Book 2—Cookbook

Part C:

Bootstrapping TriMedia

TriMedia

Version 2.1

Table of Contents

Chapter 7

Bootstrapping TriMedia in Autonomous Mode

Introduction

Bringing up TriMedia in stand-alone mode involves a number of steps. This chapter outlines the essential steps common to different stand-alone configurations. It also includes sample programs that you can modify to suit your needs.

In order to fully understand this chapter, you must be familiar with the TriMedia architecture and you will need to have read Chapter 12 of the appropriate TriMedia data book, which is the official document on both stand-alone and host-assisted boot procedures.

Overview of Stand-Alone Boot

During power-on reset, TriMedia boot block reads some configuration information from the EEPROM through I2C. The contents of the EEPROM determine, among other things, whether TriMedia continues to boot from the EEPROM or expects another processor (such as a PC or a Mac) to complete the TriMedia boot sequence. In a host-assisted boot, the EEPROM contains just 10 bytes that set a few parameters such as **TRI_CLKIN**, **PCI Subsystem Id**, **Vendor Id**, **MM_CONFIGs**, and **PLL_RATIOs**. The task of downloading an application to SDRAM and taking TriMedia out of reset is left to a host-based program (such as **tmmon** on the PC or Mac).

In a stand-alone boot, the EEPROM contains, in addition, the initial boot program whose size is restricted to 2K bytes. This initial boot program, called *L1 boot program*, is transferred by the TMs1000 boot block from EEPROM to SDRAM and then executed. It is the responsibility of the L1 boot program to load other programs (we will call them L2 programs) from any attached device, such as on-board UVEPROMs (or flash) or networks and to execute them.

PCI Signals

If TriMedia is to be used in a stand-alone system, certain PCI signals need to be properly accounted for (Request, Grant, and IDSEL, for example). The request and grant signals are always needed even if the application does not use PCI. Similarly IDSEL must be connected to an address line for the function **procGetCapabilities** to work properly.

Creating an EEPROM image

The L1 boot EEPROM consists of a 47-byte header followed by the L1 boot program.

EEPROM Header

Contents of the EEPROM header are documented in Chapter 12 of the data book. The memory system parameters are documented in Chapter 11.

This chapter includes a sample program *l1rom.c*, which creates an EEPROM image file (binary) of the L1 boot program. The l1rom.c program adds a 47-byte header to the given L1 boot program and swaps the bytes of the L1 boot program when creating the EEPROM image. l1rom.c uses fixed values for **TRI_CLKIN**, **PLL** clock ratios, and so on. Stand-alone system developers need to examine and change the first 8 bytes of the EEPROM header in l1rom.c, if necessary, to suit their system.

L1 Boot Program

L1 boot code needs to do some initialization of TriMedia, such as setting the PCSW, BIU_CTRL, setting up stack and frame pointers, initializing PCI devices (if any) and copying the L2 code to SDRAM. It then jumps to the beginning of L2 code.

The sample L1 program consists of two files:

- l1start.trees

  This file defines a function **__start** which initializes PCSW and BIU_CTRL; sets up SP (stack pointer), FP (frame pointer), and RP (return pointer); and calls **L1main**. On return from **L1main**, it jumps to the L2 load address returned by **L1main**.

- l1main.c

  The function **L1main** simply copies L2 code from a PCI-slave UVEPROM to SDRAM. After copying L2 code to SDRAM, the data cache is flushed and then invalidated. After that, the instruction cache is cleared. L1main() returns the L2 load address to the caller, **__start.**

  Note
  If you are using TM-1000 chips earlier than revision 1s1.1, I2C might be in some stuck state after autoboot. The l1main.c file contains a simple workaround.

On the TM-1000 debug board, the UVEPROM is located at (PCI) address **0xFFC00000**. The sample L1 boot code loads the sample L2 code from (PCI) address **0xFFC0000** to (SDRAM) address **0x840** (the first cache aligned address after 2 K, because L1 code can be at most, 2K bytes).

Steps in creating an EEPROM image.

1. Compile l1start.trees and l1main.c as follows.

```
cp l1start.trees l1start.t
tmcc  -x -v -c -eb -DL2_LOAD_ADDR=0x840 \
      -DL2_CODE_SIZE=200000 \
      -DL2_ROM_DEV_ADDR=0xFFC00000 \
      l1start.t l1main.c
```

   The L1 boot program needs to know the size of L2 code. The **tmcc** option **-DL2_CODE_SIZE=150000** defines **L2_CODE_SIZE**. The sample L2 code fits within 200000 bytes. L1 boot code sets up SP and FP starting at **MEMORY_SIZE** (defined to be 8 MB, because IREF boards have 8 MB memory). For stand-alone systems, **MMIO_BASE**

is defined to be **0xEFE0000**. This value must agree with that used in l1rom.c as part of the 47-byte EEPROM header.

