -> r52,
```

```
uimm(4) -> r53,
uimm(5) -> r54;

(* cycle 1 *)
nop, nop, nop, nop, nop;

(* cycle 2 *)
new_label:   (* Illegal. This instruction is reached without an explicit *)
             (* jump instruction. *)

uimm(mem_loc) -> r100,
nop, nop, nop, nop;
```

WARNING
No control-flow fall through to branch targets.

Boolean Values

When interpreting a bit pattern as a Boolean value, only the least-significant bit is used. A 0 in the least-significant bit is interpreted as **FALSE**, and a 1 as **TRUE**. Thus, if a register has the value **0xFFFFFFF0**, it is interpreted as **FALSE** in contrast to the way it is interpreted in C (which is **TRUE**).

Special Register Semantics

This section is for advanced assembly programmers who intend to use special registers and/or MMIO locations to influence the instruction execution.

PCSW Writes and Reads

writepcsw and rounding mode

writepcsw has a latency of one. Thus, if a **writepcsw** updates the rounding mode in **cycle i**, its value becomes effective in **cycle i+1**.

```
cycle i-1   fadd r50 r51 -> r52,
            nop, nop, nop, nop;
            (* here fadd uses old rounding mode;*)

cycle i     fadd r60 r61 -> r62,
            nop,
            writepcsw r100 r101,
            nop, nop;
            (* this writepcsw is effective in next cycle *)
            (* fadd in this cycle uses old rounding mode;*)
cycle i+1   fadd r70 r71 -> r72,
            nop, nop, nop, nop;
            (* fadd here uses the new rounding mode; *)
```

writepcsw and BSX

The BSX (byte order or endianness) gets set once shortly after or during system booting and is not to be changed again. The precise time when a change in the PCSW.BSX bit

affects loads and stores is undefined and you should refrain from changing the byte order during program execution.

readpcsw, PCSW Modifying Operations, and Exceptions

For any operation whose execution changes some PCSW bits, the time at which the change becomes effective is decided by the latency of the operation. Thus, if an **fadd** that modifies **PCSW** is issued in cycle i, the change in PCSW becomes effective in cycle i+3 (**fadd** has a latency of three). If a store that causes a misaligned store exception is issued in cycle i, then the MSE bit in PCSW gets set in cycle i+3. In both cases, control will pass to the exception handler at the next successful interruptible jump that is issued at or after cycle i+3. When the number of writebacks exceeds five in cycle i, the WBE bit in PCSW will be set in cycle i. Control will pass to the exception handler in the next successful interruptible jump issued at or after cycle i.

All the exception bits that are set by the hardware remain set unless reset by an explicit write to the PCSW register. The following example illustrates the effect of **fadd** that modifies PCSW by generating a floating-point exception:

```
cycle i     fadd r50 r51 -> r52,
            nop, nop, nop, nop;  (* let us say the above fadd modifies PCSW *)

cycle i+1   nop, nop,
            readpcsw -> r53,
            lsli(3) r52 -> r54,
            nop;  (* readpcsw returns old value, lsli uses old value of r52 *)

cycle i+2   nop, nop,
            readpcsw -> r55,
            lsli(3) r52 -> r56,
            nop;  (* readpcsw returns old value, lsli uses old value of r52 *)

cycle i+3   nop, ijmpi(_out_of_here),
            readpcsw -> r60,
            lsli(3) r52 -> r61,
            nop;  (* readpcsw returns new value, lsli uses new value of r52 *)

(* at cycle i+7, control passes to the exception handler  instead of *)
(* passing to out_of_here.                                            *)
```

Changes to DPC

ijmp and readdpc

DPC gets updated at the cycle when an **ijmp** causes the control to transfer to a new location. Thus, if a successful **ijmp** is issued in cycle i to a label **LABEL**, the change in DPC becomes effective in cycle i+4 (that is, at the instruction in **LABEL**).

```
cycle i     nop,
            ijmpi(LABEL),
            nop, nop, nop;
cycle i+1   nop,nop,
            readdpc -> r10,
```

 ©1999 Philips Semiconductors 10/08/99

```
            nop,nop; (* readdpc returns old value of DPC *)
cycle i+2   nop,nop,
            readdpc -> r11,
            nop,nop; (* readdpc returns old value of DPC *)
cycle i+3   nop,nop,
            readdpc -> r12,
            nop,nop; (* readdpc returns old value of DPC *)
            LABEL:
cycle i+4   nop,nop,
            readdpc -> r13,
            nop,nop; (* readdpc returns new value (LABEL)of DPC *)
```

writedpc and readdpc

`writedpc` has a latency of one. Thus the change in DPC is observable in cycle i+1 if a `writedpc` was issued in cycle i.

```
cycle i-1   nop,nop,
            readdpc -> r10,
            nop,nop; (* readdpc returns old value of DPC *)
cycle i     nop,nop,
            writedpc r100,
            nop, nop; (* readdpc returns old value of DPC; *)
cycle i+1   nop,nop,
            readdpc -> r10,
            nop,nop; (* readdpc returns new value of DPC *)
```

writedpc and ijmp

It is possible that both **writedpc** and **ijmp** effect the change in DPC at the same cycle. This happens if a successful **ijmp** is issued in cycle i and a **writedpc** is issued in cycle i+3. In such a case, DPC gets the value written by **writedpc** (that is, software takes precedence over the hardware update).

CCCOUNT and ijmp

CCCOUNT is updated *only when a successful interruptible jump is taken*. The value of CCCOUNT can be accessed only through the instructions **cycles** and **hicycles**. If a successful **ijmp** is issued in cycle i, the new value of CCCOUNT can be accessed by issuing cycles and **hicycles** operations in cycle i+4.

MMIO Location Updates

Note that writes to MMIO locations do not take effect immediately. For example, if a write to the IPENDING location in cycle i generates an interrupt, then the interrupt will not be triggered if an **ijmpi** operation is executed in cycle i+1. The interrupt will be taken if the **ijmpi** operation was executed in cycle i+2. The amount of delay required for a write to an MMIO location, is dependent on the location and this data will be available soon.

Forward Compatibility

When a result producing operation of latency x is issued in cycle i, the value of the destination register is altered only in cycle i+x. This means that in cycles i,...,i+x the destination register can be used for other purposes as illustrated by the following example:

```
cycle i-1  uimm(0x123) -> r10, nop, nop, nop, nop;

cycle i    fadd r20 r21 -> r10,
           iadd r10 r30 -> r31,
           nop, nop, nop;             (* r10 has 0x123 in this cycle *)

cycle i+1  iadd r10 r32 -> r33,
           nop, nop, nop, nop;        (* r10 has 0x123 in this cycle *)

cycle i+2  iadd r10 r32 -> r34,
           nop, nop, nop, nop;        (* r10 has 0x123 in this cycle *)

cycle i+3  fsub r10 r40 -> r41,
           nop, nop, nop, nop;        (* r10 has the result of fadd  *)
```

In this example, both **uimm** and **fadd** write to register **r10**. In spite of **fadd** being issued in cycle i, **r10** has the value 0x123 from the **uimm** operation in cycles i, i+1, and i+2. The use of register **r10** is *overlapping*. To insure forward compatibility, you should not overlap the use of registers. This is because if (in the future) the latency of **fadd** is reduced from three to two, the value used in cycle i+2 will be incorrect. In the example, the register overlap could have been avoided by using different registers for the destinations of **uimm** and **fadd**.

WARNING
Avoid register overlap.

Crossover of Register Writes

Whenever an interruptible jump is issued in cycle i, any operation that has been issued at or before cycle (i + jump delay) must complete at or before cycle (i + jump delay + 1). It is often useful to schedule at the end of a loop those operations that produce results that are used in the first few cycles of the next iteration of the loop. In such situations, either noninterruptible jumps should be used in the loop or any register writes by operations scheduled in the loop should complete at the latest 1 cycle after the interruptible jump completes. Further, the registers used for carrying values from one iteration to the next across interruptible jumps should be global registers since local registers get overwritten by interrupt service routines. (See page -1 for definitions of global and local registers.) Thus, the following code is incorrect:

```
cycle i-1  nop, fsqrt r127 -> r40, nop, nop, nop;
           (* fsqrt completes in cycle i+16 and so is illegal due to the  *)
           (* operation in cycle i *)

cycle i    nop, ijmpi(new_label), nop, nop, nop;

cycle i+1  uimm(0x123) -> r10, nop, nop, nop, nop;
```

 ©1999 Philips Semiconductors 10/08/99

```
cycle i+2  fadd r20 r21 -> r10,
           iadd r10 r30 -> r31,
           nop, nop, nop;
           (* fadd completes in cycle i+5 and so is illegal due to the  *)
           (* operation in cycle i *)

cycle i+3  iadd r10 r32 -> r33,
           nop, nop, nop, nop;

new_label:
cycle i+4  iadd r10 r32 -> r34,
           nop, nop, nop, nop;

cycle i+5  fsub r10 r40 -> r41,
           nop, nop, nop, nop; (* r10 has the result of fadd  *)
```

Assembly Program Checklist

The following checklist can be used to ensure that an assembly language program satisfies all the constraints imposed by the architecture. Many of these conditions are checked and reported by the assembler, but some of them are not. Some of the checks depend on the program semantics and the assembler can, at best, only issue some warnings.

- Are all the operations issued in proper issue slots?

  This is checked by the assembler.

- For each operation issued, have all the computations into the argument registers (including any guard register) finished at the issue cycle? Is this true even in the presence of loops? (This is a semantic check and cannot be verified by the assembler.)
- Do at most five result-producing operations complete at each cycle? Is this true in the presence of loops (that is, merging control flow)? (Again, the assembler cannot check for this condition fully.)
- For every branch target (an instruction with a label), does control flow reach the instruction only by an explicit jump instruction? (This, again, cannot be checked by the assembler.)
- Do all jump instructions jump only to instructions that have a label? (Note that **r2**, the register containing the return pointer, does point to a branch target.)
- At each cycle, is it true that no two operations complete and write to the same register simultaneously?
- Are immediates not guarded?

  The assembler checks this.

- Do all guard values and the conditions in jump instructions rely only on the least-significant bit of the register?

- If an interruptible jump is placed in cycle i, do all operations that are scheduled in or before cycle (i+ jump delay) and write to registers complete at or before cycle (i+ jump delay+ 1)?

Interfacing C with Assembly Language Programs

To obtain optimum performance for certain pieces of code, it is sometimes necessary to write that code in assembly language. This chapter describes how to interface assembly programs with C programs.

Memory Layout and C Calling Conventions

To be able to call routines coded in assembly language from C programs, and vice versa, you must understand the memory layout, C calling conventions, and the register usage conventions of the C compiler. These topics are described in detail in Chapter 8, *TriMedia C/C++ Languages*. We recommend that you go through the parts of that chapter that deal with the calling conventions and memory layout before proceeding further in this chapter, although some of the conventions are explained in the examples.

Using the C Preprocessor

While writing assembly code, it is often convenient to define the registers in terms of symbolic names. Using symbolic names for registers makes the code more readable. In addition, the overlaps in the liveliness of the registers become more obvious with symbolic names, allowing for a better register usage. It may also be necessary to include other files in the assembly program (for example, to initialize a block in the data segment). This can be accomplished by using the directives provided by the C preprocessor and invoking the C preprocessor on the assembly program before passing it on to the **tmas** assembler. The driver program **tmcc** for the TriMedia compilation system provides an option (-x) that can be used to invoke the C preprocessor on assembly programs. With this option, **tmcc** passes the right flags to the C preprocessor so that the C preprocessor does not generate lines starting with a # sign, which the **tmas** assembler does not understand.

Calling Assembly Code from C Code

This section uses examples to illustrate methods to pass data back and forth between assembly code and C code.

Passing Integers and Returning a Value

The following C code passes two integers to a routine written in assembly, which adds them up and returns the sum of the two integers.

 ©1999 Philips Semiconductors 10/08/99

C File

```
/* file : main.c */
#include <stdio.h>
#include <stdlib.h>

extern int add(int, int);

main(){
   int i;

   i = add(4,5);
   printf("added value i is %d\n", i);
}
```

Assembly File

```
(* file : param.s *)
.text
.global _add
_add:

(* add the two parameters and return the result; r5 and r6 have the input
parameters on exit r5 will have the sum; r2 has the return address, so that
the ijmpt operation passes the control back to main(); Note that in the
assembly file, the function is called "_add" and not "add". The compiler pref
ixes the names with an underscore. *)

iadd r5 r6 -> r5, nop, ijmpt r1 r2, nop, nop;

nop,nop,nop,nop,nop;

nop,nop,nop,nop,nop;

nop,nop,nop,nop,nop;  (* We have three cycles of nops to fill in the
                         delay slots of ijmpt in the first instruction. *)
```

Compiling and Running

The following code calls the C compiler on main.c and param.s using **tmcc**. This results in a.out being generated. The code then invokes **tmsim**, the TriMedia machine-level simulator on a.out, to view the results.

```
tmcc main.c param.s
main.c:
param.s:
tmsim a.out
added value i is 9
```

Accessing Arguments from the Stack

When the arguments passed to a function do not fit in the four argument registers (**r5, r6, r7**, and **r8** are used to pass arguments), the arguments that do not fit are passed on the stack. This example shows how to access such arguments. The example computes the maximum of the six integers in the assembly code.

C File

```
#include <stdlib.h>
#include <stdio.h>

extern int maximum(int, int, int, int, int, int);
main(){
   int m;

   m = maximum(40, 2, 6, 5, 8, 1);
   printf("max is %d\n", m);
}
```

Assembly File

```
.global _maximum
_maximum:

imax r5 r6 -> r127,
nop,
imax r7 r8 -> r126,
ld32d(16) r4 -> r125,
ld32d(20) r4 -> r124;

nop, nop, nop, nop, nop;

imax r127  r126  -> r123,
nop, nop, nop, nop;

imax r125  r124  -> r122,
ijmpt r1 r2,
nop, nop, nop;

nop, nop, nop, nop, nop;

imax r123  r122  -> r5,
nop, nop, nop, nop;

nop, nop, nop, nop, nop;
```

Compiling and Running

```
tmcc main.c max.s
main.c:
max.s:
tmsim a.out
max is 40
```

Points to Note

Observe the following with respect to the previous assembly program:

- **imax** is a **DSPALU** type operation and has a latency of two. The second slot in the first cycle is a **nop** because **DSPALU** operations can only be issued in slots one and three.
- The loads in the first cycle belong to the **DMEM** functional unit type and have a latency of three. They are loading the last two parameters passed to the function

 © 1999 Philips Semiconductors 10/08/99

_maximum. r4 is the stack pointer of the caller of this function and the arguments are located in ascending order from that stack pointer (which is the frame pointer for the current function).

- The jump **ijmpt r1 r2** completes even before the operation **imax r123 r122 –> r5** has completed. As a result, **r5** is written to only after the control gets back to the main function. This works because there are no zero-latency operations. It should be noted that you cannot allow an overlap of more than one cycle. For example, issuing the same **imax** operation in the last cycle could result in an error. It is possible that the first cycle executed after returning from this function has a latency of one operation writing into **r5**, which would result in an undefined value in **r5**. This prevents moving up the **ijmpt** operation by one cycle.
- The intermediate results are computed into registers starting from **r127** downward. The reason is that the first 32 registers are reserved for global register allocation and must be saved and restored if you must use them. Secondly, in the future, more registers may be reserved for global register allocation (say up to 64) and, thus, it is safer to use registers from the higher end to maintain forward compatibility.

Passing Structure Arguments

Structure arguments are always passed on the stack and not in argument registers. The following example illustrates passing structs to an assembly function.

C File

```
typedef struct {
   int a;
   int b;
} sname;
extern int foo(int, sname, int);
main(){
   sname st;
   int i;

   st.a = 12;
   st.b = 34;
   i = foo(33, st, 44);
   printf("i is %d\n", i);
}
```

Assembly File

```
.global _foo
_foo:

iadd r5 r6 -> r127,        (* r127 gets 33 + 44 *)
nop,nop,
ld32d(4) r4 -> r126,       (* r126 gets 12      *)
ld32d(8) r4 -> r125;       (* r125 gets 34      *)

nop,                       (* result of iadd above available here. *)
ijmpt r1 r2,
nop, nop, nop;

nop, nop, nop, nop, nop;

iadd r125 r126 -> r124,    (* result of loads available here. *)
nop, nop, nop, nop;

iadd r127 r124 -> r5,
nop, nop, nop, nop;
```

Compiling and Running

The following code calls the C compiler on main.c and foo.s using **tmcc. T**his results in a.out being generated. The code then invokes **tmsim**, the TriMedia machine-level simulator, on a.out to view the results.

```
tmcc main.c foo.s
main.c:
foo.s:
tmsim a.out
i is 123
```

Points to Note

- st.a and st.b are at locations **(fp+4)** and **(fp+8)**, where **fp** is the frame pointer of the function **_foo** (which is the stack pointer of the caller). Integer arguments 33 and 44 are passed in registers **r5** and **r6**. (**fp+0**) and (**fp+12**) are allocated on stack for these arguments, but those locations are not initialized.

Implementing Loops in Assembly Programs

This section provides a larger example that involves writing a loop. The example illustrates the use of the C preprocessor, and also is a good case study for becoming familiar with the various machine constraints. The example sums up the elements of an array and prints the value. The code for summing up the elements of the array is written in assembly. The comments in the assembly file indicate how the assembly code is structured.

 ©1999 Philips Semiconductors 10/08/99

C File

```
#include <stdio.h>
#include <stdlib.h>

#include <custom_defs.h> /* included for using cycles() custom_op. */

#define ARR_SIZE 2000
int arr[ARR_SIZE];
int sum_val;
int arr_size = ARR_SIZE;

main(){
   int i;
   int enter_cycles = 0, end_cycles = 0;

   for( i=0; i < ARR_SIZE; i++ ){
      arr[i] = i;
   }
   sum_val=0;
   enter_cycles = cycles();

   sum_val = sum10(arr, ARR_SIZE);

   end_cycles = cycles();
   printf("sum is %d \n", sum_val);
   printf("cycles is %d \n", (end_cycles - enter_cycles));
}
```

Assembly File

```
(* implement the following function in assembly:
   sum10(int *a, int arr_size){
      int sum_val, i;
      sum_val = 0; i = 0;
      while (arr_size - i >= 10) {
         sum_val = sum_val + a[i] + a[i+1] + a[i+2] + a[i+3] +
         a[i+4] + a[i+5] + a[i+6] + a[i+7] + a[i+8] + a[i+9];
         i += 10;
      }
      while (i < arr_size) {
         sum_val += a[i];
      }
      return sum_val;
   }
*)

(* loads for next iteration should be interspersed with adds for the
   current loop. The logic is:*)
init_loop: set up for the loop. Be simple here
   1. _a array base is in register r5
   2. array size is passed in register r6
   3. load 10 values a[0] ... a[9] to set up for the first iteration of the
      loop
   4. set index to 10
   5. compute the loop condition that gets used by the jumps in the loop
      (to determine if there is a second iteration of the loop)
   6. jmp to loop  (if number of elements in the array >= 10;
      otherwise, jump to exit_loop)
```

```
loop:
   1. load values for next loop
   2. add values of current loop
   3. compute the loop condition that gets used in the next iteration.
      Doing this helps in scheduling the compute operations of the current
      even after the jump is placed
   4. jmp back to loop or to loop_exit

exit_loop: take care of exit conditions;
   1. compute the sum of the leftover elements of the array
   2. at most 9 are left - the array values are already in registers -
      loaded from the loop just add them up after finding out how many
      to be added;
*)

(******************ACTUAL CODE STARTS HERE*************************)
#define RLOOP_INDEX r127   /* keeps track of the number of iterations.    */
#define RSUM_VAL r125      /* the current sum.                            */
#define RINIT_COND r124    /* the condition that determines loop entry    */
#define RLOOP_LABEL r123   /* the address of first instruction of loop.   */
#define RLOOP_COND r122    /* condition to see if the loop to be repeated.*/
#define REXIT_LOOP r121    /* address of the first ins. after the loop.   */
#define RREMAINS r120      /* number of elements yet to be processed    . */
                           /* after the loop.                             */

/* The following registers are used for holding values from the
   array and the intermediate sums. RA_n stands for the register
   holding the value in a[i+n], where 0 < n < 10. Registers that
   have more than 1 digit as a suffix hold the corresponding
   sums - for example, RA_0123 has the sum a[i]+a[i+1]+a[i+2]+a[i+3]. */
#define RA_0 r119
#define RA_1 r118
#define RA_2 r117
#define RA_3 r116
#define RA_4 r115
#define RA_5 r114
#define RA_6 r113
#define RA_7 r112
#define RA_8 r111
#define RA_9 r110
#define RA_01 r109
#define RA_23 r108
#define RA_45 r107
#define RA_67 r106
#define RA_89 r105
#define RA_012 r104
#define RA_456 r103
#define RA_4567 r102
#define RA_0123 r101
#define RA_01234 r100
#define RA_012345 r99
#define RA_0123456 r98
#define RA_01234567 r97
#define RA_012389 r96
#define RA_012345678 r95
#define RA_0123456789 r94

/* The following are for processing the array elements after the loop. */
#define RZERO_LEFT r93
#define RONE_LEFT r92
#define RTWO_LEFT r91
#define RTHREE_LEFT r90
```

 ©1999 Philips Semiconductors 10/08/99

```
#define RFOUR_LEFT r89
#define RFIVE_LEFT r88
#define RSIX_LEFT r87
#define RSEVEN_LEFT r86
#define REIGHT_LEFT r85
#define RNINE_LEFT r84

.text
.global _sum10
_sum10:
init_loop:

   uimm(20)   -> RLOOP_INDEX,
   uimm(10)   -> RINIT_INDEX,
   iadd r0 r0 -> RSUM_VAL,
   nop,nop;

   igtri(10) r6 -> RINIT_COND,
   nop,
   uimm(loop)   -> RLOOP_LABEL,

   ld32d(0) r5  -> RA_0,
   ld32d(4) r5  -> RA_1;
   nop,nop,nop,
   ld32d(8) r5  -> RA_2,
   ld32d(12) r5 -> RA_3;

   nop,nop,nop,
   ld32d(16) r5 -> RA_4,
   ld32d(20) r5 -> RA_5;

   igtr RLOOP_INDEX r6 -> RLOOP_COND,
   IF RINIT_COND jmpi(exit_loop),
   jmpf RINIT_COND RLOOP_LABEL,
   ld32d(24) r5 -> RA_6,
   ld32d(28) r5 -> RA_7;

   iaddi(40) r5        -> r5,
   uimm(exit_loop)     -> REXIT_LOOP,
   isub RLOOP_INDEX r6 -> RREMAINS,
   ld32d(32) r5        -> RA_8,
   ld32d(36) r5        -> RA_9;

   nop,nop,nop,nop,nop;
   nop,nop,nop,nop,nop;

loop:
   iaddi(10) RLOOP_INDEX -> RLOOP_INDEX,
   iadd RA_0 RA_1        -> RA_01,
   iadd RA_2 RA_3        -> RA_23,
   ld32d(0) r5           -> RA_0,
   ld32d(4) r5           -> RA_1;

   iadd RA_4 RA_5 -> RA_45,
   jmpf RLOOP_COND RLOOP_LABEL,
   jmpt RLOOP_COND REXIT_LOOP,
   ld32d(8) r5  -> RA_2,
   ld32d(12) r5 -> RA_3;

   iadd RA_01 RA_23 -> RA_0123,
   iadd RA_6 RA_7   -> RA_67,
   iadd RA_8 RA_9   -> RA_89,
```

```
   ld32d(16) r5   -> RA_4,
   ld32d(20) r5   -> RA_5;

   iadd RA_45 RA_67   -> RA_4567,
   iadd RA_0123 RA_89 -> RA_012389,

   igtr RLOOP_INDEX r6 -> RLOOP_COND,
   ld32d(24) r5        -> RA_6,
   ld32d(28) r5        -> RA_7;

   iadd RA_0123456789 RSUM_VAL -> RSUM_VAL,
   iadd RA_012389 RA_4567      -> RA_0123456789,
   iaddi(40) r5 -> r5,
   ld32d(32) r5 -> RA_8,
   ld32d(36) r5 -> RA_9;

exit_loop:
   isub RLOOP_INDEX r6         -> RREMAINS,
   iadd RA_0123456789 RSUM_VAL -> RSUM_VAL,
   nop,nop,nop;

   nop, nop, nop, nop, nop;
   (* to prevent more than 5 writebacks without this instruction, loads *)
   (* form the last cycle in the loop and the ieql is in the next cycle in *)
   (* the next cycle all complete together - we have seven writebacks *)

   ieqli(20) RREMAINS -> RZERO_LEFT,
   ieqli(19) RREMAINS -> RONE_LEFT,
   ieqli(18) RREMAINS -> RTWO_LEFT,
   ieqli(17) RREMAINS -> RTHREE_LEFT,
   ieqli(16) RREMAINS -> RFOUR_LEFT;

   ieqli(15) RREMAINS -> RFIVE_LEFT,
   ieqli(14) RREMAINS -> RSIX_LEFT,
   ieqli(13) RREMAINS -> RSEVEN_LEFT,
   ieqli(12) RREMAINS -> REIGHT_LEFT,
   ieqli(11) RREMAINS -> RNINE_LEFT;

   IF RONE_LEFT iadd RSUM_VAL RA_0 -> RSUM_VAL,
   iadd RA_0 RA_1 -> RA_01,
   iadd RA_2 RA_3 -> RA_23,
   iadd RA_4 RA_5 -> RA_45,
   iadd RA_6 RA_7 -> RA_67;

   IF RTWO_LEFT iadd RSUM_VAL RA_01 -> RSUM_VAL,
   iadd RA_01 RA_2  -> RA_012,
   iadd RA_01 RA_23 -> RA_0123,
   iadd RA_45 RA_6 - > RA_456,
   iadd RA_45 RA_67 -> RA_4567;

   IF RTHREE_LEFT iadd RSUM_VAL RA_012 -> RSUM_VAL,
   iadd RA_0123 RA_4 -> RA_01234,
   iadd RA_0123 RA_45 -> RA_012345,
   iadd RA_0123 RA_456 -> RA_0123456,
   iadd RA_0123 RA_4567 -> RA_01234567;

   IF RFOUR_LEFT iadd RSUM_VAL RA_0123     -> RSUM_VAL,
   IF RFIVE_LEFT iadd RSUM_VAL RA_01234    -> RSUM_VAL,
   IF RSIX_LEFT iadd RSUM_VAL RA_012345    -> RSUM_VAL,
   IF RSEVEN_LEFT iadd RSUM_VAL RA_0123456 -> RSUM_VAL,
   iadd RA_01234567 RA_8 -> RA_012345678;
```

 ©1999 Philips Semiconductors 10/08/99

```
IF REIGHT_LEFT iadd RSUM_VAL  RA_012345678 -> RSUM_VAL,
ijmpt r1 r2,
nop, nop, nop;

iadd r0 RSUM_VAL -> r5,
nop,nop,nop,nop;

nop,nop,nop,nop,nop;
nop,nop,nop,nop,nop;
```

Points to Note

- Use of the C preprocessor to give logical names to the registers.
- Loop is reached by an explicit jump instruction **jmpf RINIT_COND RLOOP_LABEL**; This is to avoid falling through to a branch target.
- It is possible that some of the loads are loading undefined values. These will not get used. For example, if the size of the array is seven, there will be three extra loads and these will be useless.
- An extra cycle after the loop that has only **nop**'s prevents the number of writebacks from exceeding five in a later cycle.
- The loop for adding the elements adds ten elements in one iteration. Unrolling the loop further does not help. The reason is that in each cycle, you can have at most two loads. As a result, the total speedup is limited to a factor of two. This has already been achieved in this program. Thus, the program is optimal up to a constant additive factor. Since the goal of this example is to illustrate making the loops efficient, emphasis is placed only on the loop portion. Thus, the code in the **init_loop** and **exit_loop** portions are written for ease of understanding and not for optimality.
- Noninterruptible jumps are used in the loop to avoid saving and restoring of registers. This can increase the interrupt latency. If interrupt latency is important, you must use interruptible jumps here. In that case, the values that are carried across the interruptible jumps can use global registers. Code in the **init_loop** and **exit_loop** portions will need modification to save and restore these registers. Also, in the presence of interruptible jumps, writes to registers should complete at the latest single cycle after the jump completes. That is, if an interruptible jump is issued in cycle i, then all register writes that are part of the loop should complete at the latest in cycle (i + jump delay). Code in the loop portion must be modified to satisfy this condition.

C Variables and Corresponding Assembly Directives

In the following example showing how C variables get transformed into assembly declarations for final resolution by the linker, note the following:

- All initialized global variables are declared to be global using the assembler **.global** directive. The variable is put in the **data** section using the **.data** directive and initialized appropriately using **.word/.half/.byte/.ascii** directives.

- All initialized static variables (global to the module) are handled similar to the previous case, except that they are not made global (that is, the **.global** directive is not used for them).
- Initialized static variables within a function are handled similar to the previous case, except that the name of the variable changes to accommodate C semantics.
- Uninitialized global variables are declared as common by using the **.common** assembler directive. The variable is put in the **bss** section. The space for these variables is allocated only at link time.
- Uninitialized static variables (global to the module) are declared to be in the **bss** section using the **.reserve** directive.
- Uninitialized static variables within a function are handled similar to the previous case, except that the name of the variable changes to accommodate C semantics.

C File with Declarations

```
int initialized_global=1;
static int uninitialized_static;
static int initialized_static=1;
int uninitialized_global;
extern int external_var;

foo(){
   int initialized_local =1;
   static int initialized_static_local=10;
   static int uninitialized_static_local;

   external_var=1;
}
```

Corresponding Assembly File

```
(* TriMedia scheduler 1.0a15SunOS (v0.05.3.1) Mon Apr 8 21:15:18 PDT 1996
   [compiled with SELF_CHECK]
   activated on Tue Apr 9 13:07:36 1996
   /t/syssoft/build/tcs1.0a15SunOS/bin/tmsched -o=ext.s -eb /t/syssoft/build/
   tcs1.0a15SunOS/lib/tm1.md /usr/tmp/baaa12016.t
*)

.fileinfo.stabs "/t/syssoft/build/tcs1.0a15SunOS/bin/tmsched -o=ext.s -eb /t/
syssoft/build/tcs1.0a15SunOS/lib/tm1.md /usr/tmp/baaa12016.t" 52 0 0 0;
.fileinfo.stabs "-o=ext.s -eb /t/syssoft/build/tcs1.0a15SunOS/lib/tm1.md /
usr/tmp/baaa12016.t" 60 0 0 829080456;
.data
.align 4
.global _initialized_global
_initialized_global:
.word 1
.align 4
_initialized_static:
.word 1
.align 4
_initialized_static_local.LS0:
.word 10
```

 ©1999 Philips Semiconductors 10/08/99

```
.reserve _uninitialized_static_local.LS1, 4, "bss", 4
.text
.global _foo
_foo:
__foo_DT_0:

(* schedule generated by priority scheduler *)

(* tree _foo (0) *)

.treeinfo regmask "0x00000000000000000000001ffffffff";

(* cycle 0 *)
   IF r1       uimm(_external_var) -> r127,     (* const/Op3 *)
   IF r1       ijmpt r1 r2,                     (* branch/Op6 *)
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ;

(* cycle 1 *)
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ,
   IF r1       h_st32d(0) r1 r127,              (* dmem/Op2 *)
   IF r1       nop ;

(* cycle 2 *)
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ;

(* cycle 3 *)
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ,
   IF r1       nop ;

.reserve _uninitialized_static, 4, "bss", 4
.common  _uninitialized_global, 4, "bss", 4
```

Opcodes

The TriMedia chip has 12 functional unit types. Each operation in the TriMedia instruction set can accordingly be classified by its functional unit type. The properties latency, delay, and recovery are characteristics of the functional unit type, so all operations of a given functional unit type have the same latency, delay, and recovery (as applicable). The functional unit type also determines the issue slots in which operations can be issued, as discussed in TM-1x00 Constraints, starting on page 20.

In the section *Functional Unit Types*, each TriMedia operation is listed by functional unit types. In the section TriMedia Opcodes, each opcode is listed in alphabetical order.

Note
With the introduction of the TM-1100 and subsequent TriMedia processors, several new opcodes were added. These opcodes are described in the appropriate TriMedia data book(s). Note also that the operation of the shift operations has changed for shift values greater than 32. While this result was indeterminate on the TM-1000 (and non-zero), the TM-1100 was changed to guarantee a zero result to ASR or ASL greater than 32 bits.

Functional Unit Types

The following subsections list the latency, legal issue slots, and operations for each functional unit type. Recovery is one for all functional unit types except for the **FTOUGH** unit type, for which it is 16. Note that special register operations such as **writepcsw**, **writedpc**, and so on, are part of the **FCOMP** unit type. Even though **DMEM** operations can occur in slots four and five, in general, an additional restriction is that, if a **DMEMSPEC** operation is issued in a cycle, then a **DMEM** operation cannot be issued in the same cycle.

ALU

```
LATENCY 1
ISSUE SLOTS 1 2 3 4 5
OPERATIONS
   iadd isub
   igtr igeq ieql ineq
   ileqi igtri igeqi ilesi ieqli ineqi
   ugtr ugeq
   uleqi ugtri ugeqi ulesi ueqli uneqi
   bitand bitor bitxor bitandinv
   bitinv h_iabs
   sex16
   iaddi isubi
   carry
   izero inonzero
   packbytes
   mergemsb mergelsb pack16msb pack16lsb
   ubytesel ibytesel ;
```

BRANCH

```
DELAY 3
ISSUE SLOTS 2 3 4
OPERATIONS
   jmpf jmpt ijmpf ijmpt ijmpfnse ijmptnse jmpi ijmpi iclr ;
```

CONST

```
LATENCY 1
ISSUE SLOTS 1 2 3 4 5
OPERATIONS uimm iimm;
```

 ©1999 Philips Semiconductors 10/08/99

DMEM

```
LATENCY 3
ISSUE SLOTS 4 5
Cannot issue a dmem operation in a cycle in which dmemspec is issued.
OPERATIONS
   ild8d uld8d ild16d uld16d ld32d h_st8d h_st16d h_st32d
   ild8r uld8r ild16r uld16r ld32r
   ild16x uld16x ld32x ;
```

DMEMSPEC

```
LATENCY 3
ISSUE SLOTS 5
Cannot issue a dmem operation in a cycle in which dmemspec is issued.
OPERATIONS
  dcb dinvalid
  rdtag rdstatus
  prefd prefr pref16x pref32x prefsize
  allocd allocr allocx ;
```

DSPALU

```
LATENCY 2
ISSUE SLOTS 1 3
OPERATIONS
   ume8ii ume8uu
   dspiadd dspisub dspuadd dspusub h_dspiabs
   dspidualadd dspidualsub h_dspidualabs
   iavgonep iflip
   iclipi uclipi uclipu
   quadavg dspuquadaddui
   imax imin ;
```

DSPMUL

```
LATENCY 3
ISSUE SLOTS 2 3
OPERATIONS
   ifir16 ufir16
   ifir8ii ifir8ui ufir8uu
   dspidualmul
   quadumulmsb ;
```

FALU

```
LATENCY 3
ISSUE SLOTS 1 4
OPERATIONS
   fadd fsub fabsval
   ifixieee ufixieee ifixrz ufixrz
   ifloat ufloat ifloatrz ufloatrz
   faddflags fsubflags fabsvalflags
   ifixieeeflags ufixieeeflags ifixrzflags ufixrzflags
   ifloatflags ufloatflags ifloatrzflags ufloatrzflags ;
```

FCOMP

```
LATENCY 1
ISSUE SLOTS 3
OPERATIONS
   fgtr fgeq feql fneq
   fsign
   fgtrflags fgeqflags feqlflags fneqflags
   fsignflags
   readpcsw writepcsw cycles hicycles
(* these are here for VLSI opportunistic reasons *)
   readdpc writedpc readspc writespc;
```

FTOUGH

```
LATENCY 17
RECOVERY 16
ISSUE SLOTS 2
OPERATIONS
  fdiv fsqrt
  fdivflags fsqrtflags;
```

IFMUL

```
LATENCY 3
ISSUE SLOTS 2 3
OPERATIONS
  fmul imul umul imulm umulm dspimul dspumul
  fmulflags ;
```

SHIFTER

```
LATENCY 1
ISSUE SLOTS 1 2
OPERATIONS
  asli roli asri lsri asl rol asr lsr  (* lsl => asl, lsli => asli *)
  funshift1 funshift2 funshift3 ;
```

TriMedia Opcodes

Table 5, following, summarizes the syntax, definition, latency, functional unit type, appropriate issue slots, and modifier ranges for all TriMedia opcodes: Shaded boxes indicate the opcodes that are available available only in the TM-1100.

 ©1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
asl src1 src2 → dst	arithmetic shift left $n \leftarrow r_{src2}<4:0>$ $r_{dst}<31:n> \leftarrow src1<31-n:0>$ $r_{dst}<n-1:0> \leftarrow 0$	1, shifter	1,2
asli(n) src1 → dst	arithmetic shift left immediate $r_{dst}<31:n> \leftarrow r_{src1}<31-n:0>$ $r_{dst}<n-1:0> \leftarrow 0$	1, shifter	1,2 0 to 31
asr src1 src2 → dst	arithmetic shift right $n \leftarrow r_{src2}<4:0>$ $r_{dst}<31:31-n> \leftarrow r_{src1}<31>$ $r_{dst}<30-n:0> \leftarrow r_{src1}<31:n>$	1, shifter	1,2
asri(n) src1 → dst	arithmetic shift right immediate $r_{dst}<31:31-n> \leftarrow r_{src1}<31>$ $r_{dst}<30-n:0> \leftarrow r_{src1}<31:n>$	1, shifter	1,2 0 to 31
bitand src1 src2 → dst	bitwise logical AND $r_{dst} \leftarrow r_{src1}\ \&\ r_{src2}$	1, alu	1,2,3,4,5
bitandinv src1 src2 → dst	bitwise logical AND NOT $r_{dst} \leftarrow r_{src1}\ \&\ {\sim}r_{src2}$	1, alu	1,2,3,4,5
bitinv src1 → dst	bitwise logical NOT $r_{dst} \leftarrow {\sim}r_{src1}$	1, alu	1,2,3,4,5
bitor src1 src2 → dst	bitwise logical OR $r_{dst} \leftarrow r_{src1} \mid r_{src2}$	1, alu	1,2,3,4,5
bitxor src1 src2 → dst	bitwise logical exclusive OR $r_{dst} \leftarrow r_{src1} \wedge r_{src2}$	1, alu	1,2,3,4,5

carry src1 src2 → dst	carry bit from unsigned add
	if $(r_{src1} + r_{src2}) < 2^{32}$ then
	$r_{dst} \leftarrow 0$
	else
	$r_{dst} \leftarrow 1$;
		1, alu	1,2,3,4,5

cycles → dst	read clock cycle counter least significant word $r_{dst} \leftarrow CCCOUNT<31:0>$	1, fcomp	3

Table 5 TriMedia Opcodes (Continued)

Syntax	**Meaning**	**Latency, FU Type**	**Issue Slots, Modifier Range**

dcb(d) src1	data cache copy back
	if rguard then {
	addr ← r_{src1} + d
	if dcache_valid_addr(addr) &&
	dcache_dirty_addr(addr) then {
	dcache_copyback_addr(addr)
	dcache_reset_dirty_addr(addr)
	}
	}
		3, dmem-spec	5,
			–256 to 252 by 4

dinvalid(d) src1	invalidate data cache block
	if rguard then {
	addr ← r_{src1} + d
	if dcache_valid_addr(addr) then {
	dcache_reset_valid_addr(addr)
	dcache_reset_dirty_addr(addr)
	}
	}
		3, dmem-spec	5,
			–256 to 252 by 4

dspiadd src1 src2 → dst	clipped signed add
	temp ← sign_ext32to64(r_{src1}) +
	sign_ext32to64(r_{src2})
	if temp < 0xffffffff80000000 then
	r_{dst} ← 0x80000000
	else if temp > 0x000000007fffffff then
	r_{dst} ← 0x7fffffff
	else
	r_{dst} ← temp
		2, dspalu	1,3

dspidualadd src1 src2 → dst	dual clipped add of signed 16-bit halfwords temp1 ← sign_ext16to32(r_{src1}<15:0>) + sign_ext16to32(r_{src2}<15:0>) temp2 ← sign_ext16to32(r_{src1}<31:16>) + sign_ext16to32(r_{src2}<31:16>) if temp1 < 0xffff8000 then temp1 ← 0x8000 if temp2 < 0xffff8000 then temp2 ← 0x8000 if temp1 > 0x7fff then temp1 ← 0x7fff if temp2 > 0x7fff then temp2 ← 0x7fff r_{dst}<31:16> ← temp2<15:0> r_{dst}<15: 0> ← temp1<15:0>	2, dspalu	1,3

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
dspidualmul src1 src2 → dst	dual clipped multiply of signed 16-bit halfwords temp1 ← sign_ext16to32(r_{src1}<15:0>) × sign_ext16to32(r_{src2}<15:0>) temp2 ← sign_ext16to32(r_{src1}<31:16>) × sign_ext16to32(r_{src2}<31:16>) if temp1 < 0xffff8000 then temp1 ← 0x8000 if temp2 < 0xffff8000 then temp2 ← 0x8000 if temp1 > 0x7fff then temp1 ← 0x7fff if temp2 > 0x7fff then temp2 ← 0x7fff r_{dst}<31:16> ← temp2<15:0> r_{dst}<15: 0> ← temp1<15:0>	3, dspmul	2,3
dspidualsub src1 src2 → dst	dual clipped subtract of signed 16-bit halfwords temp1 ← sign_ext16to32(r_{src1}<15:0>) – sign_ext16to32(r_{src2}<15:0>) temp2 ← sign_ext16to32(r_{src1}<31:16>) – sign_ext16to32(r_{src2}<31:16>) if temp1 < 0xffff8000 then temp1 ← 0x8000 if temp2 < 0xffff8000 then temp2 ← 0x8000 if temp1 > 0x7fff then temp1 ← 0x7fff if temp2 > 0x7fff then temp2 ← 0x7fff r_{dst}<31:16> ← temp2<15:0> r_{dst}<15: 0> ← temp1<15:0>	2, dspalu	1,3

dspimul src1 src2 → dst	clipped signed multiply
	temp ← sign_ext32to64(r_{src1}) ×
	sign_ext32to64(r_{src2})
	if temp < 0xffffffff80000000 then
	r_{dst} ← 0x80000000
	else if temp > 0x000000007fffffff then
	r_{dst} ← 0x7fffffff
	else
	r_{dst} ← temp<31:0>
		3, ifmul	2,3

dspisub src1 src2 → dst	clipped signed subtract
	temp ← sign_ext32to64(r_{src1}) –
	sign_ext32to64(r_{src2})
	if temp < 0xffffffff80000000 then
	r_{dst} ← 0x80000000
	else if temp > 0x000000007fffffff then
	r_{dst} ← 0x7fffffff
	else
	r_{dst} ← temp<31:0>
		2, dspalu	1,3

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

dspuadd src1 src2 → dst	clipped unsigned add
	temp ← zero_ext32to64(r_{src1}) +
	zero_ext32to64(r_{src2})
	if (unsigned)temp > 0x00000000ffffffff then
	r_{dst} ← 0xffffffff
	else
	r_{dst} ← temp<31:0>
		2, dspalu	1,3

dspumul src1 src2 → dst	clipped unsigned multiply
	temp ← zero_ext32to64(r_{src1}) ×
	zero_ext32to64(r_{src2})
	if (unsigned)temp > 0x00000000ffffffff then
	r_{dst} ← 0xffffffff
	else
	r_{dst} ← temp<31:0>
		3, ifmul	2,3

dspuquadaddui src1 src2 → dst	quad clipped add of unsigned/signed bytes
	for(i ← 0, m ← 31, n ← 24; i < 4;
	i ← i + 1, m ← m – 8, n ← n – 8) {
	temp ← zero_ext8to32(r_{src1}<m:n>) +
	sign_ext8to32(r_{src2}<m:n>)
	if temp < 0 then
	r_{dst}<m:n> ← 0
	else if temp > 0xff then
	r_{dst}<m:n> ← 0xff
	else
	r_{dst}<m:n> ← temp<7:0>
	}
		2, dspalu	1,3

dspusub src1 src2 → dst	clipped unsigned subtract
	temp ← zero_ext32to64(r_{src1}) –
	zero_ext32to64(r_{src2})
	if (unsigned)temp > 0x00000000ffffffff then
	r_{dst} ← 0xffffffff
	else
	r_{dst} ← temp<31:0>
		2, dspalu	1,3

dualasr src1 src2 → dst	dual-16 arithmetic shift right
	n ← r_{src2}<3:0>
	r_{dst}<31:31–n> ← r_{src1}<31>
	r_{dst}<30–n:16> ← r_{src1}<30:16+n>
	r_{dst}<15:15–n> ← r_{src1}<15>
	r_{dst}<14–n:0> ← r_{src1}<14:n>
	if rsrc2<31:4> != 0 then {
	r_{dst}<31:16> ← r_{src1}<31>
	r_{dst}<15: 0> ← r_{src1}<15>
	}
		1, shifter	1,2

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
dualiclipi src1 src2 → dst	dual 16 clip signed to signed r_{dst}<31:16> ← min(max(r_{src1}<31:16>, –r_{src2}<15:0>–1), r_{src2}<15:0>) r_{dst}<15:0> ← min(max(r_{src1}<15:0>, –r_{src2}<15:0>–1), r_{src2}<15:0>)	2, dspalu	1,3
dualuclipi src1 src2 → dst	dual-16 clip signed to unsigned r_{dst}<31:16> ← min(max(r_{src1}<31:16>, 0), r_{src2}<15:0>) r_{dst}<15:0> ← min(max(r_{src1}<15:0>, 0), r_{src2}<15:0>)	2, dspalu	1,3

fabsval src1 → dst	floating point absolute value
	if (float)r_{src1} < 0 then
	r_{dst} ← –(float) r_{src1}
	else
	r_{dst} ← (float) r_{src1}
		3, falu	1,4

fabsvalflags src1 → dst	IEEE status flags from floating point absolute value r_{dst} ← ieee_flags(abs_val((float)r_{src1}))	3, falu	1,4
fadd src1 src2 → dst	floating point add r_{dst} ← (float) r_{src1} + (float) r_{src2}	3, falu	1,4
faddflags src1 src2 → dst	IEEE status flags from floating point add r_{dst} ← ieee_flags((float) r_{src1} + (float) r_{src2})	3, falu	1,4
fdiv src1 src2 → dst	floating point divide r_{dst} ← (float)r_{src1} / (float)r_{src2}	17, ftough	2, recovery=16
fdivflags src1 src2 → dst	IEEE status flags from floating point divide r_{dst} ← ieee_flags((float)r_{src1} / (float)r_{src2})	17, ftough	2, recovery=16

feql src1 src2 → dst	floating point compare equal
	if (float)r_{src1} = (float)r_{src2} then
	r_{dst} ← 1
	else
	r_{dst} ← 0
		1, fcomp	3

feqlflags src1 src2 → dst	IEEE status flags from floating point compare equal r_{dst} ← ieee_flags((float)r_{src1} = (float)r_{src2})	1, fcomp	3

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

fgeq src1 src2 → dst	floating point compare greater or equal
	if (float)r_{src1} ≥ (float)r_{src2} then
	  $r_{dst} \leftarrow 1$
	else
	  $r_{dst} \leftarrow 0$
		1, fcomp	3

fgeqflags src1 src2 → dst	IEEE status flags from floating point compare greater or equal $r_{dst} \leftarrow$ ieee_flags((float)r_{src1} ≥ (float)r_{src2})	1, fcomp	3

fgtr src1 src2 → dst	floating point compare greater
	if (float)r_{src1} > (float)r_{src2} then
	  $r_{dst} \leftarrow 1$
	else
	  $r_{dst} \leftarrow 0$
		1, fcomp	3

fgtrflags src1 src2 → dst	IEEE status flags from floating point compare greater $r_{dst} \leftarrow$ ieee_flags((float)r_{src1} > (float)r_{src2})	1, fcomp	3
fmul src1 src2 → dst	floating point multiply $r_{dst} \leftarrow$ (float)r_{src1} × (float)r_{src2}	3, ifmul	2,3
fmulflags src1 src2 → dst	IEEE status flags from floating point multiply $r_{dst} \leftarrow$ ieee_flags((float)r_{src1} × (float)r_{src2})	3, ifmul	2,3

fneq src1 src2 → dst	floating point compare not equal
	if (float)r_{src1} != (float)rsr_{src2} then
	  $r_{dst} \leftarrow 1$
	else
	  $r_{dst} \leftarrow 0$
		1, fcomp	3

fneqflags src1 src2 → dst	IEEE status flags from floating point compare not equal $r_{dst} \leftarrow$ ieee_flags((float)r_{src1} != (float)r_{src2})	1, fcomp	3

fsign src1 → dst	sign of floating point value
	if (float)r_{src1} = 0.0 then
	  $r_{dst} \leftarrow 0$
	else if (float)r_{src1} < 0.0 then
	  $r_{dst} \leftarrow$ 0xffffffff
	else
	  $r_{dst} \leftarrow 1$
		1, fcomp	3

fsignflags src1 → dst	IEEE status flags from floating point sign $r_{dst} \leftarrow$ ieee_flags(sign((float)r_{src1}))	1, fcomp	3
fsqrt src1 → dst	floating point square root $r_{dst} \leftarrow$ square_root(r_{src1})	17, ftough	2, recovery=16

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
fsqrtflags src1 → dst	IEEE status flags from floating point square root $r_{dst} \leftarrow$ ieee_flags(square_root((float)r_{src1}))	17, ftough	2, recovery=16
fsub src1 src2 → dst	floating point subtract $r_{dst} \leftarrow (float)r_{src1} - (float)r_{src2}$	3, falu	1,4
fsubflags src1 src2 → dst	IEEE status flags from floating point subtract $r_{dst} \leftarrow$ ieee_flags((float)r_{src1} – (float)r_{src2})	3, falu	1,4
funshift1 src1 src2 → dst	funnel-shift 1 byte $r_{dst}<31:8> \leftarrow r_{src1}<23:0>$ $r_{dst}<7:0> \leftarrow r_{src2}<31:24>$	1, shifter	1,2
funshift2 src1 src2 → dst	funnel-shift 2 bytes $r_{dst}<31:16> \leftarrow r_{src1}<15:0>$ $r_{dst}<15:0> \leftarrow r_{src2}<31:16>$	1, shifter	1,2
funshift3 src1 src2 → dst	funnel-shift 3 bytes $r_{dst}<31:24> \leftarrow r_{src1}<7:0>$ $r_{dst}<23:0> \leftarrow r_{src2}<31:8>$	1, shifter	1,2

h_dspiabs r0 src2 → dst	clipped signed absolute value
	if r_{src2} >= 0 then
	  $r_{dst} \leftarrow r_{src2}$
	else if r_{src2} = 0x80000000 then
	  $r_{dst} \leftarrow$ 0x7fffffff
	else
	  $r_{dst} \leftarrow -r_{src2}$
		2, dspalu	1,3

h_dspidualabs r0 src2 → dst	dual clipped absolute value of signed 16-bit half-words temp1 ← sign_ext16to32($r_{src2}<15:0>$) temp2 ← sign_ext16to32($r_{src2}<31:16>$) if temp1 = 0xffff8000 then temp1 ← 0x7fff if temp2 = 0xffff8000 then temp2 ← 0x7fff if temp1 < 0 then temp1 ← –temp1 if temp2 < 0 then temp2 ← –temp2 $r_{dst}<31:16> \leftarrow temp2<15:0>$ $r_{dst}<15:\ 0> \leftarrow temp1<15:0>$	2, dspalu	1,3

h_iabs r0 src2 → dst	hardware absolute value
	if r_{src2} < 0 then
	  $r_{dst} \leftarrow -r_{src2}$
	else
	  $r_{dst} \leftarrow r_{src2}$		

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

h_st16d(d) src1 src2	hardware 16-bit store with displacement
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 1
	else
	  bs ← 0
	$mem[r_{src2} + d + (1 \oplus bs)] \leftarrow r_{src1}$<7:0>
	$mem[r_{src2} + d + (0 \oplus bs)] \leftarrow r_{src1}$<15:8>
		3, dmem	4,5
			–128 to 126 by 2

h_st32d(d) src1 src2	hardware 32-bit store with displacement
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 3
	else
	  bs ← 0
	$mem[r_{src2} + d + (3 \oplus bs)] \leftarrow r_{src1}$< 7: 0>
	$mem[r_{src2} + d + (2 \oplus bs)] \leftarrow r_{src1}$<15: 8>
	$mem[r_{src2} + d + (1 \oplus bs)] \leftarrow r_{src1}$<24:16>
	$mem[r_{src2} + d + (0 \oplus bs)] \leftarrow r_{src1}$<31:24>
		3, dmem	4,5
			–256 to 252 by 4

h_st8d(d) src1 src2	hardware 8-bit store with displacement $mem[r_{src2} + d] \leftarrow r_{src1}$<7:0>	3, dmem	4,5 –64 to 63
hicycles → dst	read clock cycle counter, most-significant word $r_{dst} \leftarrow$ CCCOUNT<63:32>	1, fcomp	3
iadd src1 src2 → dst	signed add $r_{dst} \leftarrow r_{src1} + r_{src2}$	1, alu	1,2,3,4,5
iaddi(n) src1 → dst	add with immediate $r_{dst} \leftarrow r_{src1} + n$	1, alu	1,2,3,4,5 0 to 127
iavgonep src1 src2 → dst	signed average $r_{dst} \leftarrow (r_{src1} + r_{src2} + 1) >> 1;$	2, dspalu	1,3

ibytesel src1 src2 → dst	signed select byte
	if $r_{src2} = 0$ then
	  $r_{dst} \leftarrow$ sign_ext8to32(r_{src1}<7:0>)
	else if $r_{src2} = 1$ then
	  $r_{dst} \leftarrow$ sign_ext8to32(r_{src1}<15:8>)
	else if $r_{src2} = 2$ then
	  $r_{dst} \leftarrow$ sign_ext8to32(r_{src1}<23:16>)
	else if $r_{src2} = 3$ then
	  $r_{dst} \leftarrow$ sign_ext8to32(r_{src1}<31:24>)
		1,alu	1,2,3,4,5

iclipi src1 src2 → dst	clip signed to signed $r_{dst} \leftarrow \min(\max(r_{src1}, -r_{src2}-1), r_{src2})$	2, dspalu	1,3

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

iclr	invalidate all instruction cache blocks
	block ← 0
	for all blocks in instruction cache {
	  icache_reset_valid_block(block)
	  block ← block + 1
	}
		?, icache	

ieql src1 src2 → dst	signed compare equal
	if $r_{src1} = r_{src2}$ then
	  $r_{dst} \leftarrow 1$
	else
	  $r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5

ieqli(n) src1 → dst	signed compare equal with immediate
	if $r_{src1} = n$ then
	  $r_{dst} \leftarrow 1$
	else
	  $r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5
			–64 to 63

ifir16 src1 src2 → dst	sum of products of signed 16-bit halfwords
	$r_{dst} \leftarrow$ sign_ext16to32(r_{src1}<31:16>) ×
	  sign_ext16to32(r_{src2}<31:16>) +
	  sign_ext16to32(r_{src1}<15: 0>) ×
	  sign_ext16to32(r_{src2}<15: 0>)
		3, dspmul	2,3

ifir8ii src1 src2 → dst	signed sum of products of signed bytes
	$r_{dst} \leftarrow$ sign_ext8to32(r_{src1}<31:24>) ×
	  sign_ext8to32(r_{src2}<31:24>) +
	  sign_ext8to32(r_{src1}<23:16>) ×
	  sign_ext8to32(r_{src2}<23:16>) +
	  sign_ext8to32(r_{src1}<15: 8>) ×
	  sign_ext8to32(r_{src2}<15: 8>) +
	  sign_ext8to32(r_{src1}< 7: 0>) ×
	  sign_ext8to32(r_{src2}< 7: 0>)
		3, dspmul	2,3

ifir8ui src1 src2 → dst	signed sum of products of unsigned/signed bytes
	$r_{dst} \leftarrow$ zero_ext8to32(r_{src1}<31:24>) ×
	  sign_ext8to32(r_{src2}<31:24>) +
	  zero_ext8to32(r_{src1}<23:16>) ×
	  sign_ext8to32(r_{src2}<23:16>) +
	  zero_ext8to32(r_{src1}<15: 8>) ×
	  sign_ext8to32(r_{src2}<15: 8>) +
	  zero_ext8to32(r_{src1}< 7: 0>) ×
	  sign_ext8to32(r_{src2}< 7: 0>)
		3, dspmul	2,3

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
ifixieee src1 → dst	convert floating point to integer using PCSW rounding mode $r_{dst} \leftarrow$ (long) ((float)r_{src1})	3, falu	1,4
ifixieeeflags src1 → dst	IEEE status flags from converting floating point to integer using PCSW rounding mode $r_{dst} \leftarrow$ ieee_flags((long) ((float)r_{src1}))	3, falu	1,4
ifixrz src1 → dst	convert floating point to integer with round toward zero $r_{dst} \leftarrow$ (long) ((float)r_{src1})	3, falu	1,4
ifixrzflags src1 → dst	IEEE status flags from converting floating point to integer with round toward zero $r_{dst} \leftarrow$ ieee_flags((long) ((float)r_{src1}))	3, falu	1,4

iflip src1 src2 → dst	if non-zero negate
	if rsrc1 = 0 then
	$r_{dst} \leftarrow r_{src2}$
	else
	$r_{dst} \leftarrow -r_{src2}$
		2, dspalu	1,3

ifloat src1 → dst	convert signed integer to floating point $r_{dst} \leftarrow$ (float) ((long)r_{src1})	3, falu	1,4
ifloatflags src1 → dst	IEEE status flags from convert signed integer to floating point $r_{dst} \leftarrow$ ieee_flags((float)((long)r_{src1}))	3, falu	1,4
ifloatrz src1 → dst	convert signed integer to floating point with rounding toward zero $r_{dst} \leftarrow$ (float) ((long)r_{src1})	3, falu	1,4
ifloatrzflags src1 → dst	IEEE status flags from converting signed integer to floating point with rounding toward zero $r_{dst} \leftarrow$ ieee_flags((float)((long)r_{src1}))	3, falu	1,4

igeq src1 src2 → dst	signed compare greater or equal
	if rsrc1 ≥ rsrc2 then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5

igeqi(n) src1 → dst	signed compare greater or equal with immediate
	if rsrc1 ≥ n then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5
			–64 to 63

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	**Meaning**	**Latency, FU Type**	**Issue Slots, Modifier Range**

igtr src1 src2 → dst	signed compare greater
	if $r_{src1} > r_{src2}$ then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5

igtri(n) src1 → dst	signed compare greater with immediate
	if $r_{src1} > n$ then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5
			–64 to 63

iimm(n) → dst	signed immediate $r_{dst} \leftarrow n$	1, const	1,2,3,4,5 0x80000000 to 0x7fffffff

ijmpf src1 src2	interruptible indirect jump on false
	if (r_{src1} & 1) = 0 then {
	DPC ← r_{src2}
	if exception is pending then
	service exception
	else if interrupt is pending then
	service interrupts
	else
	PC, SPC ← r_{src2}
	}
		delay=3, branch	2,3,4

ijmpi(address)	interruptible jump immediate
	DPC ← address
	if exception is pending then
	service exception
	else if interrupt is pending then
	service interrupts
	else
	PC, SPC ← address
		delay=3, branch	2,3,4
			0 to 0xffffffff

ijmpt src1 src2	interruptible indirect jump on true
	if (r_{src1} & 1) = 1 then {
	DPC ← r_{src2}
	if exception is pending then
	service exception
	elseif interrupt is pending then
	service interrupts
	else
	PC, SPC ← r_{src2}
	}
		delay=3, branch	2,3,4

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

ild16d(d) src1 → dst	signed 16-bit load with displacement
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 1
	else
	  bs ← 0
	temp< 7:0> ← mem[(r_{src1} + d + (1 ⊕ bs)]
	temp<15:8> ← mem[(r_{src1} + d + (0 ⊕ bs)]
	r_{dst} ← sign_ext16to32(temp<15:0>)
		3, dmem	4,5
			–128 to 126 by 2

ild16r src1 src2 → dst	signed 16-bit load with index
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 1
	else
	  bs ← 0
	temp< 7:0> ← mem[(r_{src1} + r_{src2} + (1 ⊕ bs)]
	temp<15:8> ← mem[(r_{src1} + r_{src2} + (0 ⊕ bs)
	r_{dst} ← sign_ext16to32(temp<15:0>)
		3, dmem	4,5

ild16x src1 src2 → dst	signed 16-bot load with scaled index
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 1
	else
	  bs ← 0
	temp< 7:0> ← mem[(r_{src1} + (2 × r_{src2}) + (1 ⊕ bs)]
	temp<15:8> ← mem[(r_{src1} + (2 × r_{src2}) + (0 ⊕ bs)]
	r_{dst} ← sign_ext16to32(temp<15:0>)
		3, dmem	4,5

ild8d(d) src1 → dst	signed 8-bit load with displacement r_{dst} ← sign_ext8to32(mem[r_{src1} + d])	3, dmem	4,5 –64 to 63
ild8r src1 src2 → dst	signed 8-bit load with index r_{dst} ← sign_ext8to32(mem[r_{src1} + r_{src2}])	3, dmem, 4,5	4,5

ileqi(n) src1 → dst	signed compare less or equal with immediate
	if r_{src1} ≤ n then
	  r_{dst} ← 1
	else
	  r_{dst} ← 0
		1, alu	1,2,3,4,5
			–64 to 63

ilesi(n) src1 → dst	signed compare less with immediate
	if r_{src1} < n then
	  r_{dst} ← 1
	else
	  r_{dst} ← 0
		1, alu	1,2,3,4,5
			–64 to 63

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

imax src1 src2 → dst	signed maximum
	if $r_{src1} > r_{src2}$ then
	$r_{dst} \leftarrow r_{src1}$
	else
	$r_{dst} \leftarrow r_{src2}$
		2, dspalu	1,3

imin src1 src2 → dst	signed minimum
	if $r_{src1} > r_{src2}$ then
	$r_{dst} \leftarrow r_{src2}$
	else
	$r_{dst} \leftarrow r_{src1}$
		2, dspalu	1,3

imul src1 src2 → dst	signed multiply temp ← sign_ext32to64(r_{src1}) × sign_ext32to64(r_{src2}) r_{dst} ← temp<31:0>	3, ifmul	2,3
imulm src1 src2 → dst	signed multiply, return most-significant 32 bits temp ← sign_ext32to64(r_{src1}) × sign_ext32to64(r_{src2}) r_{dst} ← temp<63:32>	3, ifmul	2,3

ineq src1 src2 → dst	signed compare not equal
	if r_{src1} != r_{src2} then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5

ineqi(n) src1 → dst	signed compare not equal with immediate
	if r_{src1} != n then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5
			–64 to 63

inonzero src1 src2 → dst	if nonzero select zero
	if r_{src1} != 0 then
	$r_{dst} \leftarrow 0$
	else
	$r_{dst} \leftarrow r_{src2}$
		1, alu	1,2,3,4,5

isub src1 src2 → dst	subtract $r_{dst} \leftarrow r_{src1} - r_{src2}$	1, alu	1,2,3,4,5
isubi(n) src1 → dst	subtract with immediate $r_{dst} \leftarrow r_{src1} - n$	1, alu	1,2,3,4,5 0 to 127

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

izero src1 src2 → dst	if zero select zero
	if $r_{src1} = 0$ then
	  $r_{dst} \leftarrow 0$
	else
	  $r_{dst} \leftarrow$ rsrc2
		1, alu	1,2,3,4,5

jmpf src1 src2	indirect jump on false if $(r_{src1}$ & $1) = 0$ then PC $\leftarrow r_{src2}$	delay=3, branch	2,3,4
jmpi(address)	jump immediate PC ← *address*	delay=3, branch	2,3,4 0 to 0xffffffff
jmpt src1 src2	indirect jump on true if $(r_{src1}$ & $1) = 1$ then PC $\leftarrow r_{src2}$	delay=3, branch	2,3,4

ld32d(d) src1 → dst	32-bit load with displacement
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 3
	else
	  bs ← 0
	$r_{dst}<7:0> \leftarrow mem[r_{src1} + d + (3 \oplus bs)]$
	$r_{dst}<15:8> \leftarrow mem[r_{src1} + d + (2 \oplus bs)]$
	$r_{dst}<23:16> \leftarrow mem[r_{src1} + d + (1 \oplus bs)]$
	$r_{dst}<31:24> \leftarrow mem[r_{src1} + d + (0 \oplus bs)]$
		3, dmem	4,5
			–256 to 252 by 4

ld32r src1 src2 → dst	32-bit load with index
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 3
	else
	  bs ← 0
	$r_{dst}<7:0> \leftarrow mem[r_{src1} + r_{src2} + (3 \oplus bs)]$
	$r_{dst}<15:8> \leftarrow mem[r_{src1} + r_{src2} + (2 \oplus bs)]$
	$r_{dst}<23:16> \leftarrow mem[r_{src1} + r_{src2} + (1 \oplus bs)]$
	$r_{dst}<31:24> \leftarrow mem[r_{src1} + r_{src2} + (0 \oplus bs)]$
		3, dmem	4,5

ld32x src1 src2 → dst	32-bit load with scaled index
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 3
	else
	  bs ← 0
	$r_{dst}<7:0> \leftarrow mem[r_{src1} + (4 \times r_{src2}) + (3 \oplus bs)]$
	$r_{dst}<15:8> \leftarrow mem[r_{src1} + (4 \times r_{src2}) + (2 \oplus bs)]$
	$r_{dst}<23:16> \leftarrow mem[r_{src1} + (4 \times r_{src2}) + (1 \oplus bs)]$
	$r_{dst}<31:24> \leftarrow mem[r_{src1} + (4 \times r_{src2}) + (0 \oplus bs)]$
		3, dmem	4,5

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	**Meaning**	**Latency, FU Type**	**Issue Slots, Modifier Range**
lsr src1 src2 → dst	logical shift right $n \leftarrow r_{src2}<4:0>$ $r_{dst}<31:32-n> \leftarrow 0$ $r_{dst}<31-n:0> \leftarrow r_{src1}<31:n>$	1, shifter, 1,2	1,2
lsri(n) src1 → dst	logical shift right immediate $r_{dst}<31:32-n> \leftarrow 0$ $r_{dst}<31-n:0> \leftarrow r_{src1}<31:n>$	1, shifter	1,2 0 to 31
mergedual16lsb src1 src2 → dst	merge dual-16 least significant bytes $r_{dst}< 7: 0> \leftarrow r_{src2}< 7: 0>$ $r_{dst}<15: 8> \leftarrow r_{src2}<23:16>$ $r_{dst}<23:16> \leftarrow r_{src1}< 7: 0>$ $r_{dst}<31:24> \leftarrow r_{src1}<23:16>$	1, shifter	1,2
mergelsb src1 src2 → dst	merge least significant bytes $r_{dst}< 7: 0> \leftarrow r_{src2}< 7:0>$ $r_{dst}<15: 8> \leftarrow r_{src1}< 7:0>$ $r_{dst}<23:16> \leftarrow r_{src2}<15:8>$ $r_{dst}<31:24> \leftarrow r_{src1}<15:8>$	1,alu	1,2,3,4,5
mergemsb src1 src2 → dst	merge most significant bytes $r_{dst}< 7: 0> \leftarrow r_{src2}<23:15>$ $r_{dst}<15: 8> \leftarrow r_{src1}<23:15>$ $r_{dst}<23:16> \leftarrow r_{src2}<31:24>$ $r_{dst}<31:24> \leftarrow r_{src1}<31:24>$	1,alu	1,2,3,4,5
pack16lsb src1 src2 → dst	pack least significant 16-bit halfwords $r_{dst}<15: 0> \leftarrow r_{src2}<15:0>$ $r_{dst}<31:16> \leftarrow r_{src1}<15:0>$	1,alu	1,2,3,4,5
pack16msb src1 src2 → dst	pack most significant 16-bits $r_{dst}<15: 0> \leftarrow r_{src2}<31:16>$ $r_{dst}<31:16> \leftarrow r_{src1}<31:16>$	1,1lu	1,2,3,4,5
packbytes src1 src2 → dst	pack least significant bytes $r_{dst}< 7:0> \leftarrow r_{src2}<7:0>$ $r_{dst}<15:8> \leftarrow r_{src1}<7:0>$	1,1lu	1,2,3,4,5

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
quadavg src1 src2 → dst	unsigned byte-wise quad average temp ← (zero_ext8to32(r_{src1}<7:0>) + zero_ext8to32(r_{src2}<7:0>) + 1) / 2 r_{dst}<7:0> ← temp<7:0> temp ← (zero_ext8to32(r_{src1}<15:8>) + zero_ext8to32(r_{src2}<15:8>) + 1) / 2 r_{dst}<15:8> ← temp<7:0> temp ← (zero_ext8to32(r_{src1}<23:16>) + zero_ext8to32(r_{src2}<23:16>) + 1) / 2 r_{dst}<23:16> ← temp<7:0> temp ← (zero_ext8to32(r_{src1}<31:24>) + zero_ext8to32(r_{src2}<31:24>) + 1) / 2 r_{dst}<31:24> ← temp<7:0>	2, dspalu	1,3
quadumax src1 src2 → dst	unsigned byte-wise quad maximum r_{dst}< 7: 0> ← if r_{src1}<7:0> > r_{src2}<7:0> then r_{src1}<7:0> else r_{src2}<7:0> r_{dst}<15: 8> ← if r_{src1}<15:8> > r_{src2}<15:8> then r_{src1}<15:8> else r_{src2}<15:8> r_{dst}<23:16> ← if r_{src1}<23:16> > r_{src2}<23:16> then r_{src1}<23:16> else r_{src2}<23:16> r_{dst}<31:24> ← if r_{src1}<31:24> > r_{src2}<31:24> then r_{src1}<31:24> else r_{src2}<31:24>	2, dspalu	1,3
quadumin src1 src2 → dst	unsigned byte-wise quad minimum r_{dst}< 7: 0> ← if r_{src1}<7:0> < r_{src2}<7:0> then r_{src1}<7:0> else r_{src2}<7:0> r_{dst}<15: 8> ← if r_{src1}<15:8> < r_{src2}<15:8> then r_{src1}<15:8> else r_{src2}<15:8> r_{dst}<23:16> ← if r_{src1}<23:16> < r_{src2}<23:16> then r_{src1}<23:16> else r_{src2}<23:16> r_{dst}<31:24> ← if r_{src1}<31:24> < r_{src2}<31:24> then r_{src1}<31:24> else r_{src2}<31:24>	2, dspalu	1,3
quadumulmsb src1 src2 → dst	unsigned quad 8-bit multiply most significant temp ← zero_ext8to32(r_{src1}<7:0>) × zero_ext8to32(r_{src2}<7:0>) r_{dst}< 7: 0> ← temp<15: 8> temp ← zero_ext8to32(r_{src1}<15:8>) × zero_ext8to32(r_{src2}<15: 8>) r_{dst}<15: 8> ← temp<15:8> temp ← zero_ext8to32(r_{src1}<23:16>) × zero_ext8to32(r_{src2}<23:16>) r_{dst}<23:16> ← temp<15:8> temp ← zero_ext8to32(r_{src1}<31:24>) × zero_ext8to32(r_{src2}<31:24>) r_{dst}<31:24> ← temp<15:8>	3, dspmul	2,3

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
rdstatus(d) rsrc1 → rdest	read data cache status bits set_addr ← r_{src1} + d /* set_addr<10:6> selects set */ r_{dst}< 9: 0> ← dcache_LRU_set(set_addr) r_{dst}<17:10> ← dcache_dirty_set(set_addr) r_{dst}<31:17> ← 0	3, dmem-spec	5, –256 to 252 by 4
rdtag(d) rsrc1 → rdest	read data cache address tag block_addr ← r_{src1} + d /* block_addr<13:11> selects element, block_addr<10:6> selects set */ r_{dst}<20: 0> ← dcache_tag_block(block_addr) r_{dst}<31:21> ← 0	3, dmem-spec	5, –256 to 252 by 4
readdpc → dst	read destination program counter r_{dst} ← DPC	1, fcomp	3
readpcsw → dst	read program control and status word r_{dst} ← PCSW	1, fcomp	3
readspc → dst	read source program counter r_{dst} ← SPC	1, fcomp	3
rol src1 sc2 → dst	rotate left n ← r_{src2}<4:0> r_{dst}<31:n> ← r_{src1}<31–n:0> r_{dst}<n–1:0> ← r_{src1}<31:32–n>	1, shifter	1,2
roli(n) src1 → dst	rotate left by immediate r_{dst}<31:*n*> ← r_{src1}<31–*n*:0> r_{dst}<*n*–1:0> ← r_{src1}<31:32–*n*>	1, shifter	1,2 0 to 31
sex16 src1 → dst	signed extend 16 bits r_{dst} ← sign_ext16to32(r_{src1}<15:0>)	1,alu	1,2,3,4,5

ubytesel src1 src2 → dst	select unsigned byte
	if r_{src2} = 0 then
	  r_{dst} ← zero_ext8to32(r_{src1}< 7: 0>)
	else if rsrc2 = 1 then
	  r_{dst} ← zero_ext8to32(r_{src1}<15: 8>)
	else if rsrc2 = 2 then
	  r_{dst} ← zero_ext8to32(r_{src1}<23:15>)
	else if rsrc2 = 3 then
	  r_{dst} ← zero_ext8to32(r_{src1}<31:24>)
		1,alu	1,2,3,4,5

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
uclipi src1 src2 → dst	clip signed to unsigned $r_{dst} \leftarrow \min(\max(r_{src1}, 0), r_{src2})$	2, dspalu	1,3

uclipu src1 src2 → dst	clip unsigned to unsigned
	if $r_{src1} > r_{src2}$ then
	$r_{dst} \leftarrow r_{src2}$
	else
	$r_{dst} \leftarrow r_{src1}$
		2, dspalu	1,3

ueqli(n) src1 → dst	unsigned compare equal with immediate
	if $r_{src1} = n$ then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5
			0 to 127

ufir16 src1 src2 → dst	sum of products of unsigned 16-bit halfwords $r_{dst} \leftarrow$ zero_ext16to32(r_{src1}<31:16>) × zero_ext16to32(r_{src2}<31:16>) + zero_ext16to32(r_{src1}<15: 0>) × zero_ext16to32(r_{src2}<15: 0>)	3, dspmul	2,3
ufir8uu drc1 src2 → dst	unsigned sum of products of unsigned bytes $r_{dst} \leftarrow$ zero_ext8to32(r_{src1}<31:24>) × zero_ext8to32(r_{src2}<31:24>) + zero_ext8to32(r_{src1}<23:16>) × zero_ext8to32(r_{src2}<23:16>) + zero_ext8to32(r_{src1}<15: 8>) × zero_ext8to32(r_{src2}<15: 8>) + zero_ext8to32(r_{src1}< 7: 0>) × zero_ext8to32(r_{src2}< 7: 0>)	3, dspmul	2,3
ufixieee src1 → dst	convert floating point to unsigned integer using PCSW rounding mode $r_{dst} \leftarrow$ (unsigned long) ((float)r_{src1})	3, falu	1,4
ufixieeeflags src1 → dst	IEEE status flags from convert floating point to unsigned integer using PCSW rounding mode $r_{dst} \leftarrow$ ieee_flags((unsigned long) ((float)r_{src1}))	3, falu	1,4
ufixrz src1 → dst	Convert floating point to unsigned integer with round toward zero $r_{dst} \leftarrow$ (unsigned long) ((float)r_{src1})	3, falu	1,4
ufixrzflags src1 → dst	IEEE status flags from convert floating point to unsigned integer with round toward zero $r_{dst} \leftarrow$ ieee_flags((unsigned long) ((float)r_{src1}))	3, falu	1,4
ufloat src1 → dst	convert unsigned integer to floating point $r_{dst} \leftarrow$ (float) ((unsigned long)r_{src1})	3, falu	1,4

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
ufloatflags src1 → dst	IEEE status flags from convert unsigned integer to floating point r_{dst} ← ieee_flags((float) ((unsigned long)r_{src1}))	3, falu	1,4
ufloatrz src1 → dst	convert unsigned integer to floating point with rounding toward zero r_{dst} ← (float) ((unsigned long)r_{src1})	3, falu	1,4
ufloatrzflags src1 → dst	IEEE status flags from convert unsigned integer to floating point with rounding toward zero r_{dst} ← ieee_flags((float) ((unsigned long)r_{src1}))	3, falu	1,4

ugeq src1 src2 → dst	unsigned compare greater or equal
	if (unsigned)r_{src1} ≥ (unsigned)r_{src2} then
	  r_{dst} ← 1
	else
	  r_{dst} ← 0
		1, alu	1,2,3,4,5

ugeqi(n) src1 → dst	unsigned compare greater or equal with immediate
	if (unsigned)r_{src1} ≥ (unsigned)n then
	  r_{dst} ← 1
	else
	  r_{dst} ← 0
		1, alu	1,2,3,4,5
			0 to 127

ugtr src1 src2 → dst	unsigned compare greater
	if (unsigned)r_{src1} > (unsigned)r_{src2} then
	  r_{dst} ← 1
	else
	  r_{dst} ← 0
		1, alu	1,2,3,4,5

ugtri(n) src1 → dst	unsigned compare greater with immediate
	if (unsigned)r_{src1} > (unsigned)n then
	  r_{dst} ← 1
	else
	  r_{dst} ← 0
		1, alu	1,2,3,4,5
			0 to 127

uimm(n) → dst	unsigned immediate r_{dst} ← uimm(*n*)	1, const	1,2,3,4,5 0 to 0xffffffff

uld16d(d) src1 → dst	unsigned 16-bit load with displacement
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs ← 1
	else
	  bs ← 0
	temp< 7:0> ← mem[r_{src1} + d + (1 ⊕ bs)]
	temp<15:8> ← mem[r_{src1} + d + (0 ⊕ bs)]
	r_{dst} ← zero_ext16to32(temp<15:0>)
		3, dmem	4,5
			–128 to 126 by 2

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range

uld16r src1 src2 $\rightarrow$ dst	unsigned 16-bit load with index
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs $\leftarrow$ 1
	else
	  bs $\leftarrow$ 0
	temp< 7:0> $\leftarrow$ mem[r_{src1} + r_{src2} + (1 $\oplus$ bs)]
	temp<15:8> $\leftarrow$ mem[r_{src1} + r_{src2} + (0 $\oplus$ bs)]
	r_{dst} $\leftarrow$ zero_ext16to32(temp<15:0>)
		3, dmem	4,5

uld16x src1 src2 $\rightarrow$ dst	unsigned 16-bit load with scaled index
	if PCSW.bytesex = LITTLE_ENDIAN then
	  bs $\leftarrow$ 1
	else
	  bs $\leftarrow$ 0
	temp< 7:0> $\leftarrow$ mem[r_{src1} + (2 $\times$ r_{src2}) + (1 $\oplus$ bs)]
	temp<15:8> $\leftarrow$ mem[r_{src1} + (2 $\times$ r_{src2}) + (0 $\oplus$ bs)]
	r_{dst} $\leftarrow$ zero_ext16to32(temp<15:0>)
		3, dmem	4,5

uld8d(d) src1 $\rightarrow$ dst	unsigned 8-bit load with displacement r_{dst} $\leftarrow$ zero_ext8to32(mem[r_{src1} + *d*])	3, dmem	4,5 –64 to 63
uld8r src1 src2 $\rightarrow$ dst	unsigned 8-bit load with index r_{dst} $\leftarrow$ zero_ext8to32(mem[r_{src1} + r_{src2}])	3, dmem	4,5

uleqi(n) src1 $\rightarrow$ dst	unsigned compare less or equal with immediate
	if (unsigned)r_{src1} $\le$ (unsigned)n then
	  r_{dst} $\leftarrow$ 1
	else
	  r_{dst} $\leftarrow$ 0
		1, alu	1,2,3,4,5
			0 to 127

ulesi(n) src1 $\rightarrow$ dst	unsigned compare less with immediate
	if (unsigned)r_{src1} < (unsigned)n then
	  r_{dst} $\leftarrow$ 1
	else
	  r_{dst} $\leftarrow$ 0
		1, alu	1,2,3,4,5
			0 to 127

ume8ii src1 src2 $\rightarrow$ dst	unsigned sum of absolute values of signed 8-bit differences r_{dst} $\leftarrow$ abs_val(sign_ext8to32(r_{src1}<31:24>) – sign_ext8to32(r_{src2}<31:24>)) + abs_val(sign_ext8to32(r_{src1}<23:16>) – sign_ext8to32(r_{src2}<23:16>)) + abs_val(sign_ext8to32(r_{src1}<15: 8>) – sign_ext8to32(r_{src2}<15: 8>)) + abs_val(sign_ext8to32(r_{src1}< 7: 0>) – sign_ext8to32(r_{src2}< 7: 0>))	2, dspalu	1,3

 © 1999 Philips Semiconductors 10/08/99

Table 5 TriMedia Opcodes (Continued)

Syntax	Meaning	Latency, FU Type	Issue Slots, Modifier Range
ume8uu src1 src2 → dst	sum of absolute values of unsigned 8-bit differences $r_{dst} \leftarrow$ abs_val(zero_ext8to32(r_{src1}<31:24>) – zero_ext8to32(r_{src2}<31:24>)) + abs_val(zero_ext8to32(r_{src1}<23:16>) – zero_ext8to32(r_{src2}<23:16>)) + abs_val(zero_ext8to32(r_{src1}<15: 8>) – zero_ext8to32(r_{src2}<15: 8>)) + abs_val(zero_ext8to32(r_{src1}< 7: 0>) – zero_ext8to32(r_{src2}< 7: 0>))	2, dspalu	1,3
umul src1 src2 → dst	unsigned multiply temp ← zero_ext32to64(r_{src1}) × zero_ext32to64(r_{src2}) $r_{dst} \leftarrow$ temp<31:0>	3, ifmul	2,3
umulm src1 src2 → dst	unsigned multiply, return most significant 32 bits temp ← zero_ext32to64(r_{src1}) × zero_ext32to64(r_{src2}) $r_{dst} \leftarrow$ temp<63:32>	3, ifmul	2,3

uneqi(n) src1 → dst	unsigned compare not equal with immediate
	if (unsigned)r_{src1} != (unsigned)n then
	$r_{dst} \leftarrow 1$
	else
	$r_{dst} \leftarrow 0$
		1, alu	1,2,3,4,5
			0 to 127

writedpc src1	write destination program counter DPC ← r_{src1}	1, fcomp	3
writepcsw src1 src2	write program control and status word PCSW ← (PCSW & ~r_{src2}) | (r_{src1} & r_{src2})	1, fcomp	3
writespc src1	write source program counter SPC ← r_{src1}	1, fcomp	3

© 1999 Philips Semiconductors 10/08/99

Chapter 11

Linking TriMedia Object Modules

Topic	Page
Introduction	68
Overview	68
Object File Contents	73
Static Linking	80
Dynamic Linking	82
Section Renaming	113
Link Optimizations	116
Multiprocessor Support	117
SDRAM Memory Images vs Load Images	119
Constructing Load Images Using tmld	122
Download Symbols	123
tmld Options	126
List Construction by tmld	130

Introduction

TriMedia executables typically consist of a mix of user code, runtime support, and several user- and system-provided libraries. Using the compiler tools **tmccom**, **tmsched** and **tmas**, these are all separately compiled into *object files*, which are finally merged into executables using the linker, **tmld**.

Separate compilation has a number of obvious advantages over compiling applications entirely from source. First, many of the software components involved have been independently developed, and for a variety of reasons the authors might not want to publish their sources. Second, separate compilation simply saves time when earlier results are reused. This reuse of results can be at the level of applications, where (precompiled) software libraries can be shared between applications, or at the development level in recompiling a single application, where intermediate compilation results can be left untouched as long as their sources have not been changed.

Therefore, central to the process of compilation and application building, the *object file* can be seen as an intermediate representation by which software can be distributed, and from which executables can be formed by linking.

Overview

This chapter describes the conceptual structure of object files and the tools with which they can be manipulated. It does not contain a description of the *physical* structure of the object files, as users are shielded from the physical object format details by the various TriMedia tools and libraries.

Object Files

Object files will appear in a number of variations that are all instances of the TriMedia object format. Initially created by the assembler **tmas** as straightforward translations of assembly files,"*dot o*" files (*.o*), they can be linked with other object files into larger object files, or into *executables* or *dynamic libraries*. (Refer to below.) Executables are fully linked, self-contained object files that can be loaded onto a processor and subsequently started. Dynamic libraries are fragments into which executables can be divided, and which can be independently loaded and unloaded *after* the executable has started.

 ©1999 Philips Semiconductors 10/08/99

Such “fragments,” with executables themselves as special cases, are commonly referred to as *code segments*.

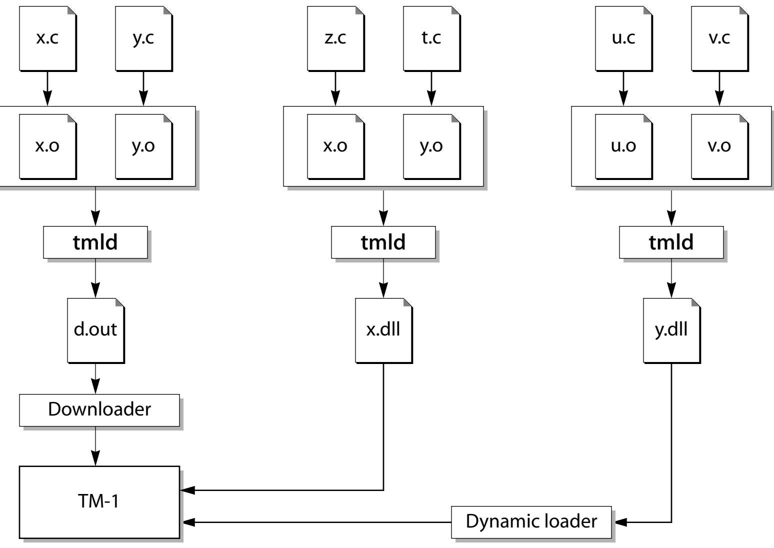


Figure 1 Dynamic Libraries

The initial loading of an executable onto a processor before its execution is often called *downloading* (in contrast to the loading of a dynamic library during execution, which is referred to as *dynamic loading*). Downloading is generally performed by a monitor program like **tmmon** or **tmrun**, while dynamic loading is performed by a library that has been linked with the first of the executable’s code segments (that is, the segment initially downloaded). Executables are usually downloaded to the start of SDRAM of the processor on which they must be executed. Memory for dynamically loading a code segment is allocated on the heap by the dynamic loader or by a user-provided memory manager.

Object File Structure

An object file contains the units from which any executable is eventually constructed: binary instruction sequences and initialized or uninitialized global data items. The instruction sequences are translations of decision trees (dtrees) at the trees/assembly level, and the data items almost always correspond with global variables at the C level. This is illustrated in , which shows a sample C program, its intermediate trees representation, and its object file. As opposed to an assembly file, which is a more or less readable ASCI file, an object file is a binary representation which is fit for efficient handling of the last stages of program manipulation: *linking*, *relocation,* and *loading* to a TriMedia processor for execution.

fib.c

```
int fibtab[100];
external void print_new();

int fib( int i ){
   if( i<=2 ){
      return 1;
   } else if (fibtab[i] > 0) {
      return fibtab[i];
   } else {
      fibtab[i]= fib(i-1) + fib(i-2);
      print_new( i, fibtab[i] );
      return fibtab[i];
   }
}
```

fib.t

```
{_fib:}
entree (0)
   1 rdreg (4);
   ...
   if 16 (0.5) then
      ...
   else (16)
      26 uimm (_fibtab);
      ...
      if 28 (0.5) then
         ...
         cgoto 13
      else (28)
         ...
         37 uimm (__fib_DT_1);
         ...
         gotree {_fib}
      end (28)
   end (16)
endtree
```

```
{.common _fibtab,400,"bss",4}
```

```
{__fib_DT_3:}
tree (0)
   ...
   4 uimm (_fibtab);
   ...
   cgoto 7
endtree
```

```
{__fib_DT_2:}
tree (0)
   ...
   7 uimm (_fibtab);
   9 uimm (__fib_DT_3);
   ...
   gotree _print_new
endtree
```

```
{__fib_DT_1:}
tree (0)
         ...
   6 uimm (__fib_DT_2);
         ...
   gotree {_fib}
endtree
```

fib.o

Figure 2 C module with corresponding decision trees and resulting object file

 ©1999 Philips Semiconductors 10/08/99

The essence of an object file is that it is a representation that contains instruction- and data units in binary forms that are practically identical to their eventual placement into SDRAM, but without any assumptions about actual load addresses or the relative loading order of the instruction and data units. This frees the linker and loaders to reorder these program units, or even to delete several of them during specific link time optimizations, and it frees the loaders to accept any desired memory address for loading. In other words, while it has been the responsibility of the compilation tools **tmccom**, **tmsched** and **tmas** to generate efficient instruction sequences, it is the responsibility of **tmld** and the loaders to map them efficiently to SDRAM without further interpretation of their contents.

This address independence has a price: a considerable part of the object file consists of descriptors and administration needed for *relocation* of the executable or dynamic library.

Relocation involves "fixing up" the loaded executable or dynamic library for the actual address in memory where it is loaded. The descriptors and administration needed for relocation are not used during execution, and hence are discarded after loading.

Relocation is traditionally performed by the linker (especially in virtual memory based systems like UNIX, which always load their executables at a fixed address in virtual memory address space). In contrast to this, the 32-bit TriMedia architecture does not include a memory management unit: the download address is a *physical* address that depends on the position in PCI space of the SDRAM of the processor that has been selected for execution. Because this address is generally unknown at link time, the TriMedia SDE postpones relocation to the loaders.

Load addresses of executables are generally unknown when such executables need to be run on a variety of TriMedia boards (each board has its own SDRAM base address that in turn depends on the specific PC plus PCI slot in which the board has been mounted). However, a simpler download procedure is possible where one single load address will be used, for instance in the case of an executable that is intended for a specific stand-alone TriMedia board: relocation can then still be performed during linking, by specifying the anticipated download address to **tmld**. This will result in a *load image*, written to a file. A load image is a relocated binary representation of an executable that can, for example, be stored in EEPROM and that can simply be copied to SDRAM for execution. A load image is no longer an object file. It cannot be relocated again and cannot be executed from a load address that is different from the anticipated one. Note that constructing a load image is also an internal phase during downloading.

In addition to the simple merging of the program units (contained by the linked input files) into the resulting output file, linking also involves *symbol resolution*. This process is necessitated by the fact that program units may *refer* to each other. Each dtree and global variable has a name which may be used as a placeholder of the yet-unknown SDRAM address where the program unit will eventually be loaded. Program unit names are also referred to as *symbols*. Examples of usage of such symbols are in instruction sequences, where the addresses might be used as branch targets or for fetching from- or storing to

memory; in a similar way, variables might be initialized with addresses of program units. Example of references are shown in , where dtree _**fib** loads the address of variable *_fibtab* in order to fetch from it and store to it, and it performs a recursive call to itself after having saved the address of dtree *__fibtab_DT_1* as return address.

Descriptions of all references in an object file are stored as part of the relocation information of that object file. Until a load image is constructed, references must be kept in a symbolic form. This is because the final address values are not yet known prior to relocation, and because object files that have not yet been fully linked might refer to program units in other object files (a situation that occurs with the use of an external function or variable in the C program from which the object file was compiled). Because no information other than the name of the referenced external program unit is stored in the object file, such references are referred to as *unresolved references*. Similarly, the names in unresolved references are known as *unresolved symbols*. It is the responsibility of **tmld** to match unresolved references to their similarly named program units while linking a number of input files. This process, called *symbol resolution*, is illustrated in Figure 3.

Note

Relocation is nothing more than a traversal of all references of an executable or code segment after the load address has been established, while placing the binary address of the referred program unit into the appropriate position in the image of the referring program unit.

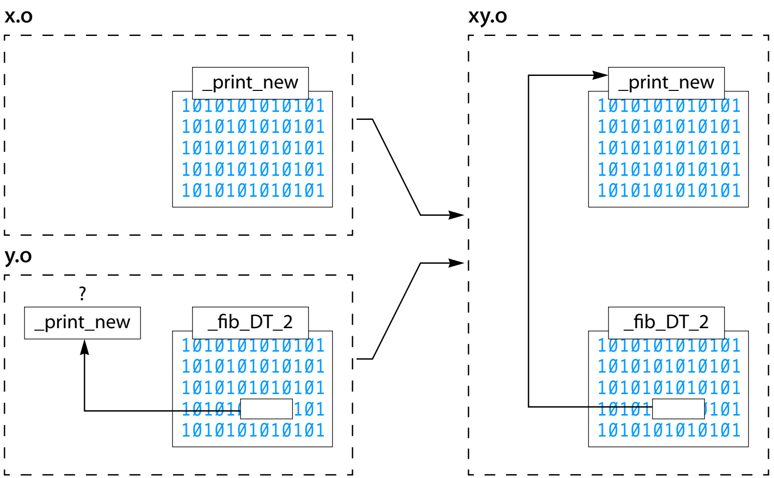


Figure 3 Linking = object file merging + symbol/reference resolution

Executables and dynamic libraries are *fully linked*, which means that they contain no unresolved references. For a simple executable, this means that all instructions and data units are present in the object itself. Generally, in the case of dynamic linking this means that a referred program unit is either present or that it has been recorded (in the referring

©1999 Philips Semiconductors 10/08/99

code segment), so that the dynamic loader can be invoked at runtime when the program unit is needed. This recording is automatically performed by the linker.

Object Manipulation Tools

The following table lists the tools and libraries involved in object file manipulation.

Tools and Libraries	Description
tmas	Creates object file from assembly input.
tmld	Links object files, creates executables and dynamic libraries, maps instruction sequences and global data into sections, renames sections, assigns caching- or shared memory properties to sections, exports symbols for dynamic linking, defines download symbols, generates load images.
tmar	Maintains archives of object files.
tmnm	Lists symbols and symbol properties in object files.
tmsize	Lists sections and section sizes in object files.
tmstrip	Removes any administration that is not strictly needed for further object manipulation.
tmdump	Dumps object file contents to ASCII.
tmmon, tmgmon, tmrun, tmmprun, (tmsim)	Monitors and execution shells, for loading and starting an executable object on one or more TriMedia processors.
downloader library	Relocates executable object, and constructs load image. This library is used for this purpose by the tools **tmmon**, **tmgmon**, **tmrun**, **tmmprun**, **tmsim**, and **tmld** and is also available as a library libload.a for a variety of platforms.
dynamic loader	Relocates dynamic library, adds load image to running TriMedia program, and links all inter-code segment references. The dynamic loader is available as a TriMedia library, to be part of applications which use dynamic loading services.

Object File Contents

As described in the overview, an object file consists of *program units*, *symbols*, and *references*. Program units are images of instruction sequences and global data that must eventually be concatenated to form the load image. Symbols are names of program units. References represent positions within the program unit images where the currently unknown address values of other program units will have to be filled in.

This section describes object file contents in more detail. , a sample object file generated using **tmdump**, can be used as a running illustration throughout this section.

```
section  size  alignment  has_data  is_code  is_shared  is_read_only  caching

text     281   64         True      True     False      True          Cached
data1    0     1          True      False    False      True          Cached
data     0     1          True      False    False      False         Cached
bss      0     1          False     False    False      False         Cached

section units

section        offset        length      alignment
text           0x00000000       156      1
text           0x0000009c        34      1
text           0x000000be        40      1
text           0x000000e6        51      1

symbols

name           scope           type                attr
_fib_DT_3      LocalScope      RelativeSymbol      text, 0x000000e6
_fib_DT_2      LocalScope      RelativeSymbol      text, 0x000000be
_fib_DT_1      LocalScope      RelativeSymbol      text, 0x0000009c
_fib_DT_0      LocalScope      RelativeSymbol      text, 0x00000000
_fib           GlobalScope     RelativeSymbol      text, 0x00000000
_fibtab        GlobalScope     CommonSymbol        4,400
_print_new     GlobalScope     UnresolvedSymbol    False

marker_references

dest_offset    dest_section    src_offset          src_section
0x00000045     text            0x00000000          text
0x00000048     text            0x0000009c          text
0x000000a1     text            0x00000000          text
0x000000a4     text            0x000000be          text
0x000000cd     text            0x000000e6          text

symbol references

dest_offset    dest_section    src_offset          src_symbol
0x0000000c     text            0x00000000          _fibtab
0x000000c3     text            0x00000000          _print_new
0x000000c6     text            0x00000000          _fibtab
0x000000eb     text            0x00000000          _fibtab
```

Figure 4 Sample Output of **tmdump**

Sections

Many compilers with a Unix tradition separate the generated instruction and global data units into the standard sections *text*, *data1*, *data* and *bss,* according to some general properties. Such compilers perform in this manner so that the following actions can be quickly achieved:

- Distinguish between code and data

- Set write protection hardware for read only data after loading
- Set data sharing between different (virtual) address spaces
- Allocate and initialize memory space for uninitialized data
- Map certain data into a specific memory range
- Set cache locking or cache transparency for all data for which this is required

A specific advantage of section grouping in computers with separated data- and instruction caches, is that instructions and data are kept apart so that instruction caches are not contaminated with data and vice versa.

TriMedia objects support sections for similar reasons. As described on page 114, **tmccom** maps all program units into sections initially named *text*, *data1*, *data* or *bss*. Linking of object files will result in successively larger sections. Sections can be renamed using **tmld** to create new ones, can be assigned specific properties, and can be broken down again into their individual section units (for example, for linker optimization such as dead code removal and identical code folding). Sections contained by an object file, with their sizes, can be displayed using **tmsize**.

Example

shows a use of **tmsize** in an example which monitors section sizes after section renaming and linking. In this example, a global table maintained by source file print_new.c is placed into uncached data memory because this table is accessed from another processor. (For instance, the PC in which this application's TriMedia board has been mounted.) Because TriMedia cache coherence is the responsibility of the application, measures must be taken to ensure that table data written by the program does not pend in the data cache but actually gets written to memory, and that new table data written by the other processor does not get masked by stale cache contents. A simple solution is to avoid use of the cache altogether by placing the table into uncached memory.

Note

It is assumed that this global table is the only global variable in print_new.c, so that renaming the *data* section affects only this table.

```
tmcc -c print_new.c fib.c

tmsize print_new.o fib.o
text    data    data1   bss     dec        hex        name
428     1600    0       0       2028       0x7EC      print_new.o
281     0       0       0       281        0x119      fib.o
709     1600    0       0       2309       0x905      Total

tmld print_new.o \
   -sectionrename       data=table_section \
   -sectionproperty data=uncached          \
   -o print_new.o

tmsize print_new.o fib.o
text   data    data1   bss   table_section  dec      hex      name
428    -       0       0     1600           2028     0x7EC    print_new.o
```

```
281    0       0       0     -              281      0x119    fib.o
709    0       0       0     1600           2309     0x905    Total

tmld print_new.o fib.o -o output.o

tmsize output.o
text   data    data1   bss   table_section  dec      hex
709    0       0       0     1600           2309     0x905
```

Figure 5 Section renaming

Sizes reported by **tmsize** may differ slightly from their size *contributions* to the final linked executables. These differences may be due to linker optimizations (which decrease contributed size), or to the introduction of internal padding by the linker to enforce alignments (which increases contributed size). As a general rule, text size increases by as much as 10% when linking an executable. Link time optimizations typically produce a total size reduction of 20-30%.

After compiling the two files into object files, the section containing this table, *data*, is renamed using **tmld** to *table_section*, and given the property *uncached*. The effect of this is checked by using **tmsize**, which shows that the data section in print_new.o has indeed been moved (the new section property itself can only be shown by using **tmdump**).

Linking the two object files into a new one by using **tmld** shows that this new section indeed is kept separate during further linking. It will eventually result in a separate section in the executable which might be finally linked from the result output.o. Upon loading this executable, the TriMedia downloader will automatically place this section into uncached memory.

Program Unit Attributes

The following table lists the possible attributes of program units (that is, external variables, and functions). Because program units are always embedded in sections, they are also known as *section units* (this is how **tmdump** names them). All of these attributes are constructed from the assembly input:

section	Name of the section in which the program unit has been embedded.
offset	Position of the start of the program unit in its section, relative to the start of the section.
length	Total size of the program unit in bytes.
alignment	Required program unit alignment in bytes. This attribute is taken from the assembly source. Note that the offset value must be a multiple of the alignment value.

©1999 Philips Semiconductors 10/08/99

Section Attributes

The following table lists the possible section attributes. Most section properties are interpreted by the loader library when downloading an executable. Several section attributes can be set by the user by using **tmld**:

name	Name of the section. This name is initially chosen by **tmccom**, but can be changed by **tmld** using option **-sectionrename**. The purpose of section renaming is to separate particular code or data in further linking (for instance because specific section properties need to be assigned).
size	Total size of section in bytes, including padding caused by program unit alignment.
alignment	Required section alignment in bytes, which is a multiple (or usually just equal to) the least common multiple of the alignments of all program units that it contains. This attribute is interpreted by the TriMedia downloader, and is automatically set by **tmld**.
has_data	This attribute specifies whether the section has initial contents. If this is the case (*has_data* == *True*), then these initial contents occupy space in the object file; otherwise no space is reserved in the object file. Sections which do not "have" data are always renamed instances of *bss* sections created by **tmccom**. This attribute is interpreted by the TriMedia downloader, and is automatically set by **tmas** (which initially creates object files). The downloader reserves space for sections which do not "have" data, and sets their contents to zero.
is_code	When this attribute is equal to *True*, this section will be assumed to contain instructions. This attribute is interpreted by the TriMedia downloader and by **tmld**. It is automatically set by **tmas** (which initially creates object files).
is_shared	When this attribute is equal to *True*, this section will be shared between all the executables on a multi-TriMedia configuration. This attribute is interpreted by the TriMedia downloader, and can be set by **tmld** using the option **-sectionproperty**.
is_read_only	When this attribute is equal to *True*, this section will be assumed to contain non-modifiable data. This attribute is currently not used by the TriMedia downloader, but its value has an effect during the linker optimization as triggered by option **-foldcode**: members of a class of identical and read-only program units are candidates for removal, with each use substituted by a use of one representative of the class. It can be set by **tmld** using option **-sectionproperty**.
caching_property	This attribute has a number of values which are described below. It defines whether (and how) the section contents should be cached. The value of this attribute is interpreted by the TriMedia downloader, and can be set by **tmld** using option **-sectionproperty,** using either of the following arguments: Cached — Section contents will be subject to normal caching.

Uncached	Not allowed for instruction sections (**is_code == True**), due to hardware restrictions of 32-bit TriMedia processors. Uncached (data) sections will be placed into uncached memory by the downloader library.
CacheLocked	Cachelocked. Allowed for instruction and data sections. Cachelocked sections will be placed into cachelocked memory by the downloader library.

More information on section renaming and assigning properties can be found on pages 113, and 117.

System Sections and Sections Introduced by tmld

Sections are not only used for storing program units in an object file, but also for storing debugger information and the object file administration itself. This is partly because a section appears to be a general mechanism within an object file for storing information. More fundamentally in the context of dynamic loading, under specific conditions, certain object file administration such as reference data must also be loaded for use onto TriMedia, and hence must be treated similar to “normal” object file data. Therefore, no strict separation can be made between program units and object file administration and all must be given a similar treatment. Since these additional sections are all fully handled by the object manipulation tools, the user does not need to be concerned with them. However, **tmsize** shows all sections that will actually be downloaded, and therefore sections introduced by the compiler tools sometimes show up in its output. Sections that might be introduced by the compiler tools and **tmld** are described on page 115.

```
tmcc -btype dynboot main.c

tmsize a.out
text   data data1 bss   $String$DynLoad$ $Ext_Mod$DynLoad$ _mdescs_ _mdesc_ dec hex
153472 2152 1672  19628  32               2418              98       179496  0x2BD28
```

Figure 6 Special sections showing up in tmsize output

Symbols

Symbols are entities that have yet-unknown 32-bit values associated with them, which can be used as placeholders for those values by program units. Although these values are generally addresses, they may represent any information that can be encoded in a 32-bit value.

The following table lists the possible symbol attributes. Most of these attributes are taken directly from the assembly input, but some of them can be set by using **tmld**:

name	Name of the symbol
type	Symbol type:

 ©1999 Philips Semiconductors 10/08/99

AbsoluteSymbol A symbol of this type has a fixed 32-bit value associated; this value is stored in symbol attribute, *attr.* Absolute symbols are currently not used in the TriMedia SDE.

RelativeSymbol A symbol of this type has an address within an associated program unit. This is often just the address of the start of the program unit, but hand coding assembly might cause relative addresses strictly within program units. The relative value is encoded in *attr*, as a *(section, offset within that section)* pair. The value represented by a relative symbol becomes known as soon as a load address for *section* has been established, and is the sum of that load address and *offset*. A *(section, offset)* pair is also called a *marker.* (See *References* on page 80.)

CommonSymbol A symbol of this type represents a *tentative* uninitialized data definition with a size and alignment specified in *attr*, often resulting from global variable definitions in C without initializer. Refer to ANSI C Semantics. If an object file containing a common symbol is linked to another object file containing a relative or absolute symbol with the same name, the common symbol is merged with the absolute symbol: the common symbol is discarded and its size and alignment are ignored. Common symbols that are still present while linking an executable or code segment are converted to program units into the *bss* section. See the description on page 113.

UnresolvedSymbol Unresolved symbol. *Attr* contains a boolean value which is usually *False*, but is set to *True* for symbols that are defined using **tmld,** via option **-bdownload,** to be *download* symbols. Download symbols are the only type of unresolved symbols that are allowed to remain undefined in executables and other code segments. They are intended for passing information that becomes available at downloading time. More information on download symbols can be found on page 119.

attr *Representation of symbol value; see above.*

scope Visibility of symbol in static or dynamic linking:

LocalScope Symbols of this type generally correspond to static variables or static functions at the C level, and serve only informational purposes in the object file. Their values can be inspected using **tmnm** and **tmdump**, but they are not used for further linking. This scope value is taken from the C or assembly source.

GlobalScope Symbols of this type generally correspond to external variables or external functions at the C level. They are “visible” from within other object files in static linking. When an object file containing a specific global symbol is linked to another object file containing an unresolved reference to an identically named unresolved symbol, then the unresolved reference will be resolved. Also, errors will occur when two object files are linked that contain identically named global symbols. This scope value is taken from the C or assembly source.

DynamicScope Symbols of this type will be exported by the current code segment, and hence are visible from within other code segments. This scope value must be set using **tmld** while constructing dynamic library via option **-bexport**.

A list of symbols, with their scopes, types and attributes, can be shown by **tmnm**. (See Figure 7).

```
tmnm -an fib.o
         U  _print_new
         C  _fibtab
00000000 t  __fib_DT_0
00000000 T  _fib
0000009C t  __fib_DT_1
000000BE t  __fib_DT_2
000000E6 t  __fib_DT_3
```

Figure 7 Sample output of tmnm

References

References are descriptions of values to be filled in at a specific location within a program unit, and hence consist of two parts: a *destination of the value,* pointing into a program unit, and a *description of the value* itself, describing a 32-bit number. Destinations are always represented by means of a marker (see description of relative symbols on page 78), and referred values are always initially represented by a pair *(symbol+offset)*. The offset allows the representing of addresses within the program unit named by the symbol.

These *reference representations via symbols* are only needed for unresolved references. References resolved to relative symbols *(sym_section,sym_offset)* can be more directly represented by replacing the (*symbol,offset)* pair by a marker *(sym_section,sym_offset+offset)*. This has the advantage that the symbol is no longer needed and may be stripped from the object file when desired. References to absolute symbols can even be removed entirely by the linker after filling in the absolute value at the appropriate place. *Symbol references* and *marker references* are shown by **tmdump**, as in Figure 4 on page 74.

Static Linking

tmld links a number of object files, performs operations (such as section renaming and assigning section and symbol properties), and produces an output file. The following types of files are allowed inputs to the linker:

- dynamic libraries
- plain object files (that is, object files that are not code segments)
- archive files constructed from plain object files using **tmar**.

 ©1999 Philips Semiconductors 10/08/99

The output file is either an object file or a load image file, and in the case of an object file, is either again a plain object file (default) or a code segment, depending on the use of **tmld** options **-exec** and **-btype**.

Linking executables requires specification of a start symbol. Memory image generation can only be performed while linking an executable, but requires some additional information that is usually only available during downloading: MMIO base address, processor frequency, start and end of SDRAM area in which the image will be loaded, as well as some optional multiprocessor setup information.

The input files are basically linked in the order in which they are specified on the link command line. Generally this is only important for the location of the start address of a **boot** executable (That is, the most usual type of executable, which is used for loading and execution (up)on an idle TriMedia processor). Such an executable should have its transfer address at the start of its text segment, and therefore the link command for constructing such an executable should specify the object file that contains the start code as first input file. This file usually is the file *reset.o* from the TCS system directory. Linker optimizations might considerably reorganize the original order of object files, and even of their contents. However, the location of the first dtree in the *text* segment will always be preserved.

The input files are linked in order of their occurrence on the command line. Plain object files are included into the current result of linking, while resolving all references and symbols that can be resolved, and while generating errors for all duplicate symbols. Recall that resolving references involves the matching of unresolved references with corresponding symbol definitions, and that resolving symbols involves merging previously unresolved symbols and common symbols with corresponding relative- or absolute symbol definitions. Only symbols with a *non-local* scope can be used for resolution, and only symbols with a *non-local* scope are able to cause duplicate symbol errors).

When linking an archive file, **tmld** repeatedly fetches and links plain object files that define currently undefined symbols in the current result of linking. This repetition stops when the contents of the archive file are no longer able to resolve undefined symbols. Archives are not inspected again. Therefore, an unresolved reference that has a corresponding definition in one of the archive's object files might still remain unresolved if it were introduced by a "later" object file. This depends on whether the defining object file has already been fetched from the archive.

Linking a dynamic library is similar to linking an object, with the following differences:

- Unless it is linked as *embedded*, its contents are not included into the current result of linking. Rather, directives for loading it at runtime are generated.
- Only symbols with *dynamic* scope are used for resolving yet unresolved references, or for generating duplicate symbol errors.

Dynamic Linking

Dynamic linking / dynamic loading is the loading and unloading of executable code by a running application. It allows the developer to put aside many of the concerns of efficient physical construction of software programs.

The TCS supports dynamic linking with special options to the **tmcc** compiler driver, which passes them to the **tmld** linker. Dynamic linking is supported by the cycle-accurate **tmsim** simulator.

This section explains what dynamic linking is and how it is implemented on the TCS.

Why Dynamic Linking is Valuable

The classic model of program compilation and execution uses *static linking* to build an executable program. With static linking, the linker takes a number of object files, resolves all cross-references to variables and functions, and builds a merged, self-contained executable program that contains executable code for each referenced function and global data space for each referenced global variable. Such an executable can be downloaded to TriMedia and executed without the need for, and often even without the possibility of, adding executable code during runtime.

In contrast, *dynamic linking* enables system designers to postpone building an executable until runtime by including a dynamic equivalent of the static linker/loader in the TCS runtime support library. By using specific options, the static linker **tmld** can be instructed to suppress copying one or more of the input files into the produced executable. Instead, it will generate directives to the dynamic loader for loading these omitted object files at execution time, and for binding ('linking') all symbol cross references after such dynamic loading. The terms *dynamic loading* and *dynamic linking* are often used interchangeably to refer to different aspects of the same process: getting new code into a running system.

Using dynamic linking provides several new opportunities:

- Dynamic linking generally reduces disk space, because common libraries such as the standard C library and boot code need not be included in every executable object file. Instead, such libraries can be stored once, and loaded whenever they are needed.
- Dynamic linking might reduce the required amount of runtime memory in multiprocessing systems, where different applications can be loaded and executed in parallel: dynamic libraries can be shared, and hence need not be loaded for each and every application that uses them.
- Additional runtime memory might be saved by the ability of loading code only when it is needed, and by the ability of unloading code, as soon as it is no longer needed. Memory released by unloaded code can be reused.
- With certain restrictions, dynamic libraries can be updated without the need for relinking the applications that use them. This allows for upgrading already released

 ©1999 Philips Semiconductors 10/08/99

applications, by distributing new instances of libraries containing bug fixes or other improvements.

- Dynamic libraries can be used for hiding architectural details. Dynamic libraries are similar to static libraries, but allow software to run on different platforms even without relinking. This is illustrated by the TriMedia board support library, *libboard*: a dynamic loader based application can be executed on a new type of TriMedia board, simply by providing a *libboard.dll* that matches this new board.

Concepts of Dynamic Loading

The following sections describe the concepts that play a role in TriMedia dynamic linking and loading. Because these sections present an overview in somewhat abstract terms, they can best be read with a general reference in mind to the following sections: *Simple Examples* on page 87, and *More Examples* on page 110.

Difference Between Static- and Dynamic Linking

When used in static mode, **tmld** produces an *executable* from a number of object files. For this, **tmld** performs the following basic actions: it *merges* the contents of the object files, it *resolves* all references that it encounters and it *binds* them. After that, the executable can be *loaded* by the downloader, for which this downloader has to perform *address mapping* (in the SDRAM range of the used TriMedia processor) and *relocation*. The distinction between *resolving* references and *binding* them is usually not explicitly made, because they can be combined by the linker during static linking. However, in the context of dynamic linking their difference is important: *resolving* a reference is the process of locating a definition of a referenced memory object (“knowing where it is”), and *binding* a memory object is the process of creating some physical link to that object (“making it accessible”).

For dynamic linking, **tmld** goes no further than producing *code segments* (see), which are the units of code loading onto a TriMedia processor. Different types of code segments

exist, and are described in the next section, but during this overview code segments can be thought of as dynamic libraries.

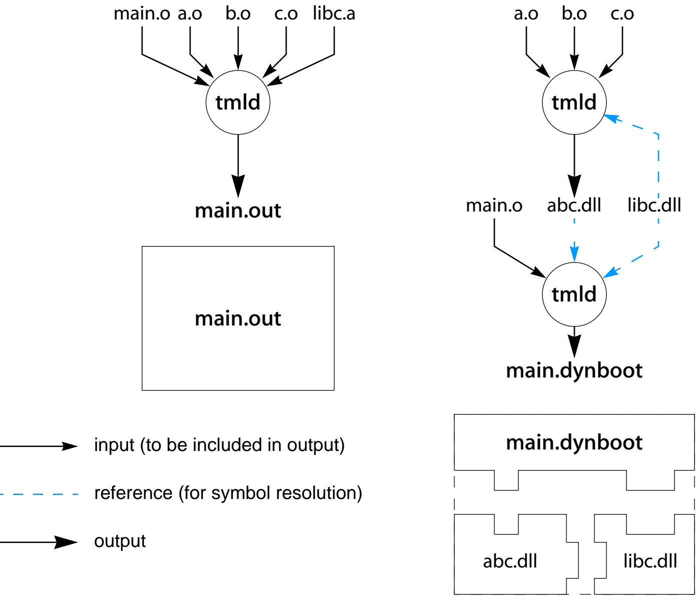


Figure 8 Static vs. Dynamic Linking

Code segments are the fragments from which executables can be constructed during execution, and hence they will not necessarily contain executable code for each referenced function and global data space for each referenced global variable. Because of this, it will be obvious that **tmld** is not able to *bind* all references while producing a code segment. However, similar to linking static executables, it still is a requirement that all references can be *resolved*. In other words, although referenced functions and variables need not yet be included in code segments, **tmld** must know which other code segments export (define) them; this allows **tmld** to generate information by which the proper code segment can be properly located and loaded by the dynamic loader during runtime, when the function or variable is actually needed. The actual *binding* of references, which have already been *resolved* to that code segment by **tmld** during static linking, will take place only when a particular code segment is loaded by the dynamic loader.

Code Segments

The previous section mentioned the similarity between the **tmld** static linker and the dynamic loader: both are used for merging code and binding object references. However, there is a major difference: while the static linker takes object files as input, the dynamic loader operates on code segments. Code segments are the basic unit of dynamic code loading and unloading, and they are constructed by means of the **-btype** option to **tmcc**/**tmld**.

No real representational difference exists between object files and code segments. Both are versions of the TriMedia object format. A code segment is “just” a particular, restricted form of an object file, and the main reason for assigning a code segment a special status is to support static, early detection of errors like unresolved references by letting **tmld** enforce several restrictions when building a code segment.

Four possible types of code segments exist, each of them tailored towards a specific use. The first, the static executable, actually does not have anything to do with dynamic loading, but is included as a particular ‘unit of code loading’; hence, code segments are generalizations of the familiar executables.

Boot segment or boot executable (-btype boot, default)

This is an executable that is intended for initial downloading to the start of SDRAM of an idle TriMedia processor by the TriMedia downloader. The executable may be subsequently activated by bringing TriMedia out of its RESET state. Usually, this sequence is performed by a monitor program like **tmgmon** upon a load or run command, respectively. A boot segment is the default type of executable produced by the compiler driver **tmcc**, but it can not be used for dynamic loading.

A boot segment is required to contain a **main** function by which user execution starts; however, **tmcc** silently includes initialization code for performing a cold processor start before entering *main*: that is, this code unpacks the load image, it sets up stack and heap, it sets up cachelocked and uncached memory regions (when these are used in the executable), it initializes various runtime system libraries (e.g. the I/O library, when I/O is used), obtains the command line arguments from the host (when present), and it performs a reset of some of the TriMedia peripheral devices. Conversely, when **main** terminates, or when it performs a call to **exit**, this code reports the exit status to the host (when present), and after that it will bring TriMedia back into a RESET state. This initialization/termination code is sometimes referred to as ‘the boot code’.

dynboot Segment or dynboot Executable (-btype dynboot)

This type of code segment has a similar function as the ‘plain’ *boot segment*, but it has been set up for dynamic loading. In particular, the dynamic loader itself has been included into the *dynboot* segment.

Similar to the *boot* segment, this segment is loaded during initial program loading by means of the TriMedia downloader library; it is not itself loaded by the dynamic loader. Per definition, and contrary to the next two types of code segments, there always is at most one *(dynamic) boot* segment present in a running TriMedia system. A dynboot segment can dynamically load additional code segments, in contrast to an ordinary boot segment.

Dynamic Library Segment or Dynamic Library (-btype dll)

This type of code segment represents a collection of functions and variables. Contrary to (dynamic) boot segments, there may be multiple dynamic libraries loaded in a running system, although the dynamic loader takes care of that no more than one instance of a particular dynamic library is loaded at the same time: dynamic libraries, with their functions and data, are *shared* between their users.

Dynamic libraries may be explicitly loaded using the explicit dynamic loader API described in Notes and Caveats of Dynamic Loading on page 107. However, more often they will be implicitly loaded by means of directives generated by **tmld** during static linking. Unloading is also possible, but only by means of an explicit call to the dynamic loader API.

Of all four types of code segments, only the dynamic library is able to export functions and variables to other code segments; these exported functions must be specified using options to **tmcc** or **tmld** while constructing the dynamic library.

Application Segment or Application (-btype app)

This type of code segment represents an independent 'program' that can be loaded, executed and unloaded by an execution shell running on TriMedia, or by an operating system using the explicit dynamic loader API.

Similar to *(dynamic) boot* segments, applications must define a *main* function, but the major difference is that, similar to dynamic libraries, more than one application may be loaded and active in a TriMedia system at the same time. Unlike dynamic libraries, however, applications are not shared, and hence even multiple instances of the same application may be loaded and active at a particular moment.

Applications will neither contain boot- nor system initialization code (which they do not need), nor runtime system- and I/O libraries (which they attach to during dynamic loading). As a result, they are usually much smaller than boot executables: while a simple 'hello world' program compiles into a boot executable ranging from 50 kb to over 100 Kb, depending on the host- and I/O libraries used, it will usually occupy less than only one Kb when compiled as an application segment. But while running, it of course requires these libraries to be either present, or to be loaded when it needs them.

```
[1] tmcc hello.c
[2] tmsize a.out
    text   data   data1   bss    dec     hex
    67584   868    640     2708   71800   0x11878
[3] tmcc hello.c -btype app
[4] tmsize a.out
    text  data1 bss __mdesc__ $String$DynLoad$ $Ext_Mod$DynLoad$ dec hex
    384   14    0   100       10               24                532 0x214
```

Figure 9 Size difference of dynboot vs. application segment

 ©1999 Philips Semiconductors 10/08/99

This size difference is illustrated in , which shows a simple program compiled as full boot (default by **tmcc**) and as an application. The three linker generated sections listed by **tmsize** for the application contain data structures required by the dynamic loader.

Simple Examples

Although not all dynamic loading concepts have been explained yet at this stage, it is useful to pause for a few demo programs. This allows the reader to relate the descriptions in previous and following sections to concrete examples, and it also gives some initial impression of the capabilities of the dynamic loader. All of the examples in this section can be found on the TCS release CD, in *$TCS/examples/dynamic_loading*.

Dynamic Library

The sources of this example (found in *$TCS/examples/dynamic_loadingsimple_dll*) provide a simple demo of implicit code loading via dynamic libraries. The (slightly) simplified sources are shown in Figure 10.

main.c

```
void dll_function( int x );
void main(){
   Int i;

   for( i=0; i<10; i++ )
      dll_function(i);
}
```

lib.c

```
void dll_function( int x ){
   printf("dll_function: x= %d\n", x);
}
```

```
tmcc -btype dll lib.c -o lib.dll -bexport _dll_function
tmcc -btype dynboot main.c -o main.out -bdeferred lib.dll -host tmsim
tmsim main.out
```

Figure 10 Demonstration of use of dynamic library

Using the first command shown in the Figure, *lib.c* is converted into a dynamic library that exports its single function for use by other code segments, by means of the option **-bexport** to **tmcc**. Any function or variable that has *not* been exported in this way (that is, the default case) remains invisible from other code segments.

The dynamic library is subsequently used in compiling and linking of its client, *main.out*. Providing dynamic libraries as input during static linking provides the linker with information for *resolving* references to their exported symbols. Only the lists of exported symbols of the dynamic libraries are examined during static linking, and none of their functions or variables are actually included in the created output. Actual loading of dynamic libraries, and *binding* of all references from already loaded code segments to its functions and variables (and vice versa), takes place at run time. This scheme allows

the use of updated versions of the dynamic libraries during execution, which is further described in more detail in *Compatibility Across Versions of Dynamic Libraries* on page 102. Also refer to the next example *Runtime Library Update*.

Due to the illustrated way of library- and program building, no programmer intervention is further required for loading of the libraries; rather, this loading is implicitly performed by the TCS runtime system as soon as it decides that the libraries are needed. The moment of loading can be slightly influenced by means of options to tmcc; this is described in Dynamic Library Roles on page 95.

During runtime, libraries are located by their names via a *search path*. In all monitors provided by TriMedia (**tmsim**, **tmmon**, **tmgmon**, **tmrun** and **tmmprun**), this search path at least includes the current directory, the TCS library directory and the TAS library directory, but it can be extended by the user. The search path can also be influenced by providing a custom I/O driver; this architectural feature is especially useful in embedded systems, and is described in Dynamic Library Search Path on page 95.

Runtime Library Update

The sources of this example can be found in *$TCS/examples/dynamic_loading/upgrade*.

main.c

```
main(){
/* Start using the library; this results in
   implicit loading of the library */
   for( i=0; i<10; i++ )
      printf("square(%d)= %d\n", i, square(i)

/* Simulate an update of the library (just a
   file copy here), and unload its old version
   * from memory */
   copy_file("lib_new.dll", "lib.dll");

   DynLoad_unload_dll("lib.dll");

/* Continue using the library; this again results
   in implicit loading of the library */
   for (i=0; i<10; i++)
      printf("square(%d)= %d\n", i, square(i) );
}
```

lib.c

```
int square(int x){
   printf("unimplemented\n");
   return -1;
}
```

lib_new.c

```
int square(int x){
   return x*x;
}
```

Figure 11 Library Upgrading

Dynamic loading allows the construction of applications from relatively independent code segments that are actually linked together after the application starts executing. As long as their exported interfaces remain ‘compatible’, the individual code segments can be replaced by upgrades without rebuilding the application. In other words: after copying such a ‘compatible’ update over the old library, the application should run as before, except for the effects of the improvements in the library.

This property greatly facilitates upgrading of already released and used software. Especially when a system is constructed from libraries released by different, independent

 ©1999 Philips Semiconductors 10/08/99

developers or even different vendors, upgrading a library usually requires rebuilding all applications that make use of it. This can present problems in the following cases:

1. When the user of the application does not have access to a TCS toolkit,
2. When the user of the application does not have access to the other libraries or sources from which his application has to be rebuilt,
3. When the user of the application is not aware that there is a library to be updated.

When the application has been set up for dynamic loading, and when it has been properly partitioned into code segments, these problems become considerably less severe. Upgrading then becomes an issue of distributing new code segments, and (for users) of just dropping these updates into the place where there applications expect them.

The example described in this section upgrades one of the dynamic libraries that it uses. It does so largely as described above: an upgrade (*lib_new.dll*) is copied over the original version (*lib.dll*). The application in the example is run with both versions, without being rebuilt. Note that although the libraries implementation may change, its name should remain the same. This is because the application 'knows' the library by its name.

Although the above text suggests that the effects of upgraded dynamic libraries are noticed at application startup, when the components are needed and hence loaded, the example implements a more intricate way of upgrading (Figure 11): it replaces the old library *during execution*, by unloading the old version from SDRAM and reloading the new one. In more extensive systems, such runtime upgrade would require knowledge that the library is not currently in use (the unload would otherwise fail), and that the library has no important global internal state that would get lost by unloading.

Application Shell

The sources of the example below (found in *$TCS/examples/dynamic_loading/appshell*), demonstrate a simple shell that uses the dynamic loader API for loading and executing application segments.

appshell.c

```
#include <tmlib/DynamicLoader.h>
Int main(Int argc, String *argv){
   DynLoad_Code_Segment_Handle  module;

   DynLoad_load_application(argv[1], &module);

   exit( (module->start) (argc-1, argv+1) );
}
```

appdemo.c

```
Int main(Int argc, String *argv){
   Int i;
   for (i=0; i<argc; i++)
   printf( "argv[%d]= '%s'\n", i, argv[i]);
}
```

```
tmcc -btype dynboot appshell.c -o appshell.MacOS -host MacOS
tmcc -btype dynboot appshell.c -o appshell.win95 -host Win95
tmcc -btype dynboot appshell.c -o appshell.tmsim -host tmsim
tmcc -btype app appdemo.c -o appdemo.app

tmsim appshell.tmsim appshell.out 1 2 3 4 5 6 7
tmrun appshell.Win95 appshell.out 1 2 3 4 5 6 7
```

Figure 12 Application Shell with Demo Application

The simplified sources of this example (error handling has been omitted here) are shown in Figure 12. Loading applications using such a shell may have the following advantages over running them directly as 'normal' executables of type boot (i.e. the default type of executable produced by the compiler driver **tmcc**):

- Applications do not include boot code, nor any of the TriMedia libraries (i.e. runtime support libraries, ANSI C library and device libraries). Rather, these libraries are either expected to be resident in the application shell or transparently loaded during runtime. As a result they are much smaller than their fully linked boot equivalents. Although this does not immediately save memory during runtime (because the necessary libraries have to be loaded into SDRAM anyway), it does save secondary storage space such as disk space, or flash. This is especially true when more than one compiled program has to be stored, because application shell and TriMedia libraries need to be stored once in the form of dynamic libraries: one application shell plus a number of application segments usually require much less storage space than the same number of self-contained executables.
- The fact that applications do not include any of the TriMedia libraries often allows them to become host- and board independent: they inherit their host dependencies (mostly I/O mechanisms) from the application shell and their board dependencies from *libboard.dll*, assuming here that all variations of TriMedia environments are

encapsulated in the TriMedia board library. More generally, applications that do not have any host- or board dependencies other than via dynamic libraries can be executed without relinking in any environment that provides a specific application shell as well as specific implementations of the (dynamic) libraries that the application uses.

This is illustrated in : the same physical *appdemo.app* can be run on Win95- as well as MacOS-hosted TriMedia boards, and on **tmsim**, by using *appshell.Win95*, *appshell.MacOS* or *appshell.tmsim*, respectively.

The TriMedia monitor programs **tmmon**, **tmgmon**, **tmrun** and **tmsim** facilitate the use of application segments. These monitors use the application shell precompiled from *$TCS/examples/dynamic_loading/appshell* when they detect that an executable object to be loaded is an application segment. In these cases, the appropriate application shell is implicitly loaded instead, and instructed to execute the specified application using the specified command line arguments. The precompiled application shells are stored in the TCS release directory, in *$TCS/lib/<endian>/<host>/appshell.out*. As a result of this support, **tmmon**, **tmgmon**, **tmrun** and **tmsim** are able to run executables like appdemo.app as if they were 'normal' boot executables. Hence, the **tmsim** and **tmrun** commands listed in can be abbreviated as follows:

```
tmsim appshell.out 1 2 3 4 5 6 7
tmrun appshell.out 1 2 3 4 5 6 7
```

Dynamic Loader Shell

This example (found in *$TCS/examples/dynamic_loading/dynamic_loader_shell*), actually is a straightforward extension of the application shell to a multitasking, pSOS-based command shell: the shell consists of one dispatcher task that repeatedly reads command lines constructed from an application name with a number of arguments. For each such command, the dispatcher creates a new pSOS task to load and run the application, while passing the specified arguments, in a way as described in the previous example. Additionally, because the command shell remains active and unused resources have to be cleaned up, the executed tasks unload their application segments when they terminate. Note that creating a new task for each command implies that the commands run in the background, freeing the dispatcher for accepting new commands. This allows starting different independent jobs in parallel, for instance a DVD player plus an independent interrupt latency sampler that monitors the player.

This example presents an interesting example of how a typical (pSOS based) dynamic loading application is transparently structured in different code segments.

Note the following facts and refer to Figure 13, following.

1. The dynamic loader shell is compiled as an application segment. This implies that it does not include boot code, nor any of the TCS runtime system/ANSI/device libraries, nor pSOS. In fact, the entire application size as reported by **tmsize** is not even 7 kb:

```
tmsize dynamic_loader_shell.out
text  data   data1    bss   __mdesc__    $String$  $Ext_Mod$    dec    hex
5440  8      255      840   100          25        48           6716   0x1A3C
```

2. When invoked using e.g. **tmgmon**, this monitor automatically detects that *dynamic_loader_shell.out* is of type *app* and lacks boot code, and hence needs an application shell as described in the previous example for running it. This shell is loaded instead by tmgmon, and instructed to load and start the dynamic loader shell.
3. For obvious reasons, the dynamic loader must always be part of a code segment of type *dynboot* (there would be no way other than dynamic loading of getting this loader into SDRAM, presenting a chicken and egg problem). The same holds for all runtime libraries, I/O libraries and ANSI libraries that are needed by the dynamic loader. In this example, this means that a number of low level system- ('dynamic') libraries have been statically linked in the application shell, and as soon as it is loaded, the dynamic loader shell binds to e.g. the symbols *printf*, *malloc*, *free* etc., and to the dynamic loader API in the preloaded library *libam.dll*.
4. Before starting the now loaded dynamic loader shell, the runtime system detects that this shell needs pSOS, which has not been loaded yet. The necessary pSOS dll is located via **tmgmon's** default search path, and loaded.

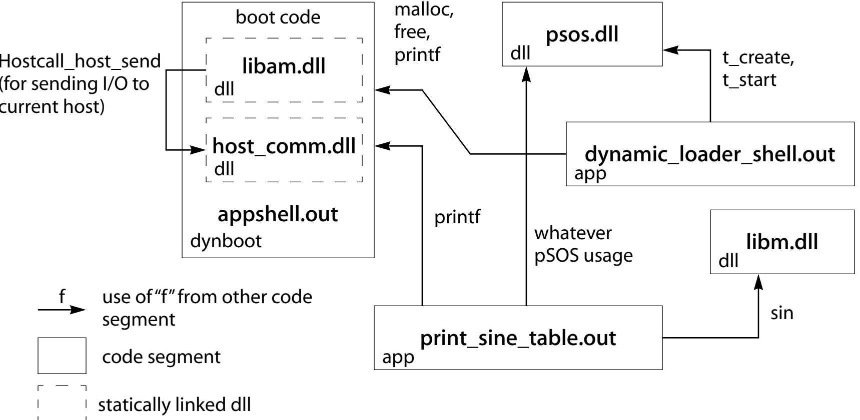


Figure 13 Partitioning of the Dynamic Loader Shell into Code Segments

After that, the dynamic loader shell binds to e.g. the symbols *t_create* and *t_start* in the loaded pSOS library.

5. The dynamic loader shell is now started. It initializes and starts pSOS, resulting in activation of the dispatcher task.

 ©1999 Philips Semiconductors 10/08/99

6. Suppose the user issues the (hypothetical) command '*print_sine_table.app 0 360*', needing both I/O, the math library, and pSOS (for whatever reason). This command causes loading of application *print_sine_table.app*. Again, the runtime system detects that this application requires *libam.dll*, *libm.dll* and *psos.dll*. The first of these libraries was already statically included in the application shell, so the reference to e.g. *printf* in the *print_sine_table* application can immediately be bound to *libam.dll*. Similarly, the calls to pSOS can be bound, because the pSOS library was already loaded for use by the dynamic loader shell itself. *libm.dll*, however, is not yet loaded, so it must be located and subsequently loaded before the sine table printer is able to bind his references to e.g. function *sin*.
7. The sine table is printed, and *print_sine_table.app* unloaded by the dynamic loader shell, using an explicit call to *DynLoad_unload_application* (exported by *libam.dll*, static part of the application shell). The math library *libm.dll,* however, used only during construction of the sine table, is left in memory; it remains unused, and occupies memory until a next application requires math functions, or until it is removed by an explicit call to either *DynLoad_unload_dll*, or to *DynLoad_unload_all* (a call to the last function actually unloads all *unused* code segments).

Responsibilities of the Dynamic Loader

The sole purpose of the dynamic loader is loading, unloading and managing code segments, and, as described below, to keep track of which code segments are still in use and hence should not (yet) be unloaded. In other words, the purpose of the dynamic loader is getting code into and out of a running system, in a reasonably safe way to the user.

Code loading and unloading is completely orthogonal to the concept of multitasking or multiprocessing. For instance, in a multitasking based operating system like pSOS, there is no relation whatsoever between code segments and running tasks: more than one task may be executing code from one and the same code segment, and conversely, one particular task might need multiple code segments for completing its job. Apart from the fact that operation of the dynamic loader is thread-safe in operating systems supported by

TriMedia, both dynamic loading and multitasking are independent mechanisms in a system engineer's toolbox.

app1
```
extern int global;
main(){
   char *x= malloc(5);
   printf( “next global= %d\n”, global++);
}
```

app2
```
extern int global;
main() {
   char *x= malloc(5);
   printf( “next global= %d\n”, global++);
}
```

x.dll
```
int global= 0;
```

Figure 14 Independent applications using a shared dynamic library, and allocating resources

The dynamic loader is also not responsible for managing and keeping apart system resources like address spaces, memory and file descriptors that are allocated by applications.This is considered a shared responsibility of both the ‘current’ operating system, and of the user who should be aware of which operating system that (s)he is using. For example, application segments started under pSOS should release all allocated memory, and all allocated pSOS objects (like semaphores, queues etc.) before they terminate. This is because these resources are considered by pSOS as independent objects that are not ‘owned’ by any particular task; hence they are not automatically released.

On the other hand, in a multiprocessing operating system where applications are run encapsulated in separate processes that ‘own’ many of the resources that they create, applications may be more careless in deallocating, because they can rely on the operating system for automatic cleanup after they terminate. Resources such as memory that are allocated, but not explicitly released, by the applications (see Figure 14) are automatically reclaimed, or left as garbage, depending on the operating system that is in use.

A similar difference holds for the internal global data state of dynamic libraries. It is not the responsibility of the dynamic loader to avoid the interference between different applications that might occur when they both make use of a shared dynamic library that maintains an internal state: address space separation between applications is the concern of the operating system. Figure 14 shows also an example of that: depending on the operating system use; both applications may print the value of 0 (if the OS provides address space protection between applications), or 0 and 1 if (as is the case for e.g. pSOS), all applications run in one shared address space).

 ©1999 Philips Semiconductors 10/08/99

More on Dynamic Libraries

Of the four different types of code segments, only the application segment and the dynamic library can actually be loaded and unloaded during a running TriMedia program. Both of the others (the two boot code segment types) are used for initial code loading to an idle processor and will never be touched by the dynamic loader.

The application segment is comparatively simple to use: it can not be referenced from within other code segments, and loading and unloading can only be achieved by means of explicit calls to the functions **DynLoad_load_application** and **Dynload_unload_application**.

In contrast, dynamic libraries are more complex objects to the user: they can export symbols that can be referenced by other code segments (of any of the four types), and they can be loaded into SDRAM in a number of different ways. These two characteristics introduce a vast number of issues related to external interfaces of dynamic libraries, which are discussed in the next sections.

Dynamic Library Roles

A dynamic library, once constructed, can be both used during static linking by **tmld** and during execution by the dynamic loader. It has different roles in these two stages:

- During static linking of other code segments, it is used for symbol resolution, as a *specification* of its interface. This is illustrated by e.g. the second command in Figure 10 on page 87: *lib.dll* is used as input in linking, but unlike 'normal' object files on the **tmcc**/**tmld** command line, it is only used for *resolving* symbols; no references to it are *bound* yet, and none of its functions and variables are copied in the resulting *main.out*.
- During execution, it is used for loading code. This is the (implicit) role of *lib.dll* in the third command in the Figure: static linking has recorded in the executable *main.out* that *dll_function* was exported by a dynamic library called '*lib.dll*'. During runtime, this dynamic library is located and loaded.

Separating symbol *resolution* and symbol *binding* allows the actually loaded dynamic library to differ from the one that was used for symbol resolution during static linking, as long as the external interface is 'compatible'. This facilitates transparent updating of dynamic libraries, and using dynamic libraries for abstraction, by providing different implementations of the same interface. See *Compatibility Across Versions of Dynamic Libraries* on page 102.

Dynamic Library Search Path

During runtime, libraries are located by their names via a *search path*. In all monitors provided by TriMedia (**tmsim**, **tmmon**, **tmgmon**, **tmrun** and **tmmprun**), this search path at least includes the current directory, the TCS library directory and the TAS library directory, but it can be extended by the user. The search path can also be influenced by

providing a custom I/O driver; this architectural feature is especially useful in embedded systems, and is described in *TriMedia Dynamic Loader Architecture* on page 106, and illustrated in the example Dynamic Loading from Flash on page 110.

The following lists for the various monitors how the dynamic library search path can be extended by the user:

tmsim	option-**dllpath**
tmmon, **tmgmon**, **tmrun** and **tmmprun** on Win95	option **DLLPath** in configuration file tmman.ini
tmmon, **tmgmon**, **tmrun** and **tmmprun** on WinNT	option **DLLPath** in configuration file tmman.ini
tmmon on MacOS	not possible

Exported Symbols

Of the four code segment types, only dynamic libraries are able to export symbols to other code segments. This directly implies that cross-references between code segments are always to variables and functions defined in (and exported from) dynamic libraries.

lib.c

```
void f() { .. }
void g() { .. }
void h() { .. }
```

```
tmcc -btype dll lib.c -o lib.dll -bexport _g,_h
```

Figure 15 Exporting Symbols from Dynamic Library

The external interface of a dynamic library, that is, the complete list of symbols that can be referenced, or used, by other code segments, must be specified during creation of the dynamic library by means of option **-bexport** to **tmcc** or **tmld**. Only symbols that are exported in this way are visible by other code segments. For instance, function *f* in the dynamic library that is constructed in can not be referenced directly from other code segments because it has not been exported with a **-bexport** option to **tmcc**. Multiple **-bexport** options are allowed, and each **-bexport** can have multiple names associated.

Note that the TriMedia compiler prepends an underscore to the C names of external functions and variables. Because dynamic library construction is a linker action based on output of the compiler, the external symbols should be specified as the linker knows them. I.e. an explicit underscore should be added, as shown in the Figure.

In order to facilitate quick lookup of the addresses of a referenced symbols during dynamic loading, the static linker **tmld** constructs an address translation table based on the order of occurrence of the exported symbols in the linker command for constructing the library. That is, symbols that precede others in the export list of the linker command,

 ©1999 Philips Semiconductors 10/08/99

will also precede these symbols in the translation table. More specifically: the occurrence of a particular symbol in the export list defines its index in the constructed translation table. This address translation table is also referred to as the library's 'jump table', and is stored as part of the code segment header that is stored in the library's module description section *__mdesc__*.

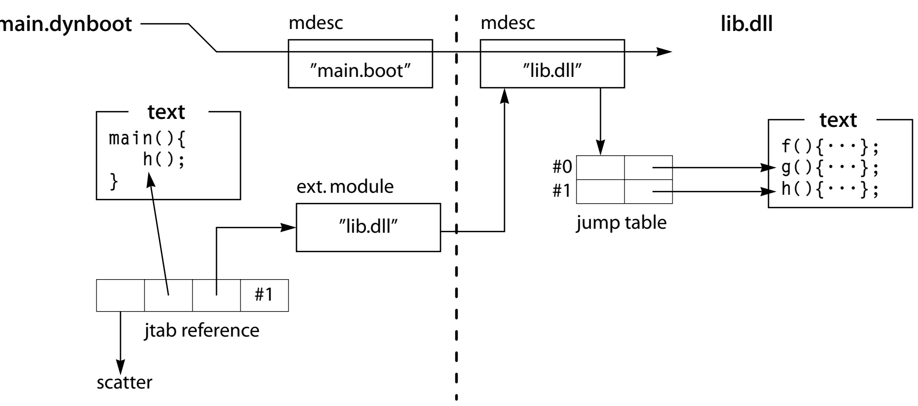


Figure 16 References to dynamic libraries via 'jump tables'

References to exported symbols from within other code segments will be encoded by **tmld** using this index, using an external reference descriptor *JTab_Reference*. This is illustrated in Figure 16. Such a descriptor encodes the address of the referred symbol by means of its index, plus an 'external module descriptor' that describes the library that exports this symbol. This way of encoding allows the exporting library to be identified, located and loaded as soon as the symbol's value is needed. Also, as soon as the library becomes loaded (for whatever reason), the external module descriptor will receive a reference to the library's module header. Because the address translation table is part of this module header, the actual symbol address can be obtained by means of a number of indirections, and using the symbol index.

The following performance related remarks can be made here:

- The module header, including the address translation table, remains in SDRAM, and occupies space, as long as the dynamic library itself remains loaded. The size of the header usually is a few hundreds of bytes, and can be obtained by inspecting the size of the module section *__mdesc__* in the dynamic library via **tmdump**, or **tmsize**.
- Address lookup via the address translation table is performed each time the corresponding symbol is used by the client of the dynamic library, *or only once, when the library is loaded*, depending on the binding mode (**deferred** or **immediate**). See *Control Over Implicit Dynamic Loading* on page 98).
- If the dynamic library was linked in **immediate** mode, address translation is performed when the library is loaded, and no additional cost is involved in accessing the

library's functions and variables. For instance, the overhead in calling the library's functions is no different from the statically linked case.

- References to the library's *variables* are always resolved at library load time, so no additional cost is involved in using variables exported by a dynamic library.
- If the dynamic library was linked in **deferred** mode, calls to its functions are routed via a linker- generated *function stub* (see next section "Function Stubs"). Each time when the library function is called, this function stub checks whether the library is (still) in memory, and it invokes the dynamic loader if this is not the case. In either way, it *binds* the library in preparation of the function call to prevent unloading of the library during the function call (see section "Binding Code Segments" on page -1), it does an lookup of the function's address in the address translation table, calls the function, and afterwards *unbinds* the library. The overhead of a function call via a function stub is about 120 cycles on TriMedia, provided that the function is currently loaded.
  Function stubs are shared between all calls to the same external function in the same code segment; their accumulated sizes can be obtained by inspecting the size of the function stub section *__fstubs__* via **tmdump**, or **tmsize**.
- Similarly, reference descriptors and external module descriptors must remain in memory as long as the referring code segment remains loaded, or can be discarded after loading of the code segment, freeing the memory for other purposes. This depends on whether they are still needed after loading, e.g. by function stubs as described above. As a general exception: for technical reasons, reference descriptors in dynboot segments are always preserved. The total size of reference data that kept in memory as long as the code segment itself remains loaded, can be obtained by inspecting the sizes of the dynamic loader data sections *xxxDynLoad$* via **tmdump**, or **tmsize**.

Control Over Implicit Dynamic Loading

Although dynamic libraries can be loaded using the dynamic loader API, they usually are loaded automatically, and implicitly, as soon as they are 'needed'. There are three different modes of automatic dynamic library loading, to be specified via an option to **tmcc**/ **tmld** when the dynamic library is used for symbol resolution:

- **immediate** mode (default):

  ```
  tmcc -o main.out -btype app main.c -bimmediate lib.dll
  tmcc -o main.out -btype app main.c lib.dll      #same as above
  tmcc -o main.out -btype dynboot main.c -bimmediate lib.dll
  tmcc -o using_lib.dll -btype dll using_lib.c -bimmediate lib.dll
  ```

  Dynamic libraries that are used in this mode by a particular code segment (here: *main.out,* or *using_lib.dll)* will be automatically loaded as soon as the code segment itself is loaded. Also, the dynamic loader will refuse to unload the library as long as the code segment itself cannot be unloaded as well (see also "Binding Code Segments" on page -1). The following examples and notes will give further clarification:

 ©1999 Philips Semiconductors 10/08/99

— Upon an explicit load using the dynamic loader API of the application segment *main.out* produced by the first (or second) command listed above, the dynamic loader will not only load main.out itself, but also library *lib.dll*.

  Upon downloading the (dynamic) boot executable *main.out* that has been produced by the third command listed above, also *lib.dll* will be located and downloaded.

  NOTE: this is how the user should consider this. Actually, it is not the downloader that loads *lib.dll,* but the executable itself after it has been started, while executing its boot code.

— As long as *main.out* (application or dynboot) is executing, the dynamic loader will refuse to unload *lib.dll*.

— Note that loading a code segment can give rise to loading of a cluster of other code segments. In case of the commands listed above, this can occur when *lib.dll* makes immediate use of other dynamic libraries (which might... etc.). Upon loading of any type of code segment, either explicitly using the dynamic loader API or implicitly, the dynamic loader will also load its transitive closure via all **immediate** uses. Cycles in this graph of **immediate** references are allowed and properly handled.

— **Immediate** libraries are only loaded with their clients when this client has at least one reference to it. In other words: occurrence of an **immediate** library on the static linking command line does not imply that is will be automatically loaded: it should also be used.

Note the caveat with respect to errors occurring during loading of **immediate** clusters, in *Notes and Caveats of Dynamic Loading* on page 107.

- **deferred** mode:

  ```
  tmcc -o main.out -btype app main.c -bdeferred lib.dll
  tmcc -o main.out -btype dynboot main.c -bdeferred lib.dll
  tmcc -o using_lib.dll -btype dll using_lib.c -bdeferred lib.dll
  ```

  Dynamic libraries that are used in this mode by a particular code segment (here: *main.out,* or *using_lib.dll)* will be automatically loaded as soon as one of their functions is called. This means that they will not occupy memory as long as they are not used by loaded clients. Dynamic libraries used in this mode can also be unloaded while their clients remain loaded, at least when they are not currently in use (more exactly: as long as they are not *bound*; see *Binding Code Segments* on page 103).

  This loose coupling to their clients has some price, partly because **deferred** libraries can become loaded anytime they are used, and partly because dynamic libraries can be accessed in other ways than calling their functions:

- The possible delay introduced by loading a library upon calling its functions might be undesirable in real time systems (since they could affect real time response). This situation can be avoided by loading such a deferred library at a more convenient stage in the application, and subsequently *binding* it using the dynamic loader API.

- Invoking the dynamic loader from an interrupt handler is disallowed, so the program must guarantee that interrupt handler code will never call one of a **deferred** library's functions with the library not currently loaded. This situation can be avoided by loading such a deferred library at a more convenient stage in the application, in non-interrupt handler code, and subsequently *binding* it using the dynamic loader API.
- Calling a function from a **deferred** library is always redirected via a linker- generated function stub (see next section), even when the library is already loaded. This function stub introduces additional calling overhead. Function stubs, with their overhead, are not used for **immediate** libraries, because these libraries are always guaranteed to remain loaded as long as their clients are loaded.
- Explicit accesses to variables exported by deferred libraries might be unsafe, because contrary to calling a function, there is no automatic mechanism for trapping such accesses. It is the responsibility of the user to guarantee that the library is loaded before the access is made. Also, and more implicit, pointers to the library's static variables and functions that have been exported by means of an earlier call to one of its functions will become invalid when the library is unloaded. They remain invalid, even after a reload of the library, because the library will very probably be loaded at a different place in memory. Again, unloading can be avoided by *binding* the library as long as there is internal state exported.
- Unloading a library will destroy all internal state that has been built up during its use since last load. For instance, a device library that keeps track of the different allocated, or 'opened' instances of a particular device class, should make sure that it is never unloaded as long as there are instances allocated, or else the knowledge about the allocation status would become lost. This can be prevented by the library itself, by *binding* itself as long as there are one or more instances opened.
- **embedded** mode:

  ```
  tmcc -o main.out -btype dynboot main.c -bembed lib.dll
  ```

  Contrary to the previous modes, only dynboot segments can use dynamic libraries in **embedded** mode. This mode lets the library be statically included in the dynboot segment, but in such a way that it is still distinguishable as a separate library by code segments that are loaded at run time. This is illustrated by the following example:

  ```
  tmcc -o shell.out -btype dynboot shell.c -bembed libc.dll
  tmcc -o main.app -btype app main.c -bdeferred lib.dll
  ```

  In this example, two executables are linked: a command shell (*shell.out*), which statically includes the ANSI library, and an application (*main.app*) which can be loaded by the shell. The application doesn't know anything about the shell, but it dynamically refers to the ANSI library that happened to be included in the shell. When loaded by the shell, the application correctly detects this embedded library, and resolves its references to it.

  Apart from the fact that the shell now actually contains the library, and will not profit from newer versions, or from the possible disk space reduction, there is no difference between loading in **embedded** or **immediate** mode.

 ©1999 Philips Semiconductors 10/08/99

The main reason of existence of this loading mode is for static preloading of the system libraries used for dynamic loading itself.

The loading mode is a property of the *usage* of a dynamic library, and not of the dynamic library itself. This means that one and the same library can simultaneously be referred to by different client code segments using different loading modes. This is already illustrated in the shell example above.

Function Stubs

Loading against a dynamic library in deferred mode explicitly allows for the absence of the library while the client itself is or remains loaded.

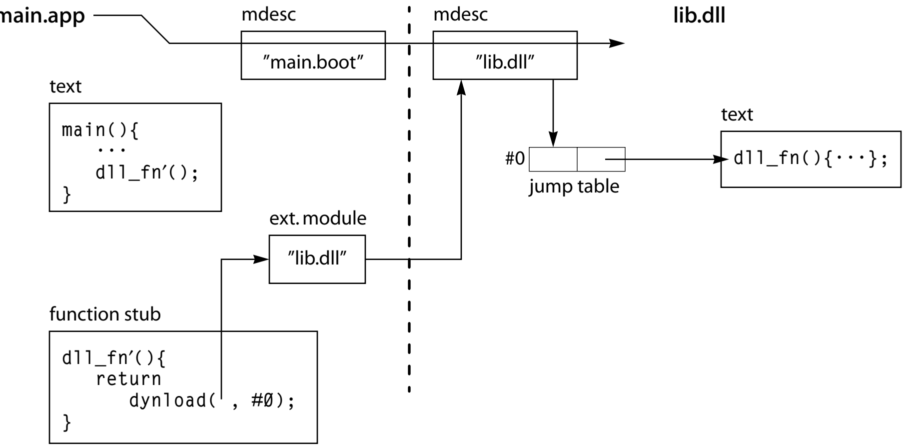


Figure 17 Function Stubs

For instance, in case of the example code listed in on page -1, the dynamic library *lib.dll* need not be loaded before the first call to its function *dll_fn* by the main executable. Also, the library may be unloaded after literally each of the calls to *dll_fn*; such transparent unloading can for instance be performed by an intervening separate task that is responsible for memory cleanup.

By the semantics of deferred loading described in the previous section, the dynamic loader automatically loads deferred libraries like *lib.dll,* as soon as one of their functions is called (provided that they were not already loaded before). The way in which this is achieved is as follows:

For referenced function *f* exported by a dynamic library linked in **deferred** mode, **tmld** generates a *function stub, f′* corresponding to *f*. Each use of *f* (usually a call) then is silently replaced by *tmld* to a use of the function stub *f′*. The function *f′* is henceforth shared among all such uses.

The purpose of this function stub is to trap each call to the corresponding external function *f*, for two reasons:

1. For checking whether the dynamic library is currently present in memory, and invoking the dynamic loader to load it if this is not the case.
2. For binding the dynamic library for the duration of the call to the actual function *f*, in order to prevent an unload of the dynamic library during the function call. The purpose of code segment binding is described in Binding Code Segments on page 103.

The structure of the stub function is shown in ; it is a wrapper function that calls one of the dynamic loader's runtime support functions with the proper arguments, which are a descriptor of the referred dynamic library, plus the index of the required function, **f**.

For visibility purposes, function stubs are placed in a separate instruction section by the name *__fstubs__*.

Function stubs are not generated for calls to dynamic libraries linked in *immediate* mode, because the dynamic loader will guarantee that such libraries are always loaded, and remain loaded, until their clients can be unloaded. Similar for dynamic libraries linked in **embedded** mode (these are statically linked, and can never be unloaded).

Specifically note that references to variables exported by libraries linked in **deferred** mode are allowed. However, unlike automatic support for library loading upon calling its functions, there is no automatic support for loading such libraries upon accessing its variables. It is the responsibility of the user to guarantee that libraries are loaded before any of their external variables are accessed. The linker **tmld** will warn against such variable access.

Compatibility Across Versions of Dynamic Libraries

The TriMedia toolset allows for restricted updating of dynamic libraries between static linking by **tmld** and use during runtime by the dynamic loader. There are two general reasons of why one should want to use a different version of the dynamic library during execution:

- It allows for library updates between linking and execution, without the need for relinking the client(s) of the library. By this, it is possible to let already released, and possibly unknown applications profit from bug fixes and enhancements of libraries that they use, by simply distributing a new dll.
- It allows for different implementations of the same interface, thereby encapsulating hardware- and other dependencies. Because the clients of such libraries only depend on the interface of the dynamic library, they will run unmodified in any environment that provides a proper implementation. One of the previous examples, *Application Shell*, demonstrates how I/O capabilities of the host are encapsulated by the (host specific) ANSI C library.

 ©1999 Philips Semiconductors 10/08/99

Although updating libraries is attractive from a user point of view, it presents potential problems for the TriMedia dynamic linking strategy, in which references to symbols in dynamic libraries from within other code segments are represented using the symbol's index in the library's address translation table (see "Exported Symbols" on page -1): it must be guaranteed that used symbol indices remain the same after a library update. For this reason, the TriMedia toolset poses a restriction on updating of a dynamic library. Violation of this restriction will result in a checksum error during dynamic loading of the library at execution time, resulting in a load failure. More precisely:

> *The dynamic loader will refuse to load a code segment when this would in any way lead to the simultaneous presence in memory of a dynamic library, plus a client of this library that considers the mentioned dynamic library 'incompatible' with the corresponding dynamic library that was used for symbol resolution during static linking.*

In other words, dynamic libraries can only be updated with 'compatible' ones. 'Compatible' is defined as follows:

> *A dynamic library dll_a is compatible with a dynamic library dll_b if it has the same name, and if the export list provided with the creation of dll_a starts with the export list provided with the creation of dll_b (in the same order).*

According to this definition, dynamic libraries may only be updated without losing 'compatibility' by changing the *implementation* of existing symbols, or by adding new definitions whose symbols must be added to the end of the export list.

The following are examples of incompatibility problems:

- (direct case of incompatible load) A code segment loads an incompatible dll.
- (*in*direct case of incompatible load) A code segment loads a dll, but it turns out that another loaded code segment also is a client of this dll. Although the other client currently does not need the dll (otherwise the dll would already have been loaded), it considers the loaded dll incompatible.
- (another indirect case of incompatible load) A code segment is loaded, and detects that an incompatible version of a dll that it references is already in memory.

Binding Code Segments

The TriMedia dynamic loader does everything that is reasonably possible to prevent 'still used' code segments from being unloaded. This protection avoids most of the untraceable errors caused by dangling references from within other code segments to libraries that accidentally became unloaded from memory. However, in the C language, almost any scheme can be defeated, and therefore full protection requires explicit cooperation by the user. This section defines the required user cooperation.

The dynamic loader maintains four parallel criteria for deciding whether a particular code segment is still in use, three automatically maintained by the dynamic loader itself, and one by the user; any attempt to unload a code segment that is still 'in use' according

to any of these criteria will result in a refusal to unload. The (recursive) definitions of the criteria are as follows:

1. The dynboot segment, as well as any **embedded** library is always considered ‘used’.
2. The TriMedia runtime system (conceptually) maintains a reference count for each pair of *(task, code segment)*. This reference count is incremented in the following cases:
   — when the task starts executing code from the code segment.
   — (when the code segment is a dynamic library): during a call to one of its exported functions.

   The reference count is cleared as soon as the task terminates. As long as any of these reference counts is non-zero, the corresponding code segment is considered ‘used’.
3. A dynamic library is considered ‘used’ as long as there is another code segment that has an **immediate** reference to it, and is still ‘used’ itself.
4. Each code segment has a global (task independent) reference count that is available to the user, and which can be explicitly updated using the *bind* and *unbind* functions exported by the dynamic loader API. A code segment with a non-zero reference count is considered ‘used’.

If a code segment is not considered ‘used’ by any of the above criteria, it is considered ‘not used’, and hence can be unloaded.

To maintain the semantics of **immediate**, the dynamic loader also must unload any immediate user of a library when the library is used (which is the reason for the third criterion). For this reason, when a particular code segment is unloaded, also its transitive closure via the reverse **immediate** usage relation is unloaded. Note that the dynamic loader API provides three functions for unloading: unloading an application, unloading a dynamic library (together with all direct or indirect **immediate** users), and a function for unloading *any* currently unused code segment.

main.c

```
void dll_function( int x );
void main(){
   int i;
   dll_function(i);
}
```

lib.c

```
#include “psos.h”
static void task_body() {...}

void dll_function( int x ){
   for( i=0; i<10; i++ ){
      start_psos_task( task_body );
   };
}
```

```
tmcc -btype dll lib.c -o lib.dll -bexport _dll_function
tmcc -btype app main.c -o main.out -bimmediate lib.dll
```

Figure 18 ‘Used’ Code Segments

The effect of all this is illustrated in . This figure shows an application segment that makes **immediate** use of a dynamic library *lib.dll*. Upon call of the function exported

from this library, a number of tasks are created that start executing from the library's code (from static function *task_body*). Even after completion of *dll_function*, it will not be possible to unload the library or the application until all of the created tasks have either completed or deleted. Note that the fact that the application will not use *lib.dll* after its (only) call to this library is unknown to the dynamic loader.

The dynamic loader API provides two types of functions for code segment binding/unbinding: one for binding a dynamic library by its name (implicitly loading it when it has not been loaded before), and one for binding a code segment by an address of one of its functions.

- The last function takes an arbitrary function pointer (which may be of an exported, external of static function), and (un)binds the code segment that contains this function.
- Although binding a dynamic library will *load* the library when it is not yet in memory, *un*binding it will never *unload* it. This should be performed using the *unload_dll* function.
- Binding of an application is implicitly performed using function *load_application*.
- Unbinding of an application is implicitly performed using function *unload_application*, but only when the application segment can actually be unloaded. In other words, application unloading will not have any effect when the application segment's reference count is not equal to 1; it will just return unload failure instead.

These asymmetrical semantics are the result of having segment unloading entirely under user control.

In general, code segments should be bound as long as they have any internal state built up that could become needed by other code segments, or as long as they have any internal state exported that would become dangling if the code segment were unloaded. These are quite general statements, but the following might give some clarification:

- An example of exported state is an installed interrupt handler function. As long as such a function is installed in TriMedia's vectored interrupt controller, the code segment containing this function should remain *bound*. This is just an example, because the TriMedia device libraries *tmInterrupts* and *tmExceptions* will automatically *bind* the code segments of installed interrupt and exception handlers, and *unbind* these code segments upon de-installation.
- Any device library that keeps track of the different allocated or 'opened' instances of a particular device class should *bind* itself as long as there are one or more instances opened.
- Any library that accumulates statistics in whatever form should be *bound*.
- Any library that returns an internal pointer (to global data, or a function pointer) to another code segment by which it is used in **deferred** mode should be *bound* until the pointer is discarded.

TriMedia Dynamic Loader Architecture

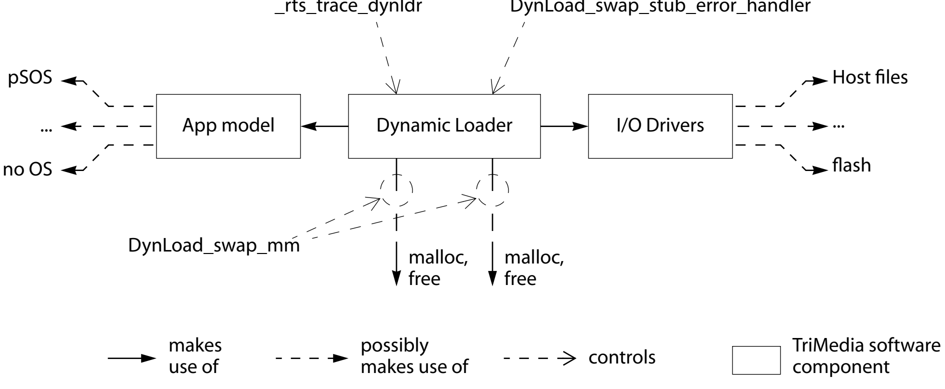


Figure 19 Dynamic Loader Architecture

The architecture of the TriMedia dynamic loader is illustrated in . As indicated by this figure, the dynamic loader is an independent component that is very loosely coupled to its environment:

- It does not make any assumption about the current operating system. Rather, it derives its reentrancy properties from AppModel, the TCS operating system abstraction component.
- It does not make any assumption on the location of the code segments to be loaded, or about the physical medium on which these are stored. Rather, it makes use of the POSIX I/O functions *open, close, read, write*, and one TCS specific function *open_dll*, all of which can be mapped to any medium by installing a proper *I/O Driver.* An example of specifying a custom IO Driver is described in *Dynamic Loading from Flash* on page 110.
- It does not make any assumption about the memory manager that it uses. Rather, it provides a function to install two different memory managers:

1. a temporary memory manager, for allocating and freeing memory that is only needed *during* code segment loading,
2. a permanent memory manager, for allocating and freeing memory for the code segment itself.

   By default, both memory managers make use of the current TCS memory manager. An example of specifying a custom memory manager can be found in Memory Manager Customization on page 111.

- It does not make any assumption on the way in which errors are to be handled. Rather, explicit loader calls to the dynamic loader API return appropriate error codes, and errors during implicit loading in function stubs (while calling a function

exported by a **deferred** library) are handled by a user-installable error handler. By default, this error handler prints a diagnostic and calls *exit*, but user handlers are allowed to e.g. clean up memory and retry, or raise an exception (e.g. perform a *longjmp*) in order to let the error be handled at a higher level in the application. An example of recovering from an error during implicit loading from a function stub can be found in Implicit Loading Error Handling on page 112.

Finally, it exports a flag by which tracing can be enabled: as long as the external variable *_rts_trace_dynldr* has a nonzero value, the dynamic loader will print a diagnostic for each attempt to load (including result status), and for each unload. This is useful for tracing the loading of **immediate** clusters.

Notes and Caveats of Dynamic Loading

Note the following when using dynamic libraries:

Carefully Consider Transitive Errors

Because of **immediate** usage, explicitly or implicitly loading a specific code segment might result in the loading an entire cluster of others. Any errors during loading of such an 'other' code segment will result in an abort of the current load 'transaction': all members of the cluster that have been loaded during this transaction will be unloaded again, and the failure code will be returned as top level error code.

For instance, for the example shown in Figure 18 on page 104, a call to **DynLoad_load_application** for loading *main.out* will result in a *File Not Found* error, not only if *main.out* cannot be found, but also when *lib.dll* cannot be found.
This can be quite confusing, but the real cause of the error can be immediately detected by enabling dynamic loader tracing (by setting *_rts_trace_dynldr* to 1).

Real Time Issue

Although using libraries in **deferred** mode is the most flexible use, it may cause undesirable hiccups due to invocation of the dynamic loader during real time execution. This can be avoided by using libraries in **immediate** mode, or by preloading and binding during a non-real time part of the execution.

Carefully Consider Exporting Internal State

Be very careful with passing pointers to a code segment's global variables or functions to other code segments. These can become dangling when the library is unloaded. Use binding of the code segment, or use the dynamic library in **immediate** mode.

Function Stub is Part of Referring Segment

main.c

```
void dll_function( int x );
Func ff(){
   return dll_function;
}
```

lib.c

```
void dll_function( int x ){
   ...;
}
```

```
tmcc -btype dll lib.c -o lib.dll -bexport _dll_function
tmcc -btype app main.c -o main.out -bxx lib.dll
```

Figure 20 Effect of Stub Functions

In , depending on the way *lib.dll* is used (**immediate**, **deferred**, or **embedded**), the function address returned by function *ff* is either the address of the function *dll_function* in *lib.dll,* or of the corresponding function stub in the code segment where the address of *dll_function* is taken, i.e. the corresponding function stub local to *main.out*.

This could lead to two unexpected effects to users who are not aware of function stubs:

1. When the returned function address is tested for equality with *dll_function* by a third code segment, different stubs of *dll_function* could be compared with each other, thereby producing an unexpected comparison result.
2. When binding *dll_function*'s code segment by means of *DynLoad_bind_code_segment*, the 'owner' of the stub function could be bound instead of *lib.dll*. This might lead to unexpected unloading of *lib.dll*.

Executables are Generally Larger

Enabling dynamic loading generally requires more memory than running an equivalent statically linked executable. This is for two reasons:

1. The dynamic loader administration takes up to a few hundreds of bytes per code segments. This dynamic loader administration is stored in the object file in the sections *__mdesc__* and *xxDynLd$,* and can be shown using **tmsize**. For instance, on page -1 shows an administration size of 134 bytes for an application segment. This size generally grows with 8 bytes per exported symbol (dynamic libraries only).
2. Various linker code size optimizations have less effect. For instance, identical code folding is performed for each code segment individually, so that identical code instances in different code segments may still be left. Also, unused code stripping may have less effect. This is because it is no longer possible to determine whether a particular function exported by a dynamic library is ever used. For this reason, dynamic libraries will generally contain more functions than actually required during any execution. The following examples will clarify this:
   — *libam.dll* (the resident part of the ANSI library) contains e.g. a function *tmpnam*. Although this function is rarely used, it cannot be stripped as unused code,

because it cannot be known whether a loaded code segment actually needs this function.

— *libm.dll* (the math library) consists of 25 functions. An application that uses only, say, the *sin* function will cause all 24 other functions to be loaded because it cannot be known whether a code segment is later loaded that needs *cos*, or *exp*, or *sqrt*, or any of the others.

— The pSOS dynamic library contains all pSOS kernel functions. In case of a statically linked, pSOS based application, **tmld** option **-bremoveunusedcode** will cause all unused pSOS functions to be removed (often all functions except *t_start*, *t_create*, *q_create*, *q_send* and *q_receive*); this automatically results in a tailored, minimal pSOS. However, the pSOS dynamic library must contain all kernel functions, because it cannot be known whether they might be needed by a later loaded code segment.

TriMedia provides a reasonable partitioning into dynamic libraries of all the libraries that it releases. Although the memory needed by dynamic loader administration is marginal, particular users might want to save the space occupied by functions that they know will never, or rarely, be used. This can be achieved by creating custom versions of the TriMedia dynamic libraries from their 'normal' object files. Dynamic libraries can either be partitioned (so that e.g. the use of *cos* will not lead to loading of *sqrt*, by having these functions in different partitions), or just made smaller (so that e.g. *sqrt* is not included at all, because it is not needed in a particular application).

Code Segments and PIC

In multiprocessing operating systems supporting virtual address spaces, the concept of dynamic loading is sometimes connected to the concept of *position independent code* (PIC). A shared library that is loaded only once in physical memory might need to be mapped into the address spaces of different processes, possibly at different (virtual) positions.

Because the 32-bit TriMedia architecture does not include a memory management unit (*mmu*), TriMedia-based systems run in one common physical address space. PIC is therefore not needed for code segments. When loaded, they are visible by (and can be shared between) all running applications.

Compiler Options for Dynamic Loading

The following dynamic loader related options exist; they are recognized by both **tmcc** and **tmld** (in fact, **tmcc** passes them unmodified to **tmld**):

tmcc -btype [**boot** | **dynboot** | **app** | **dll**] ***.c**

Specify type of code segment to be created. Default code segment type is **boot**, which is the usual type of executable that does not allow dynamic loading.

tmcc -btype dll -bexport *sym1*, *sym2*, **... *.c**

Specify symbols exported by the dynamic library that is constructed by the current command. Multiple **bexport** options are allowed. Note that the order of the list of exported symbols is important (see Compatibility Across Versions of Dynamic Libraries on page 102).

tmcc [**-bimmediate** | **-bdeferred** | **-bembed**] *lib.dll*, *lib2.dll*, ... ***.c**

Specify loading mode of dynamic library. Default is **bimmediate**.

More Examples

Dynamic Loading from Flash

main.c

```
#include "tmlib/IODrivers.h"
void main(){
   Int i;

   IOD_install_driver(
      (IOD_RecogFunc) RecogFlash,
      (IOD_InitFunc) Null,
      (IOD_TermFunc) Null,
      (IOD_OpenFunc) OpenFlash,
      (IOD_OpenDllFunc) OpenDLL,
      (IOD_CloseFunc) CloseFlash,
      (IOD_ReadFunc) ReadFlash,
      (IOD_WriteFunc) Null,
      (IOD_SeekFunc) SeekFlash,
      (IOD_IsattyFunc) IsattyFlash,
      (IOD_FstatFunc) StatFlash,
      (IOD_FcntlFunc) Null,
      (IOD_StatFunc) Null
   );
   for( i=0; i<10; i++ ){
      dll_function(i);
   }
}
```

Figure 21 Installing an IO Driver

The sources of this example (found in *$TCS/examples/dynamic_loading/flash_demo*), demonstrate how to perform dynamic loading from ROM (e.g. flash memory) by means of an IO Driver. IO Drivers are groups of callback functions that are used by the TCS file I/O functions (*open*, *close*, *read*, *write*, etc.) when the name of the file matches the driver's 'recognition' function.

presents a flavor of how such a driver is installed. The actual implementation of the driver can be found in *flashfiles.c*. This C source simulates flash memory by regular SDRAM, into which the flash contents (the dynamic library *lib.dll* in this example) are copied during program startup. I/O is implemented by means of memory copying.

 ©1999 Philips Semiconductors 10/08/99

Memory Manager Customization

main.c

```
#include <tmlib/DynamicLoader.h>
static Pointer temp_malloc (UInt size){
   Pointer result= malloc(size);
   printf("*** Temp malloc of size= %d: 0x%08x\n", size, result);
    return result;
}
static void temp_free (Pointer block){
   printf("*** Temp free of 0x%08x\n", block);
   free(block);
}
static Pointer perm_malloc (UInt size){···}

static void perm_free (Pointer block){···}

void main(){
   Int i;

   DynLoad_MallocFun tm= temp_malloc;
   DynLoad_MallocFun pm= perm_malloc;
   DynLoad_FreeFun   tf= temp_free;
   DynLoad_FreeFun   pf= perm_free;

   DynLoad_swap_mm( &pm, &pf, &tm, &tf );

   for( i=0; i<10; i++ ){
      dll_function(i);
   }

   DynLoad_unload_dll( "lib.dll" );
}
```

Figure 22 Installing a Memory Manager

The sources of this example (found in *$TCS/examples/dynamic_loading/ memory_management*), demonstrate how to install a custom memory manager in the dynamic loader. The “custom” memory manager used in this example (shown in) just causes tracing of the allocation/freeing process by the dynamic loader, and further use calls to the “regular” *malloc*/*free*.

Implicit Loading Error Handling

main.c

```
#include <tmlib/DynamicLoader.h>
static jmp_buf  jb;

static void error_handler (DynLoad_Status status, String dll){
   printf("*** Loading of %s failed; error= %d\n", dll, status);
   longjmp(jb,1);
}
void main(){
   Int i;
   DynLoad_ErrorFun errh = error_handler;

   DynLoad_swap_stub_error_handler( &errh );

   for( i=0; i<10; i++ ){
      if( setjmp(jb) == 0 ){
         dll_function(i);
      }else{
         printf("--> recovering\n");
      }
   }
}
```

Figure 23 Implicit Loading Error Handler

The sources of this example (found in *$TCS/examples/dynamic_loading/error_handling*), demonstrate how errors during implicit loading can be trapped and dealt with. The error handler (shown in) signals an exception, implemented by a call to *longjmp*, for letting the error be handled at a higher level in the application.

When the error handler does not signal an exception or aborts execution using e.g. a call to *exit* (that is, when it just returns to its caller), the dynamic loader will retry the load. So in this case, the error handler should try to remove the error condition. This is most useful when handling load failures due to memory overflow: by cleaning up memory and returning, the application can proceed after error handling without even noticing a problem.

 ©1999 Philips Semiconductors 10/08/99

Section Renaming

The **tmld** option **-sectionrename** changes the name of a section (either the text section or one of the three data sections) in an object produced by **tmcc**. Because **tmld** merges all sections with identical names in its input files, a renamed section in an object file produces a renamed section in the final executable. This is shown in the example below:

```
tmld datalock.o -o datalock.o -sectionrename data=my_data
tmcc *.o
tmsize a.out
text     data    data1   bss    my_data   dec      hex
113920   5492    484     3684   4         123880   0x1E3E8
```

Aside from differences in memory mapping and performance, merely renaming a section should not change the behavior of the program. However, there are a number of reasons for renaming a section. First, one might want to group similar data together in one section. An example of this is the code generated for function stubs that is produced for dynamic linking by **tmld** itself: the linker places all function stubs in a section called *__fstubs__* so that the function stub overhead can be obtained quickly by using **tmsize**. Similarly, code segment descriptors containing initialized data structures for use by the dynamic linker are placed in sections *__mdesc__* and *__mdescs__*. A second reason section renaming is sometimes performed is to allow the mapping of specific data to particular memory addresses in embedded systems. Finally, section renaming might be performed in order to give particular data or text a specific treatment in some way or another.

Illustrating the last point, **tmld** allows a number of properties to be assigned to sections: an access property **ro** (read only), caching properties **locked** and **uncached**, and the **shared** (memory) property. Currently, the linker/downloader has no further use for access properties. The shared memory property is discussed in the section on multiprocessing below. Caching properties are interpreted by the TriMedia downloader in the following way (see also the SDRAM memory map in on page 119):

1. All uncached data sections are mapped at the end of SDRAM, and the TriMedia cacheable limit is set to the base of this uncached data region. There is no limit to the amount of uncached data. Specifying uncached text is not allowed. Due to TriMedia limitations, the granularity of the amount of uncached memory is 64 kB, which will be automatically taken care of by the TriMedia downloader by rounding up the allocated amount of uncached memory to the next multiple of 64 kB.
2. All cachelocked data sections are mapped in one contiguous memory region, and the TriMedia cachelocked data region is set exactly to this region. The limit to the amount of cachelocked data is 8 KB, which would occupy half of the TriMedia data cache. The granularity of the amount of cachelocked data is 64 bytes (one cache block). Due to TriMedia limitations, the cachelocked data region must start on a 16 kB boundary. These restrictions will automatically be taken care of by the TriMedia downloader.
3. All cachelocked text sections are mapped in one contiguous memory region, and the TriMedia cachelocked text region is set to exactly this region. The limit to the amount

of cachelocked text is 16 kB, which would occupy half of the TriMedia instruction cache. The granularity of the amount of cachelocked instructions is 64 bytes (one cache block). Due to TriMedia limitations, the cachelocked instruction region must start on a 32 kB boundary. These restrictions will automatically be taken care of by the TriMedia downloader.

Caching properties are ignored by the dynamic loader, so that they have effect only for code segments of type **boot** or **dynboot**. The reason for this is that the 32-bit TriMedia architecture allows only a single range of SDRAM to be uncached and a single range of SDRAM to be cachelocked. These ranges have been assigned for use by the **(dyn)boot** executables, so none is left for code/data that is loaded at a later time.

The following example shows how the section *my_data* should be placed in cachelocked memory. Note that both **sectionrename** and **sectionproperty** refer to the *original* name of the section, *data*. The section is also renamed to prevent errors during further linking with other normally cached sections which are also named *data*.

```
tmld datalock.o -o datalock.o -sectionrename data=my_data \
     -sectionproperty data=locked
```

Sections Produced by tmccom

In order to effectively use section renaming, one must have an understanding of the initial section assignment of text and data by **tmcc**. This is as follows:

1. All program code compiled into TriMedia instructions is placed in a section named *text*.
2. Some compiler-generated data such as jumptables for switch statements are placed in a section named *data1* (read-only data).
3. All initialized global C variables are placed in a section named *data*. For example:

```
int a[300]= {0};
```

4. All uninitialized global C variables are kept as *common symbols*, to be resolved in further linking, or otherwise mapped by the linker when producing an executable. For example:

```
int a[300];
```

Common symbols are not yet mapped to a particular section, because they might be resolved during subsequent linking to initialized global variables with the same name. When they are still not resolved at the creation of the final executable, the linker allocates space for common symbols in a section named *bss*.

The latter would make it somewhat awkward to isolate uninitialized data using section renaming, because it would not yet have been mapped to any section in the output of **tmcc**. So merely renaming the *bss* section in an object file containing uninitialized data would result in an empty renamed section, with the common symbols still placed into *bss* during final linking of the executable. For this purpose, **tmld** provides an option

-map_commons. The effect of this option is to prematurely force all common symbols into the *bss* section. This option takes effect before section renaming, so that the following **tmld** command would indeed create a section *my_data* containing the uninitialized data:

```
tmld datalock.o -o datalock.o \
   -map_commons \
   -sectionrename bss=my_data \
   -sectionproperty bss=locked
```

Note that **-map_commons** might change the program semantics. For example, suppose two different source files contain definitions of a variable **a**, one as shown below.

```
int a [300] = {34};
```

ANSI C semantics require that these should be treated as two definitions of the same memory object. However, applying **-map_commons** to the object file produced from the second source file has the effect of changing the definition to

```
int a [300] = {0};
```

This results in a duplicate symbol error in further linking.

IMPORTANT
Using this option, unexpected common symbols might be caused by including header files containing variable declarations without the qualifier **extern**.

See also *$TCS/examples/misc/cachelocked* for an example defining sections in cachelocked/uncached memory.

Other Sections Produced by SDE Tools

The following table describes other sections that will sometimes be introduced by the SDE tools, and which might be encountered in the output of **tmld** and **tmdump**. All sections but the first (*debug*) contain information that must be kept with loaded code segments, and are only introduced under certain conditions when compiling/linking for dynamic loading.

Section name	Creator	Description
debug	tmccom	Debug information (stabs). Used by tmdbg.
__fstubs__	tmld	Function stub code.
__mdesc__	tmld	Module descriptor record for current code segment, used by dynamic linker/loader.
__mdescs__	tmld	List of module descriptor records for all embedded code segments, used by dynamic linker/loader.
SymDynLoad$	tmld	Descriptors of the symbols which may sometimes be left in symbol references. Used by dynamic loader.

Section name	Creator	Description
Ext_ModDynLoad$	tmld	Descriptors of other code segments to which the current code segment has references. Used by dynamic loader.
$String$DynLoad$,	tmld	String table containing names of objects from the previous two sections.
Sym_RefDynLoad$l, $Mrkr_Ref$DynLoad$, $JTab_Ref$DynLoad$, $FromDef_Ref$DynLoad$, $DefDef_Ref$DynLoad$,	tmld	Different representations of references which are needed by dynamic loader.
$Sctr$DynLoad$, $Sctr_Src$DynLoad$, $Sctr_Dest$DynLoad$	tmld	Scatter descriptors. The bit positions within code units where address values have to be filled in are generally non-contiguous; rather, they are scattered around a certain location. Scatter descriptors describe how.

Link Optimizations

tmld provides a number of options for reducing the size of a generated executable. When used, they typically achieve a size reduction of 20–30% without a noticeable performance impact. However, they impair debugging, because all of the options are incompatible with the use of **tmdbg**, because the code might be considerably reordered or reused. The options are only valid while generating code segments:

- **-bremoveunusedcode**

  This option attempts to find out which dtrees and global variables cannot be used when starting execution from either the start symbol or from any of the dynamically exported symbols. Unused code and data is removed from the output.

- **-bfoldcode**

  This option causes an analysis of all *read only* sections (typically instruction sections), in order to find identical code instances. Such identical code most often consists of identical function epilogues generated by the compiler, or (in a C++ context) of identical instantiations of a code template. For each occurrence of such identical code, one instance is chosen as a representative and all others are removed.

  Note
  This option should not be used in a program where function addresses/ pointers are compared to something other than NULL. This option causes functions with identical content to have identical addresses which may cause semantic differences in programs where function pointers are compared.

- **-bcompact**

  This option reorders the code (at the dtree level) in order to minimize the amount of instruction padding that is required by the TriMedia architecture. In contrast to **-bremoveunusedcode** and **-bfoldcode**, this option might have an adverse performance impact because it reduces instruction cache locality. Although a study on a large TriMedia benchmark set showed that reduced code size seemed to compensate for reduced locality in all cases, it is advisable to verify this in particular applications for which the **-compact** option is used.

The effects of these options can be verified using **tmnm**, **tmsize**, and **tmprof**.

Multiprocessor Support

The TCS toolkit provides basic support for a shared memory-based multiprocessor setup, in which different TriMedia processors are able to access each other's SDRAM and MMIO spaces over the PCI bus interface. Such a setup can be created by inserting multiple TriMedia boards into the PCI slots of a single PC, but it can also be devised as a multiprocessor stand-alone board.

Any form of communication in such a system is founded on two concepts: *node identification* ("node" being a TriMedia processor) and *shared memory*. Node identification is needed for distinguishing between the different nodes. Shared memory is needed as an initial means for passing data to other nodes. With support for these basics as a foundation, more intricate forms of communication can be built.

The TriMedia downloader **tmld** and the TCS standard library support the concept of a *multiprocessor cluster*, which is a number of TriMedia processors that together run a multiprocessor application. A **boot** or **dynboot** executable is loaded on each of the processors.

Processors in a multiprocessor cluster are assigned node identifications, being node numbers in the range *0 to N-1*, which are substituted for the download symbols *_node_number_init* in the executables that they run. Substitution is performed by the TriMedia downloader under control of the monitor. Processors in a cluster may run copies of the same executable, or different executables, or a mixture of these. For instance, the tool **tmmprun** allows starting a cluster as follows:

```
    tmmprun -exec a.out -exec b.out -exec a.out
```

This command allocates three processors (when available), numbers them 0,1,2, and loads *a.out* on the first and third, and *b.out* on the second while substituting their respective node numbers.

Note that the node numbers form a symbolic identification, which has no explicit relation with the corresponding physical processors. For instance, in a six-processor setup, a load command as described above could be given twice (not currently supported by **tmmprun**), resulting in two independently running three-node clusters (apart from a likely PCI congestion) which each have numbers 0,1,2. A slightly more intricate monitor

must be used where an explicit loading of cluster members on particular processors is desired. Note again that node *numbering* is the purpose of the monitor, while the node values are *filled in* by the downloader, which is a library used by the **tmmprun**.

The node numbers are available to the user (see *tmProcessor* device library). They are also used in several system libraries. For instance, pSOS+m bases its node numbering on the TCS node numbers. In addition, hardware semaphores can be accessed using the node number (see *tmSEM* device library), interrupts can be sent to other nodes using the node number (see *tmInterrupts* device library), and a macro is available for accessing the MMIO spaces of other nodes using their node numbers (see <tm1/mmio.h>).

Shared Memory

Although all TM-1 processors in a cluster run in one shared address space, specific sections can be assigned the property *shared*. Usually, in the case of non-shared sections, the downloader will load a separate copy of the executable specified for each node. For instance, in the three-processor load command specified above, node 0 and node 2 each will receive a full copy of the executable *a.out*. The only difference is in relocation, because of the different SDRAM address ranges. So each of nodes 0,1,2 will have sections *text*, *data1*, *data* and *bss*.

In contrast to this, specific sections can be assigned the property *shared*. While downloading the executables of a multiprocessor cluster, memory will be allocated for *only a single copy* for shared sections of the same name. All executables in the cluster will be made to refer to this single copy. For instance, the following will define a shared section with initialized data (note that the shared section here is also defined as uncached to avoid the need for cache coherence).

```
int shared= 357;
tmcc shared.c -o shared.o -c
tmld shared.o -o shared.o \
   -sectionrename data=shared \
   -sectionproperty data=uncached \
   -sectionproperty data=shared
```

When linked to both *a.c* and *b.c* shown below, two executables of a simple two-node multiprocessor cluster are obtained.

```
............ a.c ................
extern volatile int shared;
main() { do { printf(“shared= %d0, shared); } while (shared == 357); }
............ b.c ................
extern volatile int shared;
main() { int i; for (i=0; i<10000000; i++){} shared= 308; }
..................................
tmcc a.c shared.o -o a.out
tmcc b.c shared.o -o b.out
tmmprun -exec a.out -exec b.out
```

A more realistic example is the pSOS+m multiprocessor interface. This interface defines a message-passing library based on shared memory. One shared section contains a simple array of message queue addresses, one address per node, to be indexed by node number.

 ©1999 Philips Semiconductors 10/08/99

When coming up, each processor allocates a message queue in its own memory, and places the address in the shared array. Hence, this *directory* is to be used by each of the nodes to detect whether a particular sibling node is running, and whenever this is the case, to get access to its queue data structure.

In simple setups, shared sections should be defined as uncached. However, if they are willing to deal with cache coherence to reduce the memory access overhead over PCI, users can leave shared sections cached while implementing a coherence scheme using the routines *_cache_copyback* and *_cache_invalidate* in the TCS library. Also, the *tmDMA* device library can be used to speed up data transfer over PCI. See *$TCS/examples/multiprocessing/data_streamer* for a pSOS+m + DMA- based interprocessor data transport example.

SDRAM Memory Images vs Load Images

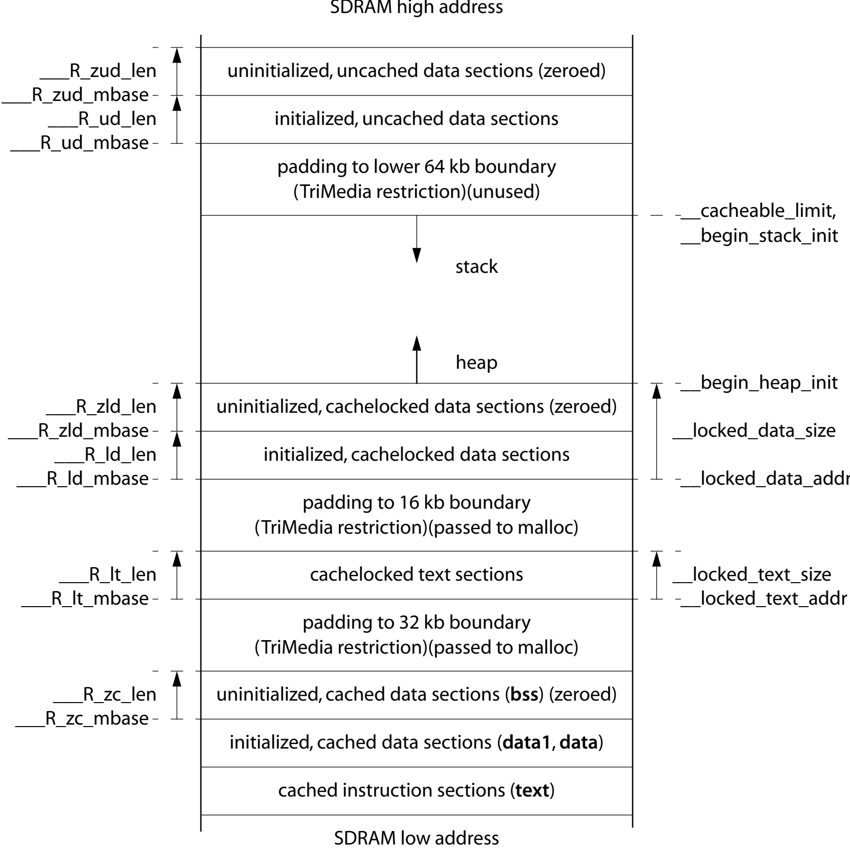


Figure 24 SDRAM Image Map

Note
The symbols at both sides are download symbols for passing mapping information to the executable. See the section on reserved download symbols on page 123.

When the downloader is used to construct a load image from an executable object file (either implicitly via **tmld**, **tmmon** or **tmrun**, or directly by calling the downloader library from user applications), it first constructs an SDRAM memory image map. This memory map identifies the locations of a number of different clusters of sections during execution. The executable's sections are clustered because the 32-bit variants of the TriMedia hardware requires that all cachelocked and uncached data sections, and all cachelocked instructions are grouped, and also for facilitating the zeroing of the different types of "uninitialized" data sections. The following section clusters are used by the downloader:

- cached instruction sections
- cached, initialized data sections
- cached, uninitialized data sections
- cachelocked instruction sections
- cachelocked, initialized data sections
- cachelocked, uninitialized data sections
- uncached, initialized data sections
- uncached, uninitialized data sections

Figure 24 on page 119 shows how these clusters are mapped into SDRAM by the downloader. Due to 32-bit TriMedia hardware restrictions, the different section clusters cannot be mapped into a single consecutive region: the first reason for this is that the cluster of cached instruction sections must be mapped at the start of SDRAM (because the start address should be located at the very beginning of SDRAM), while the uncached data sections must be grouped at the end of SDRAM; as a second reason, major areas of padding have to be introduced because the cachelocked data- and instruction memory regions must start at 16 kb and 32 kb boundaries, respectively, and because the uncached data memory region must start at a 64 kb boundary. These major areas are also shown in the Figure; two of the padding areas, with a total size of up to 48 kb, are recycled to the application's heap by the TriMedia standard boot code (contained by the files *reset.o* and *boot.o* which are automatically added to executables by the compiler driver **tmcc**), after application startup. The padding in front of the uncached data section cluster is left unused.

Note
Various minor padding between sections and sectiongroups may also exist, but this is not shown.

The gap of SDRAM that is not occupied by section clusters, is available for the executable's combined stack/heap extension area. The standard boot code will initialize the

 ©1999 Philips Semiconductors 10/08/99

stack pointer to the top of this area, and the heap extension pointer to the bottom of this area.

Note
At the start of a pSOS- based application, the entire stack/heap extension area will be allocated and given to pSOS region 0 (the pSOS default heap). All tasks will allocate their stacks from this heap, resulting in a slightly more complicated situation in which stacks, allocated data on the heap, and unallocated, available heap blocks are all mixed in region 0, in the former stack/heap extension area.

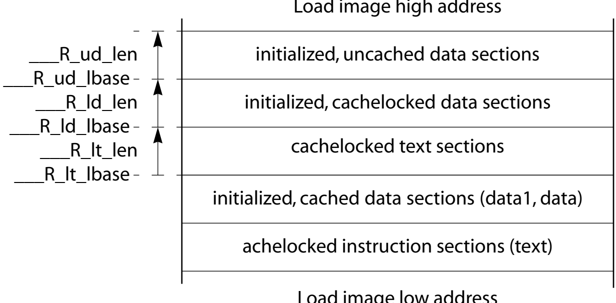


Figure 25 Packed Load Image Map

Note
The symbols at the left node are download symbols for passing mapping information to the executable. See *Reserved Download Symbols* on page 123.

In many cases, the memory image can simply be used as load image. That is, the load image to be copied to SDRAM then will contain all initial section contents at the exact positions at which they are needed during execution. However, with the restriction that a load image should be just a single sequence of bytes that can be stored in EEPROM, for example, and may simply be copied to SDRAM prior to execution, this would have the following (sometimes undesirable) consequences:

- Each image, even the simplest, that contains an uncached data section would have size of the used SDRAM (usually 8 Mb)
- Each image containing cachelocked sections would waste up to 48 kb of padding.
- Each image would waste a number of bytes that is equal to the total size of uninitialized data sections (for instance, bss sections), which becomes noticeable in programs using very large global buffers.

To avoid these consequences, the downloader constructs a load image that is different from the eventual memory image, in that it contains only the initialized section clusters, without the major padding areas (see on page 121). The cluster positions in the load image and in the memory image, with their sizes, are passed via a set of reserved down-

load symbols to the executable. Using these positions and sizes, the executable is able to unpack itself after startup by moving the initialized section clusters to their proper positions in SDRAM, and by clearing the uninitialized section clusters.

This unpacking is performed by default, by the TriMedia standard boot code. However, when the downloader suspects that an executable is not capable of image unpacking, it still generates a load image that is equal to the executable's SDRAM memory image. The downloader's criterion for this is the absence of a downloader symbol with the reserved name ___Rdo_unpack. If a download symbol with this name is present, then a packed load image will be generated; if not, then the SDRAM memory image will be used as load image. Whenever a packed load image cannot be constructed, an error will result when the executable has uncached data sections.

Constructing Load Images Using tmld

tmld generates its output by default in the TriMedia object format. This is necessary when the output is used in further links stages, or (for executables) when the output is to be downloaded at a yet-unknown SDRAM address.

Sometimes a more simple output of **tmld** is desired. For instance, in the case of standalone booting (described in Chapter 7, *Bootstrapping TriMedia in Autonomous Mode* of Book 2, the *Cookbook*) the L1 boot program is copied as-is from the boot EPROM by the hardware, and there obviously is no opportunity for relocation of this L1 boot program.

Also, the size of an L1 boot program is restricted by the TriMedia to 2 kb, so that it is again impossible to relocate the next stage, the L2 boot program. For these reasons, both the L1 and L2 boot program must be of a very simple format.

When using option **-mi** while linking an executable, **tmld** generates a relocated load image that can be directly copied to SDRAM. Because it generates a relocated image, `-mi` requires the begin and end addresses of the SDRAM area to which the image is to be loaded, as well as the MMIO base address and the processor clock frequency. Optionally, a symbol map file may be generated.

The following options are related to memory image generation:

-mi	enable load image generation
-load	specify SDRAM load area (required)
-mmio_base	specify MMIO base address (default 0xefe00000)
-tm_freq	specify TriMedia cpu frequency (default 1000000 (Hz))
-symt, -symtfile	generate symbol map to standard output, or to file
-rmap, -rmapfile	generate segment map to standard output, or to file

 © 1999 Philips Semiconductors 10/08/99

The following is an example of constructing load images using **tmld**:

```
tmcc main.c \
   -tmld -mi -start 0,0x800000 --
```

```
tmcc main.c \
   -tmld -mi -start 0x800000,0x900000 \
   -mmio_base  0xa00000  \
   -symtfile  main.symbtab --
```

Download Symbols

Aside from the usual symbol types *local*, *global*, *unresolved* and *common*, the object format also supports one particular type of symbol: the *download* symbol. A download symbol is a symbol that is intended to be resolved by the downloader, typically for passing information that cannot be known before downloading. Download symbols can be defined using **tmld** option **-bdownload**, as in

```
   tmld reset.o -o reset.o -bdownload __start_stack_init
```

tmld will refuse to resolve download symbols.

Reserved Download Symbols

The following list describes the download symbols that are reserved for use by the TriMedia downloader itself. They are automatically given values at the call to the functions **TMDwnLdr_relocate** or **TMDwnLdr_multiproc_relocate** from the downloader library. Usually these functions are called internally by the tools **tmmon**, **tmgmon**, **tmrun**, **tmmprun**, **tmsim** or **tmld**, when constructing a load image. These relocation functions map the different sections of a downloaded executable into the available SDRAM, based for example on section-caching properties, section alignment and SDRAM size, and pass information on this mapping, together with other basic information, to the loaded executable via the appropriate download symbols. The values are resolved whenever the executable actually uses the corresponding download symbols, otherwise they are ignored. Figure 24 on page 119 shows how the different sections, stack and heap are mapped in SDRAM by the downloader.

Table 6 Downloader Symbols

Symbol name	Description
__begin_stack_init	The initial top of stack address, word aligned.
__begin_heap_init	The initial start of heap address, word aligned.
__number_of_nodes_init	An integer value specifying the number of TriMedia processors (nodes) in the multiprocessor cluster to which the executable belongs.

Table 6 Downloader Symbols

Symbol name	**Description**
__node_number_init	An integer value specifying the integer node identifier of the current TriMedia processor in the multiprocessor cluster to which the executable belongs. Node numbers range from 0 to number_of_nodes-1.
__host_type_init	An integer value describing the current type of host as defined in <tmlib/tmhost.h>.
__MMIO_base_init, __MMIO_base<i>_init	The base addresses of the MMIO space of the current processor, and of processor with node number i.
__clock_freq_init	The TriMedia clock frequency.
__segment_list_init	A handle to a list of initialized data structures in SDRAM describing the code segments (dynamic boot segment, plus embedded dynamic libraries) which were part of the downloaded application.
__do_section_lock	A value describing whether TriMedia's cacheable limit and locked regions should be set according to the next values, just after start of the downloaded application. The setting is performed when this value is not equal to zero; the next values are ignored otherwise. The next values depend on whether the downloaded application contains specially cached sections.
__Rdo_unpack	A value describing whether the image should be unpacked by the boot code, at program startup, according to the values of the download symbols **_Rxx_ibase**, **_Rxx._mbase** and **_Rxx_len**, described below. The unpacking is performed when this value is not equal to zero.
__Rld_ibase	Start of block of cachelocked, initialized data sections in load image.
__Rld_mbase	Start of block of cachelocked, initialized data sections in SDRAM, during execution of the program.
__Rld_len	Length of block of cachelocked data sections.
__Rlt_ibase	Start of block of cachelocked text sections in load image.
__Rlt_mbase	Start of block of cachelocked text sections in SDRAM, during execution of the program.
__Rlt_len	Length of block of cachelocked text sections.
__Rud_ibase	Start of block of uncached, initialized data sections in load image.
__Rud_mbase	Start of block of uncached, initialized data sections in SDRAM, during execution of the program.
__Rud_len	Length of block of uncached, initialized data sections.

 © 1999 Philips Semiconductors 10/08/99

Table 6 Downloader Symbols

Symbol name	Description
__Rzc_mbase	Start of block of normally cached, uninitialized data sections in SDRAM, during execution of the program (this block should be zeroed at program start).
__Rzc_len	Length of block of normally cached, uninitialized data sections.
__Rzld_mbase	Start of block of cachelocked, uninitialized data sections in SDRAM, during execution of the program (this block should be zeroed at program start).
__Rzld_len	Length of block of cachelocked, uninitialized data sections.
__Rzud_mbase	Start of block of uncached, uninitialized data sections in SDRAM, during execution of the program (this block should be zeroed at program start).
__Rzud_len	Length of block of uncached, uninitialized data sections.
__locked _data_size	Length of the entire SDRAM memory area which must be locked in the data cache. This value spans the ld and the zld area, and is rounded to the next multiple of the data cache block size.
__locked_text_size	Length of the SDRAM memory area which must be locked in the instruction cache. This value spans the lt area, and is rounded to the next multiple of the instruction cache block size.
__cacheable_limit	Start of the SDRAM memory area which must not be cached.

Figure 26 Reserved Download Symbols

Other Download Symbols Used by the TriMedia SDE

The following list defines several host-specific download symbols. These and other user-defined download symbols must be explicitly given values using function **TMDwnLdr_resolve_symbol** by the tools that call the download library (usually **tmmon**, **tmgmon**, **tmrun**, **tmmprun**, and **tmsim**). See also the description of the downloader library starting on page 123. **tmsim** and most of the monitors and run programs check for the presence of these symbols while loading executables, in order to verify whether the executables have been compiled and linked for “their” host. This results in warnings

for executables that have been compiled with *-host=nohost* (actually: any host), because such executables do not have any "other" download symbol defined.

Symbol name	Host type	Description
__syscall	tmsim	Simulator system call trap function.
__HostCall_commvar_init	MacOS	Address of two-word communication buffer used by system call RPC implementation.
_TMMANSharedPatch	Win95	Address of shared, pagelocked communication buffer used by the *TriMedia Manager* (PC/TriMedia communication module).
_TMManShared	WinNT	Address of shared, pagelocked communication buffer used by the *TriMedia Manager* (PC/TriMedia communication module).

tmld Options

The following are **tmld** command options.

-bcluster
Code size optimizations **-bcompact**, **-bfoldcode** and **-bremoveunusedcode** usually result in reordering the contents of code and data sections, and in deletion of unused parts. This option combines the contents of each section in the output file produced by **tmld** in such a way that reordering or partial omission of these contents is prevented.

-bcompact
Reorder dtrees for code compaction.

-bdeferred *file* [**,** *file*]...
Use the code segments in the specified files for resolving symbols dynamically. Each file will get loaded at runtime upon the first call to one of its exported functions (if it is not already loaded). A warning is generated for all references to data in code_seg.

-bdownload *symbol* [**,** *symbol*]...
Allow the specified symbols to remain unresolved, to be defined by the TriMedia downloader.

-bembed *file* [**,** *file*]...
Embed the specified dynamic libraries in the output file produced during static linking. Embedding can be considered as static preloading in the sense that the embedded library is still distinguished as a separate library. For instance, applications linked against *libc.dll* (containing the ANSI library) will correctly detect this library when loaded by a boot segment in which this library was embedded. Embedding is generally

 ©1999 Philips Semiconductors 10/08/99

	used for preloading system libraries, such as the libraries needed by the dynamic loader itself. Embedding is only allowed when the output file is of type **dynboot**.
-bfoldcode	Perform identical code removal.
-bexport *symbol* [**,** *symbol*]...	Declare the symbols in the list as exported to the dynamic loader.
-bimmediate *file* [**,** *file*]...	Use the specified code segments for resolving symbols dynamically. Each file will get loaded as soon as the code segment which references it is loaded (if it is not already loaded).
-bremoveunusedcode	Perform unused code removal.
-btype [**boot** | **dynboot** | **app** | **dll**]	Specify type for the produced output file. The default is an object file which can be further linked.
-chain	Create a linked list at a specified symbol by chaining together all occurrences of a given variable at link time. For more information, see *List Construction by tmld* on page 130.
-debug | **-nodebug**	Omit source level debug information from output module.
-exec	Sets the default type of the produced output file to **boot**, instead of to an ordinary object.
-g	Inform **tmld** that debugging on the executable is required (default false). This will let **tmld** disallow options which are incompatible with debugging.
-h	Print help information on the command syntax and options, then exit.
-lib *file* [**,** *file*]...	Search the specified library modules in each given file in the given order for definitions of unresolved symbols. **tmld** loads the required library modules from each library. **tmld** searches each library in turn and loads all needed modules from it before searching the next library. Searching and loading each library is a one pass process; the search determines all the modules needed, taking into account intermodule dependencies within the library.
-load *begin_memory*, *end_memory*	Specify download memory region for option **-mi**. The arguments are numbers in C format: when starting with a digit from 1 to 9, they are interpreted as decimal values, when starting with a 0 they are interpreted as octal values, and when starting with 0x or 0X, they are interpreted as hexadecimal values.

-map_commons	Map common symbols to bss section (default false). This mapping is performed for executables, regardless of this option. However, this flag allows to do it also for object files which are intended for further linking.
-mf *file*	Specify machine description file.
-mi	Produce binary image for downloading onto TriMedia from a host.
-mmio_base *address* [**,** *address*]...	Specify MMIO base addresses. This option is only allowed when producing a memory image using option **-mi**. By default, the value 0xEFE00000 will be used, with a generated warning.
-node_number *n*	TriMedia node number when generating This option is only allowed when producing a memory image using option **-mi**. By default, the value 0 will be used.
-optf *file*	Process the **tmld** options contained by file.
-o *file*	Specify file name for output module. By default, the name a.out will be used.
-preserve	Preserve the output module even if linking failed.
-P *symbol* = *value* [**,** *symbol* = *value*]...	Patch symbol with the associated 32-bit values. This option is only allowed when producing a memory image using option -mi.
-R *symbol* = *value* [**,** *symbol* = *value*]...	Resolve a *symbol* with the associated (absolute) *value*. This option may be used to resolve download symbols.
-rmap	Print a textual relocation map to the standard output stream. The map indicates the locations of all sections from each input module and loaded library module in the output module. It also indicates gaps created to satisfy alignment requirements. It shows base address, sizes and relocation offsets for each section, as well as the total size of each linked section. This option is only allowed when producing a memory image using option **-mi**.
-rmapfile file	Print a textual relocation map to specified file. This option is only allowed when producing a memory image using option **-mi**.
-sectionrename *section* = *newname* [**,** *section* = *newname*]...	Rename sections. This option is useful for isolating the data from a particular object file and giving it a special property, while keeping it separate

 © 1999 Philips Semiconductors 10/08/99

in the further linking process. The specified old section name is the name of the section in the inputfile(s), regardless of any earlier section renamings on the command line.

-sectionproperty section = property [, section = property]...
: Set caching or access properties for sections, to be interpreted by e.g. the TriMedia downloader. The term property may be one of the following:

cached	**shared**	**locked**	
uncached	**unshared**	**ro**	**rw**

The TriMedia dynamic loader ignores these properties, so that section properties only have meaning for code segment of type boot or dynboot. The specified old section name is the name of the section in the inputfile(s), regardless any previous section renamings on the command line.

-start *symbol*
: Specify start address by global symbol. A start symbol is required for code segments of type **boot**, **dynboot** and **app**, and optional for a **dll**. The value of the specified symbol can be retrieved after an explicit call to the dynamic loader.

-stored_endian [**el** | **eb**]
: Specify endianness in which the produced output file is stored. This is needed for code segments which are loaded by the dynamic loader, to minimize its code size and necessary run time data conversions. By default the stored endian is equal to the endian of the code segment itself, assuming that the dynamic loader executes on the TriMedia processor (as opposed to on a host).

-stub *file*
: Specify stub template file used by dynamic loader for redirecting calls to other code segments and trapping the dynamic loader when this is not loaded yet.

-symt
: Print symboltable to standard output. This option is only allowed when producing a memory image using option **-mi**.

-symtfile *file*
: Print symboltable to specified file. This option is only allowed when producing a memory image using option **-mi**.

-tm_freq *freq*
: Specify TM clock frequency. This option is only allowed when producing a memory image using option **-mi**. By default, the value 100000000 (100 MHz) will be used, with a generated warning.

-u *symbol* [**,** *symbol*]...
: Enter symbols as unresolved in the symbol table prior to linking.

-v	Verbose mode: print names of all linked files.
-V	Print **tmld** version.

List Construction by tmld

Through the **-chain** option, **tmld** can build, at link time, a linked list containing global variables defined in separate object files. This ability is useful when the elements of such a list become known only when linking an executable.

An occurrence of **-chain** *symbol* on the **tmld** command line has the following effects:

- Multiple global definitions of the specified symbol (in different object files) are allowed, provided that each definition is located in an initialized data section (e.g. **data**, **data1**, or renamed instances of these sections).
- Each next occurrence of the symbol (in a next object file on the **tmld** command line) is linked after the last encountered occurrence, by patching the address of the new occurrence into the first word of the last occurrence (see simple C example below). For correct functioning, this first word must have been reserved in the symbol as a link field.
- When linking a code segment (that is, executable or dynamic library), a new variable *symbol*_ptr is created, and initialized with the address of the first occurrence of the symbol.

As a simple example, the following command will result in creation of a static, null-terminated list of strings ["aap", "noot", "mies", wim"] whose first element is stored in a new variable **_my_string_list_ptr** (where a.o, b.o, c.o and main.o have been compiled from a.c, b.c, c.c and main.c, respectively):

```
tmld -chain _my_string_list a.o b.o c.o main.o

  typedef struct Element {
     struct Element *next;
     String          string;
  } Element;

  --- a.c ---
  Element    my_string_list= { Null, "aap" };

  --- b.c ---
  Element    x1            = { Null, "mies" };
  Element    my_string_list= { &x1,  "noot" };

  --- c.c ---
  Element    my_string_list= { Null, "wim" };

  --- main.c ---
  extern Element  *my_string_list_ptr;

  main (){
     Element *l= my_string_list_ptr;
```

 ©1999 Philips Semiconductors 10/08/99

```
      while (l != Null) {
         printf("%s ", l->string);
         l= l->next;
      }
   }
```

Note that the two-element sublist in b.c is correctly preserved.

Example

An example of using linker- constructed lists is in implementing C++ static constructor/destructor functions: unlike C, where static variables can only be initialized with compile time constants, C++ allows dynamic initialization of static variables. For example, variables can be initialized with the results of a function call, as in:

```
static int x= f();
```

This more liberal initialization means that code for evaluating all static initializers has to be executed before the call can be made to the main function of a C++ program.

A complication here is that the complete set of C++ modules that constitute a particular C++ executable, with their static constructor functions, is generally unknown before linking of that executable.

For instance, different executables can be linked from the object files main.o, a.o and b.o (compiled from main.C, a.C and b.C, respectively, see below): a1.out (printing a mere "hello"), a2.out (printing a decent "hello world"), and others. Merely including a.o, b.o, or both in either order, while linking an executable, causes the proper static constructor functions to be correctly picked up and called during initialization.

```
tmcc main.o -o a1.out a.o
tmcc main.o -o a2.out a.o b.o

   ----- a.C: ----
   static int i1= printf(" world");

   ----- b.C: ----
    static int i2= printf("hello");

   --- main.C: ---
   main(){}
```

The TriMedia SDE achieves this by means of symbol chaining, as follows:

1. For each C++ module, the C++ compiler generates a static function **__0CH_STCON_v** that initializes all static variables in that module. For instance, the C definition of this function for module a.C would be

   ```
   static void __0CH_STCON_v() { i1= printf(" world"); }
   ```

   Note that **__0CH_STCON_v** has static scope; each C++ module has its own private instance.

2. Also for each C++ module, the C++ compiler generates a static external variable **__CC_Init_List**, according to the C definition:

   ```
   void* __CC_Init_List[2]= { Null, (void*)__0CH_STCON_v };
   ```

   This variable is the one on which symbol chaining is to be applied; it consists of two addresses: one link field that will be filled in by the linker (see 3), and one pointer to the module's initialization function **__0CH_STCON_v**. Note that although each C++ module has its own instance, **__CC_Init_List** has external scope. Normally these multiple definitions would result in link errors, but defining **__CC_Init_List** to **tmld** as being a chain symbol (see below) will prevent these errors.

3. **__CC_Init_List** is a reserved chain symbol to **tmcc**/**tmld**. That is, when linking a number of C++ object files, **tmld** constructs a list of all encountered global **__CC_Init_List** variables by patching pointers to the next element in the list in each first **__CC_Init_List** field. The list order of these variables is the order of occurrence of their object files on the linker command line.

4. Finally, a global variable **__CC_Init_List_ptr** is produced by **tmld**, and initialized with a pointer to the head of the list. The result of steps 3 and 4 is a list of all **__CC_Init_List** variables, with a pointer to its first element placed in **__CC_Init_List_ptr**:

   ```
      (generated by tmld)      a.o:                 b.o
      __CC_Init_List_ptr     __CC_Init_List      __CC_Init_List      ...

                            +---------------+   +---------------+
          ----------------->|    ------------->|    --------------> ... -->
   Null
                            +---------------+   +---------------+
                            | __0CH_STCON_v |   | __0CH_STCON_v |
                            +---------------+   +---------------+
   ```

5. The C++ runtime library is able to find and call each **__0CH_STCON_v** via **__CC_Init_List_ptr** at startup, before calling **main()**.

Other Uses of Chain Symbols

Although primarily provided for dealing with static constructor functions in C++, chain symbols can also be used for creating a ‘drop in’ environment (for device drivers, for example).

Reserved Chain Symbols

The TriMedia SDK defines the following reserved chain symbols:

__CC_Init_List
: Symbol for internal SDK use, for dealing with C++ initialization and constructor calling.

__custom_boot, **__custom_driver**, **__custom_start**
: Symbols by means of which users can insert initialization functions at different stages of initialization of the executable:

__custom_boot	For defining initialization functions that are to be called after initialization of the TM-1 processor, and after setup of stack and heap, but before initialization of the host call interface. This level of initialization can be used for customizing TM-1 processor initialization.
__custom_driver	For defining initialization functions that are to be called after host call initialization, but before initialization of the dynamic loader. This level of initialization can be used for adding drivers for devices on which dynamic libraries are stored that are to be 'immediately' loaded by the current executable.
__custom_start	For defining initialization functions that are to be called after initialization of the dynamic loader, and after dynamic loading of all 'immediate' code segments, but before the call to the application's main function, and (in case of a C++ application) before the call to the first static constructor function.

Initializations to be added to any of the above levels must be installed by

1. Defining the initialization function of the following prototype:

   ```
   int init(void);
   ```

   The function should return a 0 if the initialization was successful, any other value else. Failing to return 0 causes immediate termination of the executable.

2. Defining an instance of the variable **__custom_boot, __custom_driver, or __custom_start**, as follows:

   ```
   Int __custom_xxxx[]= {0, (Int) init};
   ```

   For instance, the following installs an I/O driver (for attaching a flash file system to the runtime library, for example). Merely linking the corresponding object file to the application will result in driver installation at the appropriate time:

   ```
   static Int init( void ){

      return IOD_install_fsdriver(
                 RecogFunc, InitFunc, TermFunc,
                 OpenFunc, OpenDllFunc, CloseFunc,
                 ReadFunc, WriteFunc, SeekFunc, IsattyFunc, FstatFunc,
                 Null, StatFunc, SyncFunc, FSyncFunc, UnlinkFunc,
                 LinkFunc, MkdirFunc, UnlinkFunc, AccessFunc,
                 OpendirFunc, ClosedirFunc,
                 RewinddirFunc, ReaddirFunc
             ) == NULL;
   }

   Int __custom_driver[]= { 0, (Int)init };
   ```

Note

Termination functions, when required, can be installed by the initialization functions by means of function **atexit** from the ANSI library.

 © 1999 Philips Semiconductors 10/08/99

Chapter 12

TriMedia Execution Host Utilities

Topic	Page
Summary	136
TriMedia Commands	137

Summary

TriMedia development boards that support a PCI interface can be installed in a Windows host system so you can do development. This chapter gives summary information on the tools that make up the Windows execution host support.

The TriMedia Manager

The TriMedia Manager includes the following files and utilities:

TMMan.sys	The kernel-mode driver that provides the bulk of the TMMan functionality. This driver supports multiple boards.
TMMan32.dll	The user-mode Win32 DLL that provides the TMMan application programming interface to Win32 applications. This DLL simply calls tmman.sys for the TMMan functionality.
TMRun.exe	A command-line utility (Win32 console application) for downloading and running executables on the TriMedia processor. This program is also used by TMMon as the TriMedia console.
TMmpRun.exe	A multiprocessor version of TMRun, that enables multiprocessor cluster downloading on multiple TriMedia boards plugged into the system.
TMMon.exe	The TriMedia Monitor, an interactive shell for downloading and running programs on TriMedia. It is a Win32 console-mode application that provides a command-based interface. TMMon reads its input and writes its output via standard handles so the input to TMMon including command can be redirected from an input file.
TMCRT.dll	The TriMedia C run-time server. This module accepts requests from the target and serves them. The requests are Unix level-2 I/O calls generated by the executable program running on TriMedia. The server uses the TMMan messaging mechanism to communicate with the target.
TMMan.a	The target component of TMMan. This is a static library with which 'boot' applications on TriMedia are linked. This module provides the TMMan functionality on the target.
Driver.exe	This is a helper utility that is provided in order to install the kernel-mode driver in the system. This utility is required only during software installation or when the software is uninstalled.

TMMan Setup & Removal

The driver.exe program is used to setup the kernel mode driver. The installation script uses this utility to make the relevant entries required to auto start the kernel mode

 ©1999 Philips Semiconductors 10/08/99

driver. If for any reason the user needs to remove the kernel mode driver entry from the registry the driver utility can be used with the following parameters:

```
C:\> driver -or -ntmman
```

To Reinstall the driver again, use

```
C:\> driver -oi -ntmman -sa tmman.sys
```

To install the driver so that it does not start automatically, use

```
C:\> driver -oi -ntmman -sm tmman.sys
```

This will make the driver manual start instead of auto start.

After you have installed the NT kernel-mode driver in your system, the registry key has the following or similar values:

```
[HKEY_LOCAL_MACHINECurrentControlSet\Services\tmman]
"DisplayName"="tmman"
"ImagePath"=".sys"
"TYPE"=dword:00000001
"Start"=dword:00000002
"ErrorControl"=dword:00000001
```

The driver can also be configured to start in manual mode by changing the value of **Start** to **dword:00000003**. To start the driver manually, type the following at the command prompt:

```
C:\> net start tmman
```

To stop the driver manually type the following at the command prompt

```
C:\> net stop tmman
```

The kernel mode driver tmman.sys should be present in the **%system-root%\system32\drivers** sub-directory. All other files must be in the current directory or in the executable search path.

TriMedia Commands

Other TriMedia command or utility programs exist. Some, such as **tmsim**, are discussed in depth elsewhere. This chapter summarizes the TriMedia commands.

Command	Description	Page
tmgmon	GUI-based monitor	138
TMRun	Single-processor TMRun utility	140
TMmpRun	Multiprocessor version of TMRun	141

tmgmon

tmgmon is a GUI-based Win32 application that uses the TriMedia Manager Host API and provides an interactive user interface for downloading and running TriMedia executables on the TriMedia processor. All options can be accessed by selecting the option from the window. Scrollable views are provided for the trace and memory window to aid in debugging.

The GUI-based monitor program consists of the tmgmon.exe application along with the following programs:

tmman32.dll	TriMedia Manager User Mode 32 bit.
tmld.exe	TriMedia Image Relocator and Loader.
vtmman.vxd	TriMedia Virtual Device Driver.
tmcons.exe	TriMedia Console.
RPCServ.dll	Remote Procedure Call Server.
Msvcrt.dll	Microsoft Visual C++ 4.2 runtime library.
Mfc42.dll	Microsoft Visual C++ 4.2 MFC library.

Copy all of these files into a single directory.

Running the Software

From the Windows Explorer, double click the program tmgmon.exe. The main window will appear. It is divided into the following groups:

- Processor
- Code Download
- Redirection
- Memory
- Trace

At the top right corner are the minimize and close buttons. A dialog showing copyright and version number can be displayed by selecting the 'About tmgmon…' option from the system menu at the top left corner. The details of the various groups are given in the following sections.

Processor

In this group, tmgmon displays information about the TriMedia card. The displayed information includes the card number, processor, type, name and memory addresses. **tmgmon** has support for multiple boards, as used for multi-processor systems. Initially, board number 0 is selected. To select a different board, click the up or down button in the Num field.

Code Download

Enter the filename to be downloaded in the Filename field. You can either type the name, or select the file by clicking the Filename button. If there are any command line arguments, specify them in the Arguments field. The filename and arguments field have a memory of recently used files and commands. This can be accessed through the pull down menu tab at the right of each field. This memory is updated into the registry when you quit the program.

To load the program in TriMedia, click the Download button. To run the program, click the Go button. To stop the program, click the Stop button. The Stop button will reset the TriMedia and all its on-chip peripherals.

The Go option will first load the program in TriMedia if not already loaded and then run it. If a program is already running in TriMedia and you want either to run it again or to run a new program mentioned in the Filename option, the Go option will first stop the current program, load the new program and then run it.

Memory

Specify, in hex, the physical starting address of any memory location you want to display in the Address field and then click the Display button. The program will check whether the address you typed lies within the valid ranges. If it is in range, it will display a dump of memory starting from that location. It will display up to the limit or 100000 hex bytes whichever is smaller.

You can change the memory view to byte, word, or dword by selecting the appropriate radio button. However, MMIO can be viewed only as dword and clicking the byte and word radio buttons will not change the view.

ASCII values are also shown on the right side of the memory window.

For navigation in the edit window, you can use the following keys:

Key	Action
Cursor Up	Move up one line.
Cursor Down	Move down one line.
Cursor Left	Move left one character.
Cursor Right	Move right one character.
Home	Start of line.
End	Ending byte, word or dword of line.
Page Up	Previous page.
Page Down	Next page.
Ctrl Home	Start of dump.
Ctrl End	End of dump.

Mouse operations and scroll bars also work.

To change the value at any memory location, move cursor over the displayed text and type the new value. The new memory value will actually be written to the memory when

you (1) enter the complete value and press enter or (2) move to another memory location or (3) select another window like the main window. This ensures that incomplete values do not go into memory as you are entering the new value.

To copy a particular region of the memory dump to the clipboard, first select it by dragging the mouse over the area, keeping left button pressed. Then press Ctrl-C. As an alternative, you can drag the mouse over the area and then click right mouse button for a pop-up menu from which you select Copy.

At the top right corner of the memory window are buttons to minimize, maximize, and close the window. You can also resize the window by dragging the bottom right corner.

Trace

The contents of the DP buffer can be dumped into the trace window. At the bottom of the trace group, there is a box to specify a file into which the trace should be dumped, and a check box to enable the file dump. When this is enabled, the contents of the DP buffer are placed in that file, and not in the trace window.

The Dump DP button at the lower left corner of the group triggers the dump. It does this over the PCI bus without disturbing the operation of the program on TriMedia. The TriMedia's memory is searched for a magic string that identifies the start of the DP buffer. This method was adopted to allow you to dump the contents of the trace buffer even after a warm reset due to a system crash.

You can select a region of text in the Trace window and delete, cut, or copy it using keyboard shortcuts or a pop-up menu activated from a right mouse click.

TMRun

TMRun launches and monitors TriMedia programs interactively or from a batch file. It is a command-line tool.

Syntax

TMRun [-d*DSPNumber*] [-w*WindowSize*] [-b] [-s] *ExecutableImage arg…*

Options

-b	Run in batch (or non-interactive) mode. In this mode, **TMRun** does not print the exit code or any other status information to the console and it does not wait for a keypress before exiting.

-s	This is a special mode in which TMRun does not load and execute a boot image. It just waits for C run-time requests from the target and serves them. This mode is useful if you need to load and execute boot images programmatically but do not want to initialize the C run-time server (TMCRT.dll) in its own process context.

 ©1999 Philips Semiconductors 10/08/99

For example, **TMMon** spawns **TMRun** with this switch to serve requests from the target.

-w*WindowSize*
: Controls the number of lines in the **TMRun** window. Under Windows NT, this parameter is not really required, because the size of the console can be set in the property menu of the console window.

-d*DSPNumber*
: Indicates on which processor the executable is to run. The default is 0.

TMmpRun

TMmpRun is a multiprocessor version of the **TMRun** utility described previously. As does **TMRun**, this command-line tool launches and monitors TriMedia programs interactively or from a batch file or script.

Syntax

TMmpRun [-w*WindowSize*] [-b] -exec *executable arg* ... [-exec *executable arg* ...]...

Options

-b
: Run in batch (or non-interactive) mode. In this mode, **TMmpRun** does not print exit code and other status information to the console and it does not wait for a keypress before exiting.

-w*WindowSize*
: Controls the number of lines in the **TMmpRun** window. Under Windows NT, this parameter is not really required, because the size of the console can be set in the property menu of the console window.

-exec *ExecutableImage arg ...*
: Specifies the executable image and the parameters that must be passed to it. All parameters up to the next -exec flag belong to the executable image.

  You can specify more than one executable image. Each one corresponds to a processor in your system. The number of -exec images cannot exceed the number of processors in your system.

  The -exec options are ordered: the *n*th -exec image corresponds to processor *n*–1.

 © 1999 Philips Semiconductors 10/08/99

Chapter 13

The TriMedia Simulator

Topic	Page
Introduction to Machine-Level Simulation of TriMedia Processors	144
The Simulated Architecture	146
Interrupt and Exception Handlers	160
Data Layout	161
Program Debugging	161
Operating System Emulation	162
Video-Out Support	163

Introduction to Machine-Level Simulation of TriMedia Processors

tmsim simulates the execution of a program on a TriMedia processor. The program is taken at the compiled and linked object file level, as illustrated in Figure ?? on 1. The properties of the simulated machine are determined by the machine description file.

```
tmsim foo faa fee
cuv] library [ object ... ]
o abctmld -eb a.o b.o libc.a c.o
.o a.o b.o c.o
```

The invocation above reads the application program from the object file **foo**, reads the machine description file tm1.md from the installation directory, and starts executing the application program at the start address specified in the object file. The strings **faa** and **fee** are passed as arguments to the simulated program.

tmsim serves the following purposes:

- Full behavior and cycle-accurate DSPCPU simulation as described in the machine description file.
- Full behavior and cycle-accurate memory and cache simulation if required.
- Performance-accurate architecture simulation for the application programmer.
- Cycle-accurate architecture simulation for comparison against Verilog code.

IMPORTANT
tmsim's stack dumping mechanism (the stackdump command) does not work for multiprocessing or stripped binaries.

 ©1999 Philips Semiconductors 10/08/99

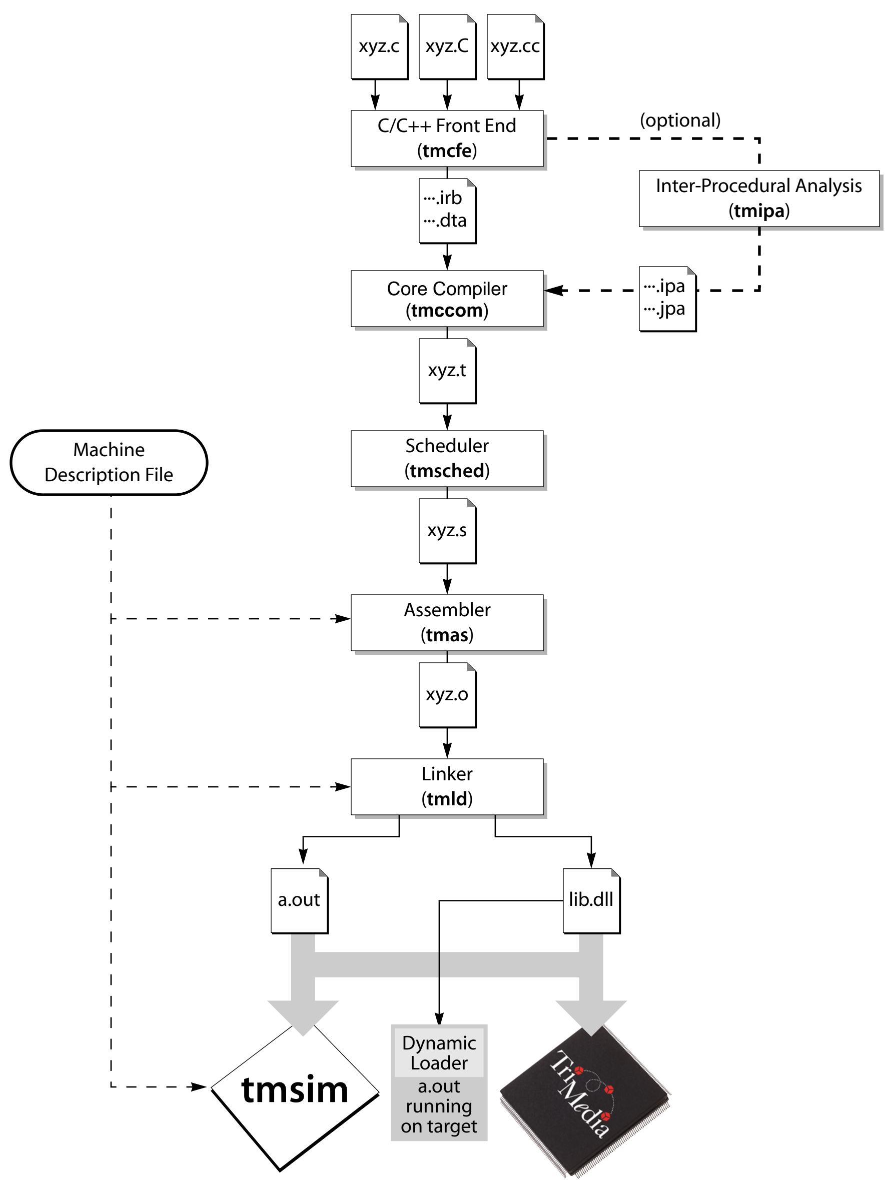


Figure 27 The Position of **tmsim** in the TCS

The Simulated Architecture

The system simulated by **tmsim** is illustrated in the following figure:

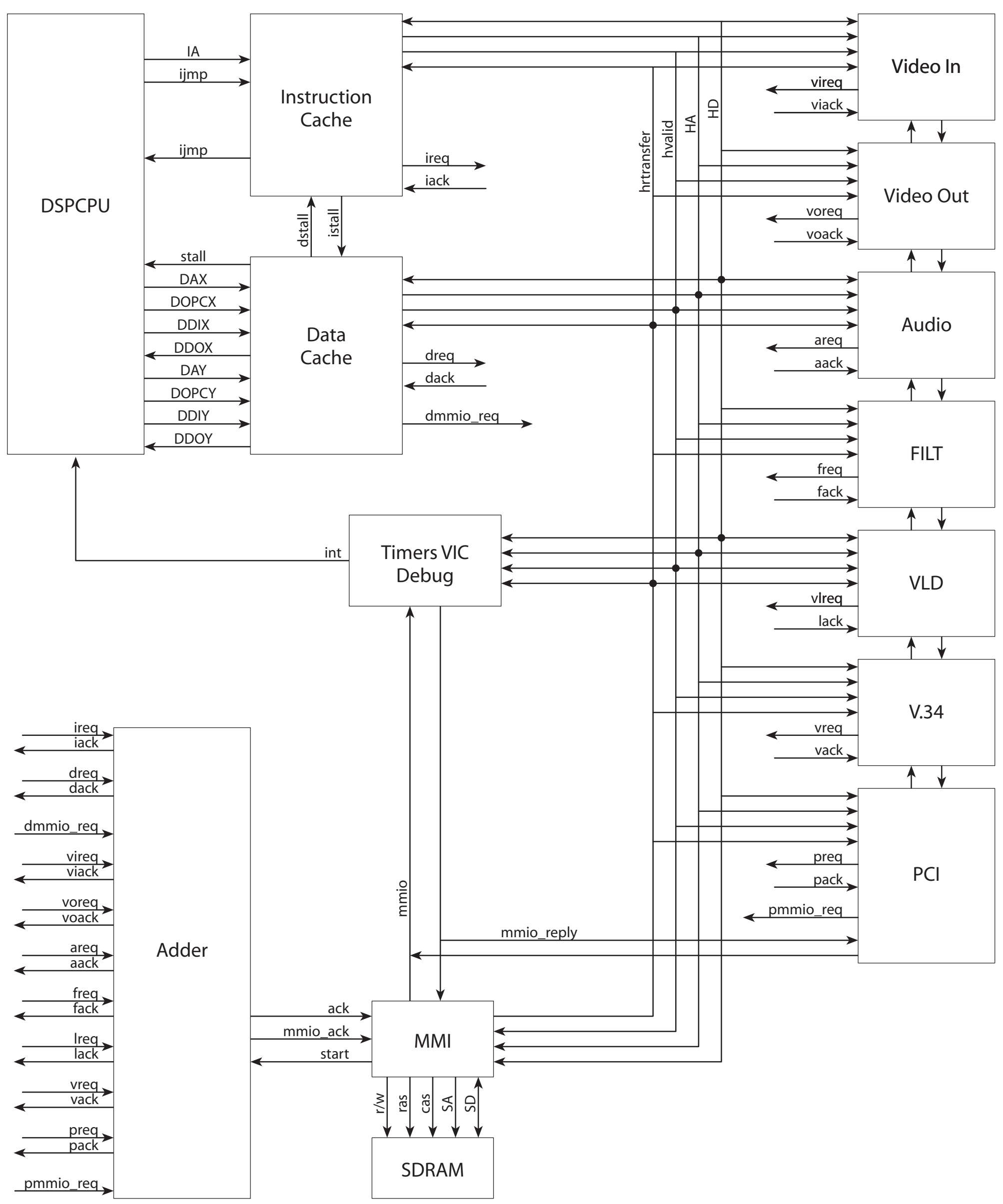


Figure 28 TriMedia System Overview

 © 1999 Philips Semiconductors 10/08/99

The system simulated by **tmsim** consists of the following:

- DSPCPU — The DSPCPU supports all architected registers, operations and system functions as described in the CPU Architecture chapter of the appropriate TriMedia data book.

  The number of registers simulated is determined by the machine description file. The functional unit types and operation latencies are determined by the machine description file.

- Memory system — By default, the real TriMedia memory system is simulated. If required, the memory system can be simulated as if only SDRAM were present and accesses to the SDRAM will not cause any stalls of the CPU. This behavior is enabled by using the –**nomm** command-line option. The simulated memory system includes:

  — Instruction cache
  The full instruction cache as described in the Cache Architecture chapter of the data book is implemented.

  — Data cache
  The full data cache as described in the Cache Architecture chapter of the data book is implemented.

  — Highway with arbiter
  TriMedia contains an address/data highway that connects the two caches, the peripherals and the memory interface. It is modeled in the simulator.

  — Main Memory Interface and SDRAM
  The simulator supports a default SDRAM data aperture of 8 MByte. It is initially located at address 0x100000 (1M). The size and location of the SDRAM can be changed through command-line options. Instructions are read from the object file and stored in the SDRAM. Instruction memory addresses are maintained and can be used in DPC, SPC, jump immediate instructions and instruction breakpoint generation. The initialized data is read from the object file and stored in the SDRAM. The full functionality of the MMI and SDRAM are modeled in the simulator (including the possibility to have the memory and CPU run at various clock relationships).

- System Peripheral Devices
  The simulator handles devices that are controlled through memory mapped device registers, sometimes referred to as MMIO. The MMIO aperture is initially located at address $EFE0,0000 (see also the CPU Architecture chapter of the data book). It can be relocated in memory by assigning to the variable MMIO_base. Devices can autonomously access data memory and can cause CPU interrupts. Devices included in the simulator are:

  — Vectored Interrupt Controller
  The VIC manages the setting and clearing of interrupts, acknowledgment generation and interrupt and exception prioritizing and handling. Interrupt priorities and modes (edge/level) can be specified. Interrupt vectors are fully programmable. A complete description is available in the data book.

— 4 timers/counters
Each timer/counter increments its *value* until *modulus* is reached. On the clock tick that would load *modulus*, the value resets to zero instead and an interrupt request is generated. Counting continues as long as the *run* bit is set. The current counter/timer model supports five sources: internal clock, internal clock divided by a prescale, external clock, data breakpoint events and instruction breakpoint events. For a detailed description, see the data book.

— Video In (VI)
The digital video input signal processing unit is simulated accurately according to the architecture specified in the data book. (Note that timing accuracy is not supported.)

— Video Out (VO)
The video output unit is simulated, with some exceptions, according to the specification found in the data book. Support for the following will be provided in a future release: YUV 4:2:2 interspersed mode, YUV 4:2:0 mode, upscaling, overlaying, the YTHR interrupt, and the use of the Y/U/V_DELTA registers.

— Audio In (AI). the audio input signal processing unit is simulated accurately according to the architecture specified in the data book. (Note that timing accuracy is not supported.)

— Audio Out (AO). The audio output unit is simulated according to the specification found in the data book.

— Image Co-Processor (ICP). Preliminary support for the ICP unit is provided. See the data book for a specification.

— PCI Interface. The interface with the PCI bus is simulated (see the architectural specification in the data book).

— JTAG. The IEEE 1149.1 (JTAG) standard on-chip controller facilitates monitoring and modification of a running system. This device is described in the data book.

— SSI. The V.34 Serial Synchronous Interface is simulated. See the data book for a description of this device.

— VLD. The Variable Length Decoder performs Huffman decoding for MPEG/MPEG2. This device is described in the data book.

— Debug Support. Instruction and/or data breakpoints can be defined. When a match is found, an event is generated that can be used as the clock input to one of the timers. After counting a number of events, the timer generates an interrupt request. For a detailed description, see the data book.

— Cache, memory and CPU control registers. All registers are supported.

— Instruction cache tag and status bits can be read through the MMIO space as described in the Cache Architecture chapter of the data book.

— The System I/O space is not currently supported.

 ©1999 Philips Semiconductors 10/08/99

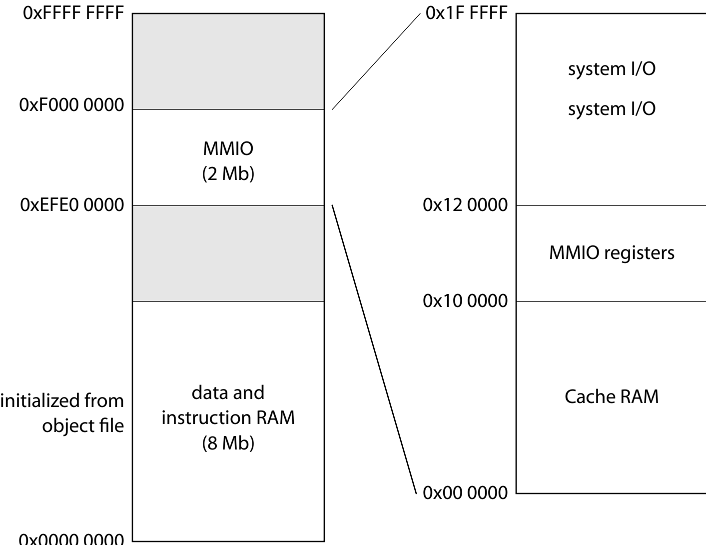


Figure 29 Memory Map of Instruction and Data Memory

For a complete description of data memory, see the appropriate data book.

Command Syntax

This is a summary of the machine level simulator command syntax. Details follow.

tmsim [*option* ...] [*machine* | *machine***.md**]] *object* [*arg* ...]

tmsim simulates a compiled and linked executable object file that would run on a TriMedia processor. **tmsim** simulates the DSPCPU as described in the machine description file *machine*. Simulation includes:

- The PCSW, DPC, SPC and CCCOUNT registers.
- The MMIO space, memory and cache control registers.
- The vectored interrupt controller.
- Four timers and debug support (instruction and data breakpoints).

Audio input/output, video input/output, SSI, JTAG, VLD, and ICP peripherals are simulated only when enabled through a command-line option.

tmsim also (1) provides operating system support, (2) catches and reports exceptions, and (3) includes commands that are useful for debugging. **tmsim** can print useful information about the state of the simulator and store trace information in a file.

tmsim can run a simulation either in batch mode or in interactive mode (with the **-i** option). When run in batch mode without specifying the **-batchfile** option, **tmsim** executes the program in the specified object file until completion or until an error occurs. In interactive mode, you can enter commands to control the simulation. The same commands can be specified in a batch file and executed by invoking **tmsim** with the **-batchfile** option.

tmsim includes commands which set up the simulation, run the simulation and allow interactive debugging, print information, deal with batch mode execution, and gather performance information and statistics. In interactive mode, you must enclose strings and filenames in double quotes ("..."). Labels need not be quoted. Numbers and addresses can be in decimal (default) or in hexadecimal format (preceded by 0x).

Command Line Options

tmsim recognizes the command-line options described below. Abbreviations for some options are also available, as shown.

-audioin *file* | **-ai** *file*

Simulates audio input, taking input from *file*. The input is a text file having one 16-bit hex number per line, where each number represents the pin states at each cycle.

Note that bit 13 (little-endian) has special significance: when the simulator sees this bit set, it remembers the point in the data file. When end-of-file is reached, the simulator returns to that point and continues reading, allowing a continuous cycle of data after the initial set-up.

The pin assignments are

```
AI_WS    0x8000
AI_SCK   0x4000
AI_SD    0x0001
```

By default, **tmsim** does not simulate audio input.

-audioout *file* | **-ao** *file*

Simulates audio output, taking input from *file*. The input is a text file having one 16-bit hex number per line, where each number represents the pin states at each cycle.

Note that bit 13 (little-endian) has special significance: when the simulator sees this bit set, it remembers the point in the data file. When end-of-file is reached, the simulator returns to that point and continues reading, allowing a continuous cycle of data after the initial set-up.

The pin assignments are

```
AO_WD      0x8000
AO_SCK     0x4000
AO_OSCLK   0x4000
AO_SD      0x0001
```

By default, **tmsim** does not simulate audio output.

 ©1999 Philips Semiconductors 10/08/99

-batchfile *file* | **-bf** *file*

Reads and executes the simulator commands from the batchfile *file*.

By default, **tmsim** does not read a batch file.

-cachetracefile *file* | **-ctf** *file*

Generates a cache trace and write it to *file*.

By default, **tmsim** generates no cache trace.

-clockfrequency *n* | **-cf** *n*

Sets the external clock frequency to *n* MHz. The CPU and memory clock frequencies are derived from the external clock frequency according to the setting of the PLL_ratios register.

By default, both the CPU and memory clock frequency are twice the external clock frequency. And by default, the external clock frequency is 50 MHz.

-cycles1 *n1* | **-c1** *n1*

Print intermediate simulator information after every *n1* cycles (if the **-v** option is also specified non-zero).

The default is 1000000 (10^6).

-cycles2 *n2* | **-c2** *n2*

Interrupts execution after $n1 \times n2$ cycles. When **tmsim** reaches this limit, it prints "Aborting: Cycle limit exceeded" and exits.

The default is 100000 (10^5).

-dse Prints a warning message when the simulator encounters a data segmentation error.

By default, **tmsim** prints no warning.

-eb Starts the simulation in big-endian mode (i.e., the value of the PCSW.BSX bit is 0). The default is big-endian.

-el Starts the simulation in little-endian mode (i.e., the value of the PCSW.BSX bit is 1). The default is big-endian.

-h Prints help information.

-i Enters interactive mode after reading the machine description file and the object file. You can issue commands at the interactive prompt.

By default, **tmsim** is not interactive.

-imagecoprocessor | **-icp**

Simulates the image coprocessor.

By default, **tmsim** does not simulate the image coprocessor.

-intlat Prints interrupt latency statistics in the report file.

By default, **tmsim** does not print these statistics.

-jtag *file* | **-jt** *file*

Simulate the JTAG unit, taking input from *file*. The input is a text file having one 16-bit hex number per line, where each number represents the pin states at each cycle.

Note that bit 13 (little-endian) has special significance: when the simulator sees this bit set, it remembers the point in the data file. When end-of-file is reached,

the simulator returns to that point and continues reading, allowing a continuous cycle of data after the initial set-up.

The pin assignments are

```
JT_TDO   0x0008
JT_TDI   0x0004
JT_TMS   0x0002
JT_TCK   0x0001
```

By default, **tmsim** does not simulate the JTAG unit.

-machinefile *file* | **-mf** *file*

Reads the machine description from *file*. For the default machine description file, **tmsim** searches for these files, in this order:

$TCS_MACHINE, if set

$TCS /lib/tm1.md, if $TCS is set

/usr/local/tcs/lib/tm1.md (or lib/tm1.md under the install directory).

-memorybase *n* | **-mb** *n*

Locates the SDRAM data memory starting at address *n*.

The default is 1048576 (0x100000 or 220).

-memorymodel | **-mm**

Simulates the TriMedia cache and memory model.

By default, **tmsim** does not simulate the cache and memory model.

-memorysize *n* | **-ms** *n*

Sets the size of data memory to *n* kilobytes.

The default is 8192 kilobytes, i.e., 8 Mb.

-reportfile *file* | **-rf** *file*

Writes simulator information and error messages to *file*.

By default, **tmsim** writes to your terminal (or PC screen).

-ssi *file*

Simulates the SSI, taking input from *file*. The input is a text file having one 16-bit hex number per line, where each number represents the pin states at each cycle.

Note that bit 13 (little-endian) has special significance: when the simulator sees this bit set, it remembers the point in the data file. When end-of-file is reached, the simulator returns to that point and continues reading, allowing a continuous cycle of data after the initial set-up.

The pin assignments are

```
V34_IO2      0x0020
V34_IO1      0x0010
V34_RXDATA   0x0004
V34_RXFSX    0x0002
V34_CLK      0x0001.
```

By default, **tmsim** does not simulate the SSI.

 ©1999 Philips Semiconductors 10/08/99

-statfile *file* | **-stf** *file*

Generates execution statistics (per decision tree) and prints them to *file*.

By default, **tmsim** generates no execution statistics. The statistics file produced by this option is an ASCII file that contains a line describing each decision tree executed during program simulation.

Each line contains nine fields:

Tree name
number of executions
number of instruction cycles.
number of instruction cache stall cycles.
number of data cache stall cycles.
number of cache copybacks.
number of cache conflicts.
number of operations.
number of useful operations.

-status

Returns an exit status (indicating the success or failure of **tmsim** itself, rather than the exit status of the simulated program).

By default, **tmsim** returns the exit status of the simulated program.

-t Times instruction execution and reports the average number of cycles simulated per second.

By default, **tmsim** performs no instruction timing.

-tracefile *file* | **-tf** *file*

Enables trace mode and write to the given *file*.

By default, **tmsim** performs no tracing.

-trappci

Traps all application program accesses to the PCI bus.

By default, **tmsim** performs no trapping.

-v Verbose mode: prints header, trailer and intermediate cycle reports.

By default, **tmsim** prints nothing.

-V Prints **tmsim** version information.

-videoin *file* | **-vi** *file*

Simulates video input, taking input from *file*. The input is a text file having one 16-bit hex number per line, where each number represents the pin states at each cycle.

Note that bit 13 (little-endian) has special significance: when the simulator sees this bit set, it remembers the point in the data file. When end-of-file is reached, the simulator returns to that point and continues reading, allowing a continuous cycle of data after the initial set-up.

The pin assignments are

```
VI_DVALID   0x8000
VI_CLK      0x4000
VI_DATA     0x03FF  (bits 0-9)
```

By default, **tmsim** does not simulate video input.

-videoout *file* | **-vo** *file*

Simulates video output, taking input from *file*. The input is a text file having one 16-bit hex number per line, where each number represents the pin states at each cycle.

Note that bit 13 (little-endian) has special significance: when the simulator sees this bit set, it remembers the point in the data file. When end-of-file is reached, the simulator returns to that point and continues reading, allowing a continuous cycle of data after the initial set-up.

The pin assignments are

```
VO_CLK   0x0400
VO_IO2   0x0200
VO_IO1   0x0100
VO_DATA  0x00FF  (bits 0-7)
```

By default, **tmsim** does not simulate video output.

-vld Simulate the variable length decoder.

By default, **tmsim** does not simulate the variable length decoder.

-wx Normally if warnings are generated, **tmsim** exits with a negative exit status. This option will cause **tmsim** to exit with status = 0 even if warnings were generated.

-xio *file*

Simulate the PCI-XIO unit as a RAM, taking initial content of the RAM from *file*.

By default, **tmsim** does not simulate the PCI-XIO unit.

Interactive debug commands

The following sections present interactive debug commands, in several categories.

Setup Commands

These are the available **tmsim** setup commands:

clockfrequency *freq*

Sets the target clock frequency (of the external clock input pin) to *freq* MHz. The default is 50 MHz. You can specify *freq* with a fractional part, e.g., 97.6 MHz.

cycles1 *n1* | **c1** *n1*

Sets the number of cycles (after which the simulator prints a short message) to *n1*. The default is 1000000 (10^6).

cycles2 *n2* | **c2** *n2*

Interrupt execution after $n1 \times n2$ cycles, printing the message “Aborting: cycle limit exceeded.” This command is useful for batch mode simulation of programs with a possible infinite loop. The default is 100000 (10^5).

 ©1999 Philips Semiconductors 10/08/99

datasegmentationerror [**ON** | **OFF**] | **dse** [**ON** | **OFF**]

Determine whether **tmsim** interrupts simulation when it detects a segmentation error in data memory addressing. ON means interrupt, OFF means do not interrupt. The default is OFF.

echo [**ON** | **OFF** | *string*]

Either turn command echoing ON or OFF or echo the given *string*. This command can be useful for batch file execution. The default is OFF.

Simulation and Debugging Commands

break *id* | **b** *id*

Define a breakpoint at the location specified by symbolic address *id* in the simulated program.

break *address* | **b** *address*

Define a breakpoint at the absolute *address* in the simulated program. **tmsim** stops execution before executing the instruction at the specified address. Any number of breakpoints can be set.

Examples:

```
break _main 0
b _printf 2
b 1275
b 0x10 4
```

delbreak *id* | **db** *id*

Remove the breakpoint at the location given by the symbolic address *id*.

delbreak *address* | **db** *address*

Remove the breakpoint at the absolute *address*.

clearbreak | **cb**

Remove all instruction breakpoints.

continue | **c**

Continue execution until an exception occurs, the cycle limit is exceeded, or the program exits normally.

end | **q**

End the **tmsim** session.

reset Reset the machine. The command resets the program counter to the start address of the simulated program. Note, however, that this command does reinitialize any variables. Specifically, it does not reset ANSI C library dependent behavior (such as dynamic memory allocation). Your program initialization code might do this. Otherwise, you must exit and restart **tmsim**.

run *n* | **r** *n*

Run *n* cycles or until an exception occurs.

runtilljmp | **rj**

Run until the next (interruptible or non-interruptible) jump operation is executed. The jump can be successful or unsuccessful.

runtilljmptaken | **rt**

Run until the next successful (interruptable or non-interruptable) jump operation is executed.

step | **s**

Single-step the program (execute one clock cycle).

stepinstruction | **si**

Single-step the program by executing one instruction.

watchread *low high* | **wrd** *low high*

Causes a read operation in the address range *low* to *high* to generate a simulator interrupt. You can disable watchread by specifying a high address that is lower that the low address.

watchwrite *low high* | **wwt** *low high*

Causes a write operation in the address range *low* to *high* to generate a simulator interrupt. You can disable watchwrite by specifying a high address which is lower that the low address.

writemem *id number format* | **wm** *number format*

Write *number* to the data memory location specified by the label *id*. Specify the number in hex. The *format* can be one of the following:

B byte, 8 bit integer

HW halfword, 16 bit integer

W word, 32 bit integer

writemem *address number format* | **wm** *address number format*

Write *number* to the given data memory *address*. Specify the number in hex. The *format* can be one of the following:

B byte, 8 bit integer

HW halfword, 16 bit integer

W word, 32 bit integer

writereg [*reg* | **PCSW** | **SPC** | **DPC**] *number* | **wr** [*reg* | **PCSW** | **SPC** | **DPC**] *number*

Write *number* to register number *reg* or to the specified register. Specify the number in hex. The number is always treated as a 32-bit value.

Information-Printing Commands

dumpinstructions [*low high*] *file* | **di** [*low high*] *file*

Dump instructions to *file*. If you specify *low* and *high*, **tmsim** dumps only instructions in the address range *low* to *high*. If you specify no address range, **tmsim** dumps the complete instruction memory (as derived from the object file). Specify addresses in hex.

The dump file contains one instruction per line. The format for each operation is:

opcode	bits 67–60
source register 1	bits 59–53
source register 2	bits 52–46
guard register	bits 45–39

 ©1999 Philips Semiconductors 10/08/99

destination registerbits 38–32

modifier bits 31–00

where the LSB is bit 0 and the MSB is bit 67.

For example, the command

```
di 20 60 "test"
```

produces:

```
startaddress 48
exitaddress 4026531840

stackorigin 6912
0000000@14

0e0a000fc000000002008000fd000000305f000007e000000071f0410080fffffff5
f000007f00000012

0900100800000000059f9fc080000000000900000890000000001f1210080ffffffd1
f0610080ffffffe

09001408a0000000009001808b000000000901f408400000000001f1610080ffffffb1
f1410080ffffffc
```

dumpmemory *file* | **dm** *file*

Dump data memory to *file*. The format used is suitable for reading by Cadence CAD tools. Addresses are written every 256 bytes in the following format:

@address

Data is written in hexadecimal format, one word per line.

help | **h**

Print a quick summary of the most commonly used commands.

pdcs *number*

Print the data cache set *number*, including all tag, status and data information.

pics *number*

Print the instruction cache set *number*, including all tag, status and data information.

printbreak | **pb**

Print all break conditions.

printsegments | **psg**

Print information about the instruction and data segments loaded from the object file.

printinfo

Print information on the simulation parameters.

printinstructions *address* | **pi** *address*

Print the instruction at *address* in symbolic form.

printinstructions *low high* | **pi** *low high*

Print the instructions in the address range *low* to *high*, in symbolic form.

printmachine | **pma**

Print information regarding the state of the machine being simulated. This command prints stack and heap addresses, return address, register contents and program counter.

printmemory *id format*
Prints the contents of data memory at the address of label *id*. See the *format* description following.

printmemory *address format*
Prints the contents of one data memory *address*. See the *format* description following.

printmemory *low high format* | **m** *low high format*
Prints the contents of data memory in the address range *low* to *high*. The *format* indicates how the memory locations are interpreted:

- **B** Byte (8-bit integer).
- **HW** Halfword, 16-bit integer).
- **W** Word, 32-bit integer.
- **A** ASCII character.
- **S** Null-terminated ASCII string. (The string format prints the string until either the null character or the high address is reached.)

printreg *n* | **PCSW** | **DPC** | **SPC** | **pr** *n* | **PCSW** | **DPC** | **SPC**
Prints the contents of register *n* or the specified register.

printfreg *n* | **prf** *n*
Prints the contents of register *n* in floating-point notation.

printregs | **prs**
Prints the contents of all registers in the register file.

printsymbol *address* | **ps** *address*
Prints the global or local symbol with the closest address preceding the specified *address*. The *address* can be of a variable or of an instruction.

printsymbol *id* | **ps** *id*
Prints the address of the given label or identifier *id*. The label can be global or local.

printsymbols [*file*] | **pss** [*file*]
Print to *file* all labels in the global and local symbol tables with their corresponding addresses. If you specify no file, the tables are printed to the current report file (which might be your terminal in interactive mode).

stackdump | **sd**
Print the subroutine call stack of the simulated program.

Batch Mode and Source Files

All commands can be used in batch mode or from source files. For example, the command

```
tmsim -bf batch.cmd test5.md opdiag
```

executes the commands from the file batch.cmd after initializing itself from the given machine description file test5.md and the object file opdiag. The following extra commands are significant for executing batch files and switching between interactive and batch mode:

 ©1999 Philips Semiconductors 10/08/99

batchmode
: Switch back to executing commands from the batch file after an interactive command in the batch file has switched to interactive mode. This command is only valid if **tmsim** has entered interactive mode from batch execution.

control
: Every batch file must start with the control command to be recognized as a legal batch file.

interactive
: Switch to interactive mode. This command is valid only in a batch file.

source *file*
: Execute commands from *file* as if you were typing them. The source file need not start with the control command.

Trace Mode and Performance Statistics

tmsim includes a trace mode which provides information about the execution of the simulated application program. The commands presented here address tracing and printing trace information.

ctracefile *file* | **ctf** *file*
: Write a cache trace to *file*.

reportfile *file* | **rf** *file*
: Write simulation information to *file*. **tmsim** will write all information requests as well as simulator error and information messages. If *file* is the null string (""), **tmsim** writes information to the standard error stream. The use of a specific report file keeps the standard output and standard error streams available for output from the application program.

  The simulator writes to the terminal by default.

stats *file*
: Write execution statistics for each decision tree to *file*. The following information is available:

  number of times each tree is executed (execs),

  execution cycles (instc) not counting stalls/conflicts,

  total issued operations (isopers) and

  total executed operations (exopers) excluding unsuccessful guarded operations.

  Furthermore, if you have requested the **-mm** option, the following are also listed:

  instruction cache stall cycles (istallc),

  data cache stall cycles excluding conflicts (dstallc),

  data cache copybacks (cpbacks) and

  data cache bank conflicts (cnflctc).

trace [**on** | **off** | **ns**]
: Enable trace (on), disable trace (off), or enable trace only for non-stall cycles (ns). Trace output occurs after the execution of each instruction. **tmsim** writes the trace

information to the report file. If combined with the **tct** or **tcf** command, symbolic information is printed for every instruction address.

The default is off.

tracecalls functions | tcf

Print a trace of all functions executed. If combined with the **trace on** command, symbolic information is printed for every instruction address.

By default, **tmsim** does not trace functions.

tracecalls trees | tct

Print a trace of all trees executed. If combined with the **trace on** command, symbolic information is printed for every instruction address.

By default, **tmsim** does not trace trees.

tracecalls off | tc off

Turn off function and tree tracing. By default, **tmsim** does not trace functions or trees.

vtracefile *file*

Enable Verilog tracing to *file*.

By default, **tmsim** does no Verilog tracing.

Interrupt and Exception Handlers

tmsim has the capabilities to simulate programs that respond to interrupts and exceptions. The mechanism in the current version of **tmsim** is fully compatible with the architectural definition of the TriMedia chip as described in the data book.

The interrupt and exception handler routines need to manipulate the PCSW and interrupt devices carefully to ensure a coherent handling mechanism. As an example, the interrupt handler should save the PCSW and DPC registers and disable further interrupts. It should also acknowledge the interrupting device before re-enabling interrupts. If nested interrupt handling is desired, a stack of PCSW and DPC registers may need to be built. Refer to the data book for details on interrupt/exception handler architecture aspects.

Contrarily to interrupt handlers, exception handlers cannot be written as procedures in C in the current release, since the SPC is not consistently saved. They should be written in decision tree (.t) intermediate code.

 ©1999 Philips Semiconductors 10/08/99

Data Layout

The **tmsim** simulator implements the register usage and data memory layout conventions in Figure 30.

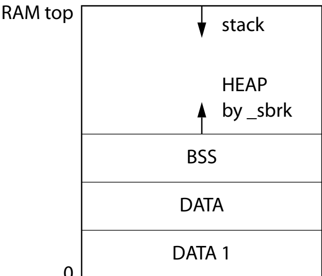


Figure 30 Default Data Layout

Program Debugging

tmsim can be started in interactive mode using the **-i** option. In interactive mode, low level debug support is provided. This support is provided primarily for debug by non-casual users.

Operating System Emulation

tmsim can be used in two fundamentally different ways:

- To execute a single program to completion.
- To load a set of processes and a real-time kernel, such as pSOS, and simulate the cooperation of such a set of processes for a given amount of clock cycles.

Simulations of the first nature execute a single program which may perform standard C library calls. These calls are executed by the standard C library that is linked in at the machine code level. This library in turn is built on a limited set of primitive handlers for level 2 UNIX system call support. These handlers are pre-loaded into simulated memory by the simulator, and execution of these decision trees leads to internal simulator actions that map to the underlying host system. The list of level 2 handlers supported is:

_close _fstat _isatty _lseek

_link _mktemp _open _read

_unlink _write gentenv time

If the application program exits with a nonzero exit status, **tmsim** will exit with that same exit status.

Simulations of the second nature truly run on the bare hardware, and do not use the built-in handlers described above. Memory management and I/O is fully handled by the real-time kernel and the simulated I/O devices. We expect to release special documentation on such simulations later.

Performance Analysis Support: The tmsim Statfile

The **tmsim** statfile is an ASCII file that provides a detailed view of the execution of an application. Every line in the statfile file corresponds to one of the decision trees in the program and contains nine fields that describe cache effects on program execution speed:

- Tree name
- Number of executions
- Number of instruction cycles
- Number of instruction cache stall cycles
- Number of data cache stall cycles
- Number of cache copybacks
- Number of cache conflicts
- Number of operations
- Number of useful operations

 ©1999 Philips Semiconductors 10/08/99

Generally, developers do not need this detailed of a performance analysis and are satisfied with the information provided by the performance analysis procedures outlined in the section describing the tmprof tool.

For those developers who require detailed performance information, this section describes the **tmsim** statfile format. The statfile is named and generated by using the **tmsim** command line form:

```
tmsim -statfile <statfile name> <executable file>
```

You can also generate a comparative statfile on the hardware by linking with the profile library using **tmprof -genstat**. For more information, refer to Miscellaneous Options, starting on page 88 of Chapter 5 of Book 4, Part A.

WARNING
To avoid generating false hits on blocks that never get executed, do *not* use the **-bfoldcode** linker option when generating a statfile .

Video-Out Support

Video-out is supported.

 © 1999 Philips Semiconductors 10/08/99

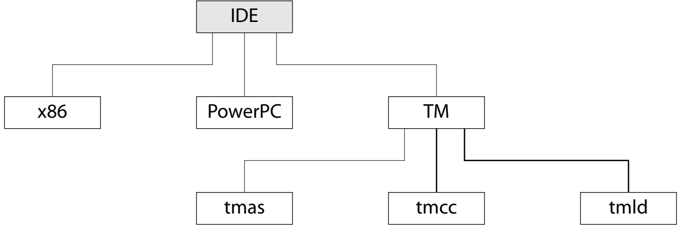
Chapter 14

Using the TriMedia CodeWarrior Plugins

Topic	Page
Overview	166
Installing the TriMedia CodeWarrior Plugins	166
Implementation Notes	167
Specifying TriMedia CodeWarrior Settings	168

Overview

The TriMedia CodeWarrior Plugins provide a TriMedia compiler and linker for the CodeWarrior Integrated Development Environment (IDE) and allow TriMedia developers to use the Metrowerks CodeWarrior IDE for developing TriMedia applications. The TriMedia plugins provide the same functionality as the following command line tools: **tmcc**, **cpp**, **tmccom**, **tmsched**, **tmas**, **tmld**, **tmar**, **tmstrip**, and **tmsize**.



Note
This document describes how to use the TriMedia-specific components of the CodeWarrior IDE. For information on how to use CodeWarrior IDE, refer to the documentation that came with it.

Installing the TriMedia CodeWarrior Plugins

Win95/98/NT

To install the TriMedia CodeWarrior Plugins, double-click the Setup program on your CodeWarrior Installation CD and follow instructions. If CodeWarrior is installed on your system, choose to install the TriMedia Plugins only. Otherwise, perform a complete installation.

MacOS

To install the TriMedia CodeWarrior Plugins, double-click the installation program on your CodeWarrior Installation CD and follow instructions. If CodeWarrior is installed on your system, choose to install the TriMedia Plugins only. Otherwise, perform a complete installation.

Known Problems

The following are known problems with the TriMedia plugins. They apply to all platforms unless otherwise noted.

- “Disassemble” IDE menu command does nothing.

 ©1999 Philips Semiconductors 10/08/99

- "Preprocess" IDE menu command only applies to C source files. Attempt to preprocess *.s, *.t or object files is ignored.
- You cannot strip static archive libraries.
- Win95/98/NT plugins do not do context-sensitive help.
- Win95/98/NT plugins do not support scripting of preference panels.
- You cannot interrupt the scheduler or assembler phase during compilation.
- The CodeWarrior IDE does not automatically launch the TriMedia simulator, debugger, or **tm(g)mon** applications. You have to launch these as a separate step after creating your TriMedia executable.
- Output of TriMedia profiling is incompatible with Metrowerks Profiler. You must use the TriMedia profiler to generate a report.
- On Win95/98 the CodeWarrior window colors get messed up when other applications are running. Problem is known to Metrowerks. Does not happen on WinNT or MacOS.

Implementation Notes

Speeding Up Compilation

To speed up compilation, do the following:

1. Uncheck the "Keep intermediate files" and "Show code and data sizes" in the TriMedia Target panel. (See TriMedia Target on page 171.)
2. Disable "Activate Browser" in the Build Extras preference panel.

These settings do not affect code generation. They only provide additional information as a result of compiling your code.

Browser Catalog

If you activate browser recording with "Activate Browser" in the Build Extras panel, the TriMedia compiler may add macro symbols to the catalog that do not appear in any source file. Internally defined symbols for which no source file exists will have the value of 1. In addition, you won't be able to *find* the definition of these symbols (such as the __TCS__ macro) with the browser.

Determining which symbols are internally defined depends on your compiler settings.

File Names and Search Paths

Files names which are specified in the TriMedia preference panels are searched by the rules specified in the Access Paths preference panel. For example, the prefix file name in the C Language panel, the graft tuning file in the TriMedia Compiler panel, and the file containing export symbols in the TriMedia Linker panel are all searched in the directories specified in the Access Paths panel.

All file names specified in the TriMedia preference panels and used by the TriMedia compiler and linker can have embedded whitespace in them.

Specifying TriMedia CodeWarrior Settings

The <Target Name> Settings command in the Edit menu of the CodeWarrior IDE allows you to specify the settings that affect a particular target in a project. These settings are grouped into a hierarchical list of panels.

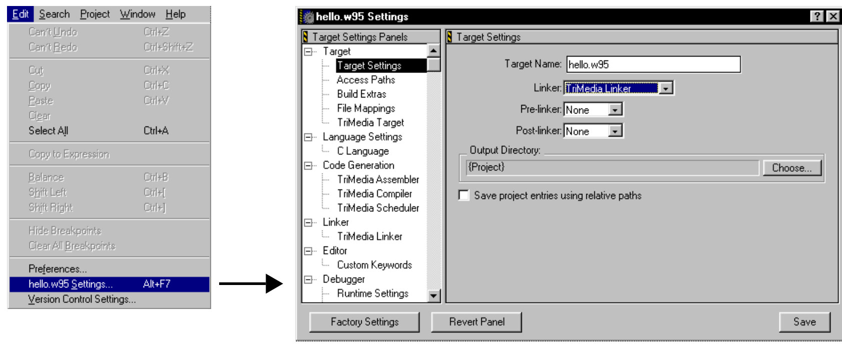


Figure 31 The Target Settings dialog box

Some of the settings are generic and some are TriMedia-specific. This section describes the panels that contain TriMedia-specific settings only. For more information about the other settings, refer to the CodeWarrior IDE documentation.

 © 1999 Philips Semiconductors 10/08/99

Target Settings

The Target Settings panel allows you to select the target compiler and the linker to use.

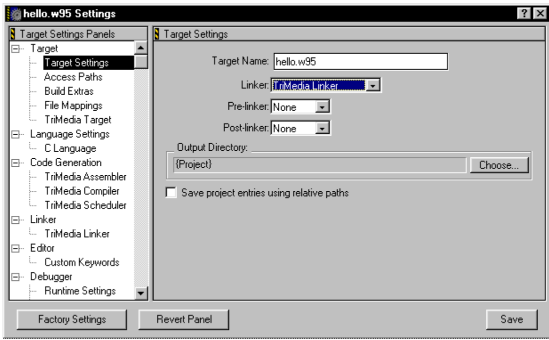


Table 7 describes the TriMedia-specific settings in the Target Settings panel.

Table 7 Target Settings Panel

Setting	Description
Target Name	Use this field to type the name of the target file.
Linker	Use this pull-down menu to select TriMedia Linker to use the TriMedia compilation system (compiler and linker) for the current target.

Access Paths

The Access Paths panel allows you to specify additional access paths for the CodeWarrior IDE to search while compiling and linking projects.

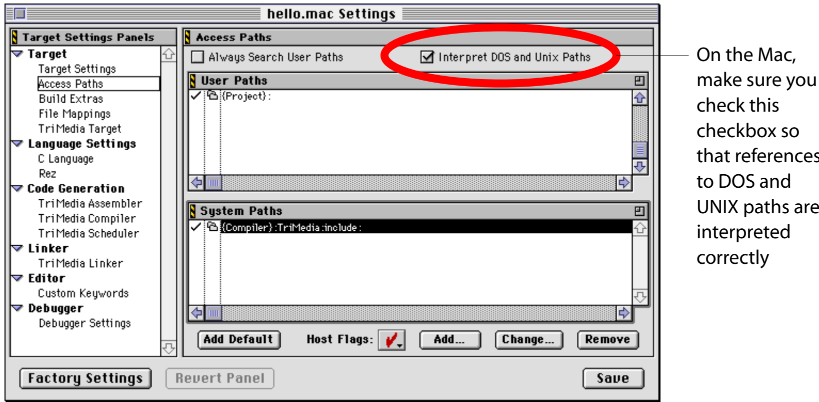


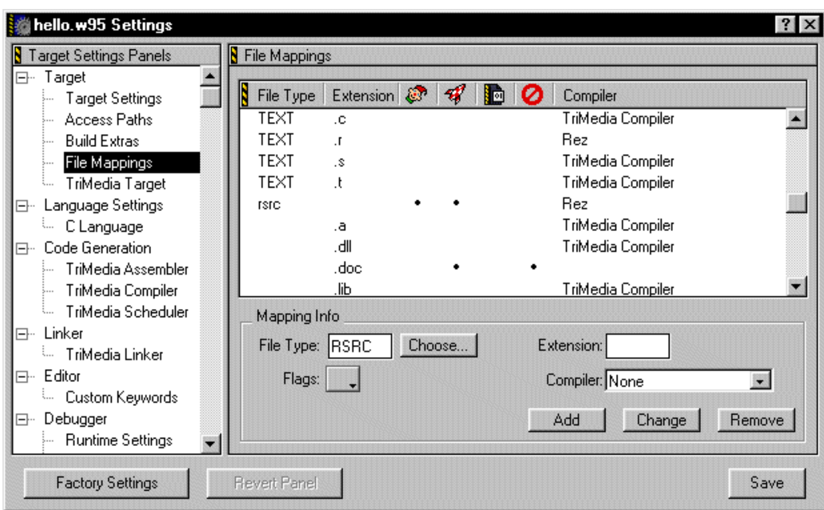
Table 8 describes the TriMedia-specific setting in the File Mappings panel.

Table 8 Access Paths Panel

Setting	Description
Always Search User Paths	Check to instruct CodeWarrior to search user paths in addition to System Paths when looking for files included with **include** <...> statements. Note: CodeWarrior automatically searches user paths and system paths for files included with **include "..."** statements.
User Paths	Win95: Click the User Paths radio button to display the User Paths list. The User Paths list provides the access paths of the files that are specific to your project.
System Paths	Win95: Click the System Paths radio button to display the System Paths list. The System Paths list provides the access paths of system headers, PowerPlant, MSL, and so on.
Host Flags	Use this menu to specify the host platform for the currently selected access path in the User Paths or System Paths list.

File Mappings

The File Mappings panel allows you to associate file name extensions with a plug-in compiler. If you select TriMedia Linker from the Linker pull-down menu in the Target Settings panel, the File Mappings panel displays TriMedia-specific file name extensions.



 ©1999 Philips Semiconductors 10/08/99

Table 9 describes the TriMedia-specific setting in the File Mappings panel.

Table 9 File Mappings Panel

Setting	Description
Compiler	Use this pull-down menu to select TriMedia Compiler for the currently selected file type in the File Mappings list.

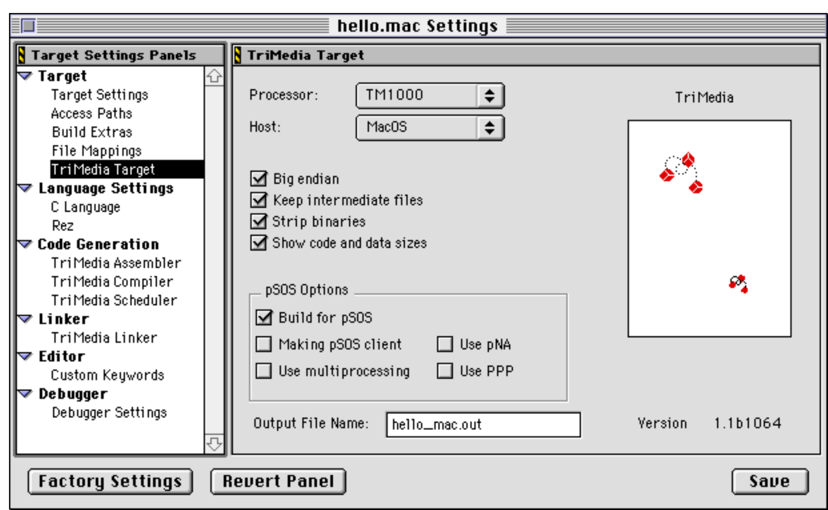
By default, the TriMedia compiler accepts the following extensions:

Extension	Description
.c	C source files. Can be preprocessed or compiled.
.t	TriMedia decision tree files. Can be scheduled and assembled.
.s	TriMedia assembly language files. Can be assembled.
.dll	TriMedia object files that can be read by the TriMedia compiler to be used for linking.
.a	
.lib	
.o	

For more information on how to add, delete, and modify file mappings, refer to the CodeWarrior IDE documentation.

TriMedia Target

The TriMedia Target panel allows you to specify the settings that are needed to generate TriMedia targets.



All the settings in this panel (described in Table 10) have equivalent commands and command line options.

Table 10 TriMedia Target Panel

Setting	**Description**	**Command Line Equivalent**
Processor	Use this pop-up menu to specify the target TriMedia processor. TM1000 TM1000 TM1100 TM2000 Currently, two options are available: TM1000 and TM1100.	tmcc -target
Host	Use this pop-up menu to specify the host OS or runtime environment. Win95 None MacOS Win95 WinNT Simulator Use the None selection for embedded systems.	tmcc -host
Big endian	Check this box to compile for big endian TriMedia execution. Uncheck this box to compile for little endian TriMedia execution.	tmcc -eb tmcc -el
Keep intermediate files	Check this box to keep intermediate results created by the pre-processor, compiler, and scheduler in the same folder as the source file. Intermediate results are stored in the *.i, *.t, *.s files. Uncheck this box to discard intermediate results created by the pre-processor, compiler, and scheduler. This saves time and disk space.	tmcc -K
Strip binaries	Check this box to strip symbol information from TriMedia object files produced by the currently active project. Uncheck this box to keep symbol information in TriMedia object files.	tmstrip outfilename
Show code and data sizes	Check this box to disable the display of code and data sizes in the project window. Uncheck this box to display the code and data sizes in the project window.	tmsize
Build for pSOS	Check this box to build for pSOS applications and enable the remaining pSOS Options checkboxes. Uncheck this box if not building for pSOS applications. The remaining pSOS Options checkboxes will be disabled.	NA

 © 1999 Philips Semiconductors 10/08/99

Table 10 TriMedia Target Panel

Setting	Description	Command Line Equivalent
Making pSOS client	Check this box to make a client pSOS application. The linker automatically links in the appropriate library.	NA
Use multiprocessing	Check this box to build multiprocessing pSOS applications. See Table 10-1.	NA
Use pNA	Check this box to use the pSOS pNA component. See Table 10-1.	NA
Use PPP	Check this box to use the pSOS PPP component. The linker automatically links in the appropriate library.	NA
Output File Name	Use this field to type the name of the output file for the project and target.	tmcc -o outfilename

When the "Build for pSOS" flag is enabled, the compiler defines the macro symbols SC_PSOS, SC_PSOSM, and SC_PNA with values as described in Table 11.

Table 11 Possible combinations of #**define** statements when building for pSOS

	☑ Use pNA	☐ Use pNA
☑ Use multiprocessing	SC_PSOS = NO	SC_PSOS = NO
	SC_PSOSM = YES	SC_PSOSM = YES
	SC_PNA = YES	SC_PNA = NO
☐ Use multiprocessing	SC_PSOS = YES	SC_PSOS = YES
	SC_PSOSM = NO	SC_PSOSM = NO
	SC_PNA = YES	SC_PNA = NO

C Language

The C Language panel allows you to specify the C language settings for preprocessing and compiling your TriMedia application. The checkboxes in this panel have equivalent **tmccom** and **cpp** command-line options.

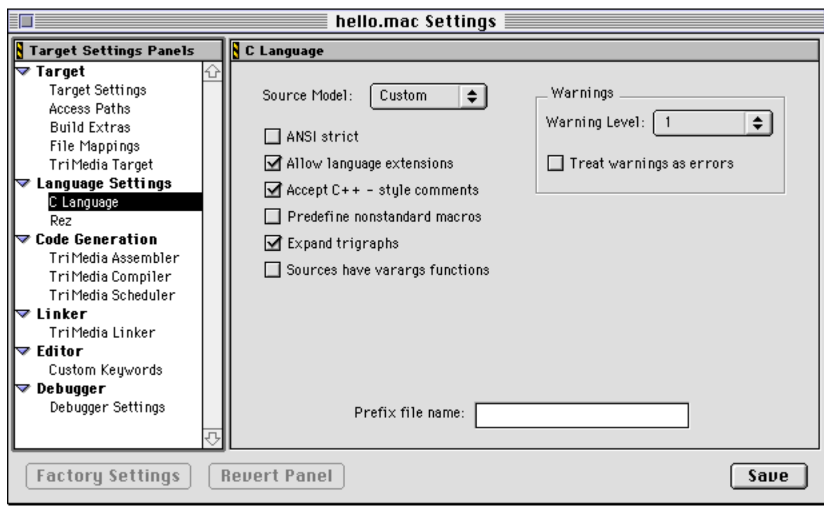


Table 12 describes the TriMedia-specific settings in the C Language panel.

Table 12 File Mappings Panel

Setting	Description	Command Line Equivalent
Source Model	Use this pop-up menu to choose a pre-selected set of C/C++ language options.	NA
ANSI strict	Check to generate error messages for strict ANSI C rule violations. Note: Using the ANSI source model (same as having the ANSI strict and Expand trigraphs checkboxes checked) is *not* recommended for TriMedia applications because TriMedia custom ops are not ANSI.	cpp -pedantic -$
Allow language extensions	Check to allow C language extensions that are not part of the ANSI C standard.	tmccom -standard
Accept C++ style comments	Check to allow use of "//" as the beginning of a C comment.	cpp -lang-c
Predefine non-standard macros	Check to predefine nonstandard macros.	cpp -undef
Expand trigraphs	Check to enable compiler recognition of trigraph characters.	cpp -trigraphs

Table 12 File Mappings Panel

Setting	Description	Command Line Equivalent
Sources have varargs functions	Check if your source files contain varargs functions.	tmcc -varargs
Warning Level	Use this pop-up menu to choose the compiler warning level.	tmccom -W n
Treat warnings as errors	Check to instruct the compiler to treat all warnings as error messages. The compiler will not compile a file until all warnings are resolved.	NA
Prefix file name	Use this field to enter the name of the header file that the compiler includes before every source file in the project.	cpp -include

TriMedia Assembler

The TriMedia Assembler panel allows you to specify the TriMedia Assembler (**tmas**) settings. The checkboxes in this panel have equivalent **tmas** command-line options.

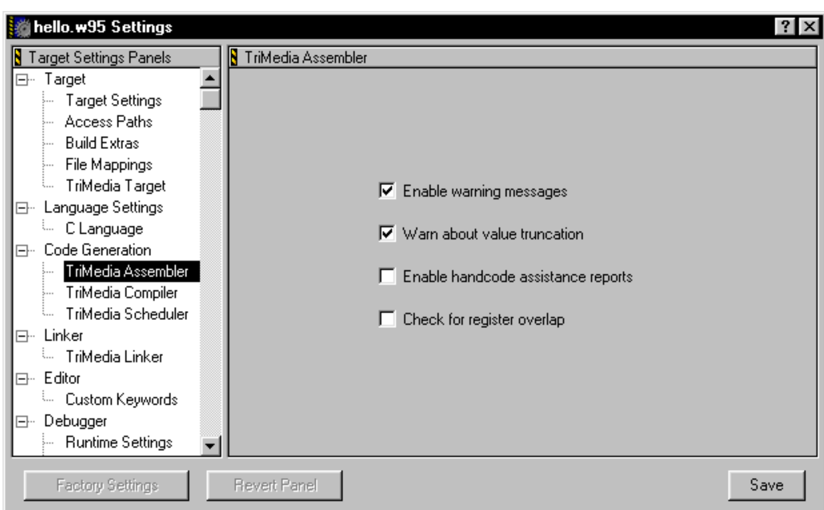


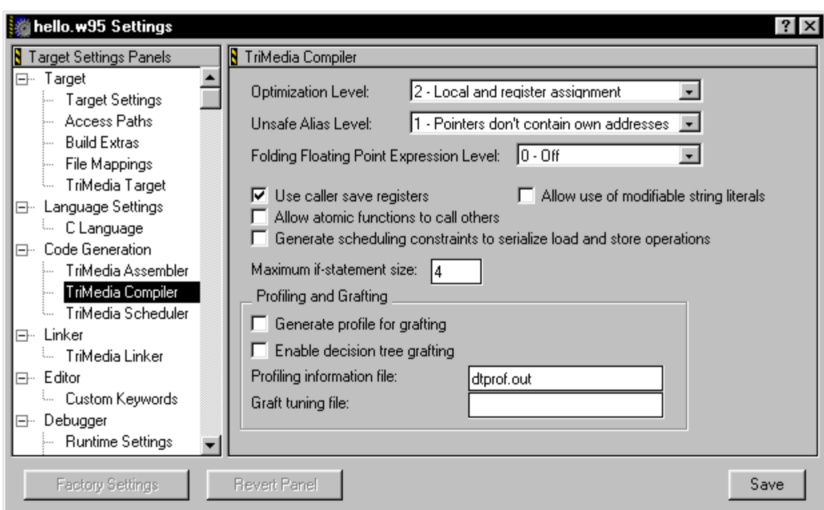
Table 13 describes the TriMedia Assembler panel settings.

Table 13 TriMedia Assembler Panel

Setting	Description	Command Line Equivalent
Enable warning messages	Check to enable Assembler warning messages. Uncheck to disable Assembler warning messages.	tmas -w
Warn about value truncation	Check to enable value truncation warnings when values are too large to fit into an object code field. Uncheck to disable value truncation warnings.	tmas -ignore-truncate
Enable hand-code assistance reports	Check to let the Assembler generate warnings and reports that are meaningful for special-purpose handcoded assembly programs. Uncheck to disable the generation of handcode assistance reports.	tmas -handcode
Check for register overlap	Check to enable register overlap checking that may be used in the future for binary compatibility. Uncheck to disable register overlap checking.	tmas -register-overlap

TriMedia Compiler

The TriMedia Compiler panel allows you specify **tmccom** options. The checkboxes, pop-up menus, and fields in this panel have equivalent **tmccom** command-line options.



 ©1999 Philips Semiconductors 10/08/99

Table 14 describes the TriMedia Compiler panel settings.

Table 14 TriMedia Compiler Panel

Setting	**Description**	**Command Line Equivalent**
Optimization Level	Use this pop-up menu to choose a pre-selected optimization level. `tmccom -conservative_03` 2 - Local and register assignment 0 - Off 1 - Local 2 - Local and register assignment 3 - Conservative Global 3 - Global The optimization levels in this pop-up menu correspond to the -O*n* command-line option where ***n*** represents the optimization level (from 0 to 3).	tmccom -On
Unsafe Alias Level	Use this pop-up menu to choose a pre-selected set of unsafe alias levels. 1 - Pointers don't contain own addresses 0 - Safe aliasing 1 - Pointers don't contain own addresses 2 - No addresses of globals	tmccom -An
Folding Floating Point Expression Level	Use this pop-up menu to choose a pre-selected set of folding levels of floating-point expressions. 0 - Off 0 - Off 1 - "if" conversions; ignores F 2 - Tree height reduction	tmccom -dirty_float n
Use caller save registers	Check to enable the use of caller save registers for leaf functions.	tmccom -no_caller_save
Allow use of modifiable string literals	Check to allow the program to use string literals to initialize writable data.	
Allow atomic functions to call others	Check to allow atomic functions to call other functions.	tmccom -allow_atomic_calls
Generate scheduling constraints to serialize load and store operations	Check to generate scheduling constraints that enforce the original source ordering of all load and store operations.	tmccom -serial
Maximum if-statement size	Use this field to specify the maximum number of operations in small **if** statements that the program transforms into mux operations. The default value is 4. Entering 0 in this field disables transformation.	tmccom -max_if_size n

Table 14 TriMedia Compiler Panel

Setting	Description	Command Line Equivalent
Generate profile for grafting	Check to instrument each compiled function with branch-level profile counters. Note: You can't check this checkbox if you have already checked the "Enable decision tree grafting" checkbox.	tmccom -genprofile
Enable decision tree grafting	Check to enable decision tree grafting. Note: You can't check this checkbox if you have already checked the "Generate profile for grafting" checkbox.	tmccom -graft
Profiling information file	Use this field to enter the name of the file containing profiling information to optimize functions being compiled.	tmccom -readprofile file
Graft tuning file	Use this field to enter the name of the file containing decision tree grafting tuning parameters.	tmccom -graft_tuning_file file

TriMedia Scheduler

The TriMedia Scheduler panel allows you to specify **tmsched** options. The checkboxes and fields in this panel have equivalent **tmsched** command-line options.

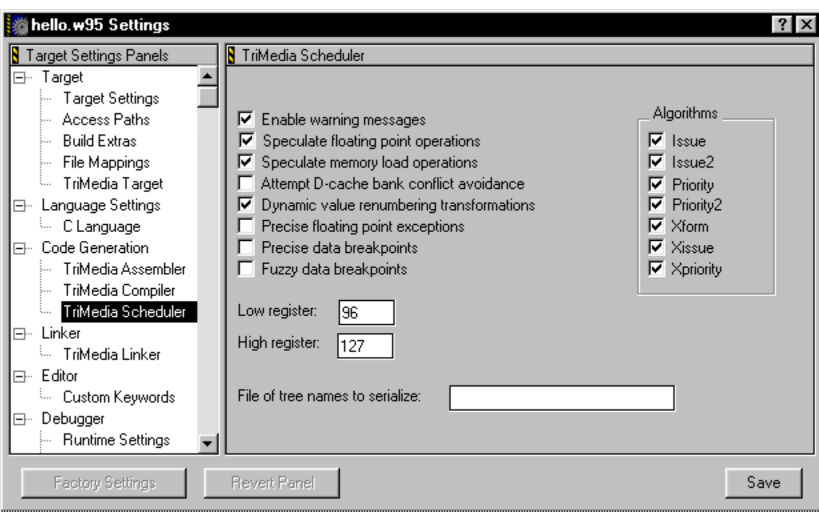


Table 15 describes the TriMedia Scheduler panel settings.

Table 15 TriMedia Scheduler panel description

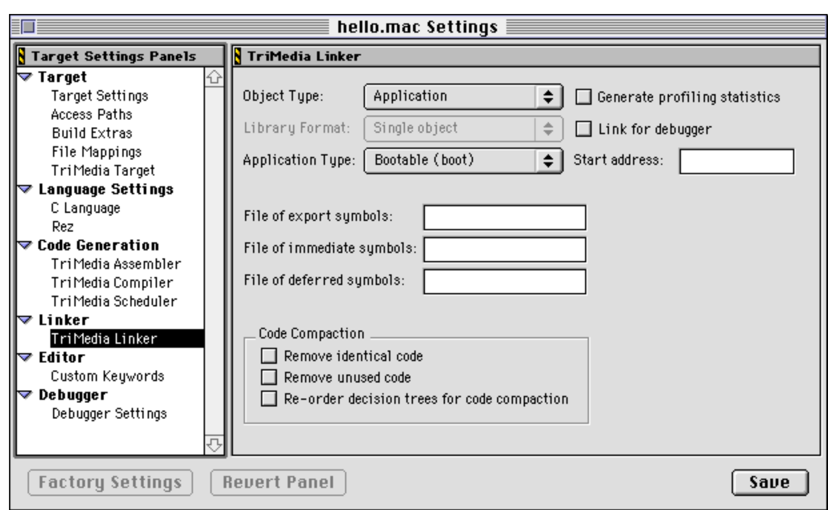
Setting	Description	Command Line Equivalent
Enable warning messages	Check to enable tmsched warning messages.	tmsched -w
Speculate floating point operations	Uncheck to disallow speculation of floating point operations	tmsched -nofloatspec
Speculate memory load operations	Uncheck to disallow speculation of memory load operations.	tmsched -noloadspec
Attempt data cache bank conflict avoidance	Check to enable tmsched to attempt its built-in ad hoc data cache bank conflict avoidance, in absence of information from the compiler or the user.	tmsched -bc
Dynamic value renumbering transformations	Uncheck to disable all dynamic value renumbering transformations.	tmsched -norenumber
Precise floating point exceptions	Check to enable support for precise floating point exceptions. Precise floating point exceptions are caught in the same decision tree.	tmsched -precise_fp
Precise data breakpoints	Check to enable precise data breakpoints. This introduces additional scheduling constraints that may affect performance. Note: Checking this checkbox unchecks the Fuzzy data breakpoints checkbox. Data breakpoints cannot be fuzzy and precise at the same time.	tmsched -precise_bp
Fuzzy data breakpoints	Check to enable fuzzy data breakpoints. This introduces additional scheduling constraints that may affect performance. Note: Checking this checkbox unchecks the Precise data breakpoints checkbox. Data breakpoints cannot be fuzzy and precise at the same time.	tmsched -fuzzy_bp
Low register	Use this field to specify the lowest register available for instruction scheduling. Note: The compiler assigns the Low Register value automatically under normal compilation conditions.	tmsched -reglow=n

Table 15 TriMedia Scheduler panel description (Continued)

Setting	Description	Command Line Equivalent
High register	Use this field to specify the highest register available for instruction scheduling. Note: The compiler assigns the High Register value automatically under normal compilation conditions.	tmsched -reghigh=n
File of tree names to serialize	Use this field to specify the name of the file containing the comma-separated list of tree names. The scheduler will serialize memory operations for these trees. This is useful in debugging aliasing problems.	tmsched -serial=treename [, treename]...
Issue	Use these checkboxes to specify the scheduling algorithms to run on the input file.	tmsched -algorithm=issue
Issue 2		tmsched -algorithm=issue2
Priority		tmsched -algorithm=priority
Priority 2		tmsched -algorithm=priority2
Xform		tmsched -algorithm=xform
Xissue		tmsched -algorithm=xissue
Xpriority		tmsched -algorithm=xpriority

TriMedia Linker

The TriMedia Linker panel allows you to specify the TriMedia Linker (**tmld**) options. The checkboxes and fields in this panel have equivalent **tmld** command-line options.



 ©1999 Philips Semiconductors 10/08/99

Table 16 describes the TriMedia Linker panel settings.

Table 16 TriMedia Linker Panel

Setting	Description	Command Line Equivalent
Object Type	Use this pop-up menu to choose the object type to create with the linker. tmld -btype [boot | dynboot | app Application Application Library Shared Library (DLL) tmld -o *filename*	tmld -btype [boot | dynboot |app | dll] tmld -o filename
Library Format	Use this pop-up menu to choose the library format to use for creating a single object or an archive of objects. This applies to static libraries only. Library Single object Single object Archive of objects tmld -o *filename* tmar -r *filename*	tmld -o filename tmar -r filename
Application Type	Use this pop-up menu to choose the type of executable to which to link. Application Single object Bootable (boot) Bootable (boot) Dynamic bootable (dynboot) Non-bootable (app)	tmld -btype [boot | dynboot | app | dll]
Generate profiling statistics	Check to enable the generation of profiling statistics during execution. Note: Checking this checkbox unchecks the Link for debugger checkbox. You cannot link for debugging and profiling at the same time.	tmcc -ptm
Link for debugger	Check to generate a TriMedia executable that can be used with the TriMedia Debugger (tmdbg). Note: Checking this checkbox unchecks the Generate profiling statistics checkbox. You cannot link for debugging and profiling at the same time. Note: The Enable Debugger option in the Project menu of the CodeWarrior IDE is not applicable to TriMedia applications.	tmld -g
Start address	Use this field to specify the global symbol as the start address. This value is required for bootable, dynamically linked, and non-bootable executables.	tmld -start=start_symbol

Table 16 TriMedia Linker Panel

Setting	Description	Command Line Equivalent
File of export symbols	Use this field to specify the name of the file containing comma-separated symbols to be exported to the dynamic loader.	tmld -bexport symbol [, symbol]...
File of immediate symbols	Use this field to specify the name of the file containing comma-separated symbols (dynamically loadable module for resolving modules dynamically). These modules are loaded as soon as the code segment, which references it, is loaded.	tmld -bimmediate code_seg [, code_seg]...
File of deferred symbols	Use this field to specify the name of the file containing comma-separated files. These modules are loaded at runtime upon the first call to a function exported by the code segment(s).	tmld -bdeferred
Remove identical code	Check to remove duplicate identical code which consists of identical function epilogues created by the compiler.	tmld -bfoldcode
Remove unused code	Check to remove unused code. The linker analyzes all read-only sections to determine parts which cannot be used.	tmld -bremoveunusedcode
Reorder decision trees for code compaction	Check to reorder code at the decision tree level to minimize the amount of instruction padding. This may affect performance.	tmld -bcompact

 © 1999 Philips Semiconductors 10/08/99