2. Link l1start.o and l1main.o and verify the executable size.

```
tmld -eb -o  l1.out l1start.o l1main.o
tmsize l1.out
```

You cannot use the **tmcc** compiler driver to link the L1 boot code, because **tmcc** adds a number of options and libraries by default to the linker command line. This step just verifies that the sum of text, data, data1, and bss section sizes is less than 2K bytes.

IMPORTANT
It is important that l1start.o appears first in the link command before all other files that are linked. ▲

3. Relocate the executable and produce a memory image.

The executable l1.out produced in Step 2 has text, data, data1, and bss sections. In addition, it contains information about the executable itself. To generate a memory image, you must specify the load start address and the memory size and pass the **-mi** option to **tmld**. This concatenates the text, data, data1, and bss sections and produces a memory image. You must also define **__clock_freq_init, __MMIO_base_init,** and **__begin_stack_init** as download parameters (**-bdownload __clock_freq_init** etc.) and then define their values (**-tm_freq 100000000** defines the TriMedia clock frequency as 100 MHz). If you use a TM1 IREF board with an 80 MHz TM1.1 chip, change this option to **-tm_freq 80000000**. Ensure that l1start.o is the first file in the list of files linked. This is because TriMedia starts execution at SDRAM BASE) and you want the startup code **__start** to be located at that address.

```
tmld  -eb -o "l1.mi" -bdownload __clock_freq_init -mi \
      -bdownload __MMIO_base_init \
      -bdownload __begin_stack_init \
      -exec -start=__L1start  -tm_freq 100000000 \
      -mmio_base 0xEFE00000  \
      -load=0,0x800000 l1start.o l1main.o
```

In the above example, memory starts at **0** and the size is 8 MB.

Note
__clock_freq_init is required because the TriMedia device libraries rely on this definition of the clock frequency to determine things like the number of ticks in a microsecond or the proper control value to set the video clock to 27 MHz.

4. Add a 47-byte header to the memory image, swap the bytes in the L1 boot program, and produce the L1 EEPROM image. Swapping bytes of the L1 boot program is always needed because of the way the boot block transfers bytes from EEPROM to SDRAM. The l1rom.c sample program has hard-coded values for the 47-byte header. You might want to modify l1rom.c and change the first 8 bytes to suit your system. The command **l1rom l1.mi** produces the l1.eeprom EEPROM image file, which is a binary file

that you can use to program an EEPROM part such as ATML646 24c16, using an EEPROM programmer such as BP 1200.

Sample Programs

This chapter includes the following sample programs:

Sample Programs	Description
makefile.unix	Makefile for SunOS and HP-UX. It is used to create L1 boot code, L2 code, EEPROM image, etc. The TCS and CC macros need to be customized for the particular compilation host platform.
makefile.win	Makefile for MKS Make on Windows 95/NT. It is used to create L1 boot code, L2 code, EEPROM image, etc. The TCS and CC macros need to be customized for the particular compilation host platform.
l1start.trees, l1main.c	These 2 files form the L1 boot code.
l1rom.c	This program is built as a host shell command. It is used to create the L1 EEPROM image.
vivot.c	This file forms the L2 code (plus standard device libraries).
seeval.c	If you are using the SEEVAL EEPROM programmer, you need this. If not, ignore this file.

makefile.unix

```
# ------------------------------------------------------------------------
#  L2 program must be compiled to have a load address of
#  L2_LOAD_ADDR, since L2_LOAD_ADDR is used in l1main.c
# ------------------------------------------------------------------------
CP  = /bin/cp
MV  = /bin/mv
RM  = /bin/rm
CC  = /t/lang/acc

TCS     = /t/qasoft/build/tcs1.1z/1054/SunOS
TMCC    = $(TCS)/bin/tmcc
TMLD    = $(TCS)/bin/tmld
TMSIZE  = $(TCS)/bin/tmsize

L1ROM        = l1rom
MMIO_BASE    = 0xefe00000
SDRAM_BASE   = 0x0
SDRAM_LIMIT  = 0x800000
TM_FREQ      = 100000000

# ------------------------------------------------------------------------
# L1 boot program can be 2048 bytes long atmost.
# L2_LOAD_ADDR is the next cache aligned address, i.e 2112
# ------------------------------------------------------------------------
L2_LOAD_ADDR    = 2112
L2_CODE_SIZE    = 150000
L2_ROM_DEV_ADDR = 0xffc00000

ENDIAN    = -el
L1_CFLAGS = -v $(ENDIAN) -host nohost      \
            -DL2_LOAD_ADDR=$(L2_LOAD_ADDR) \
            -DL2_CODE_SIZE=$(L2_CODE_SIZE) \
            -DL2_ROM_DEV_ADDR=$(L2_ROM_DEV_ADDR)

L1_LDFLAGS = $(ENDIAN) -btype boot         \
             -bdownload __clock_freq_init  \
             -bdownload __MMIO_base_init   \
             -bdownload __begin_stack_init \
             -exec -start=__start

L1_MIFLAGS = $(ENDIAN)                     \
             -bdownload __clock_freq_init  \
             -bdownload __MMIO_base_init   \
             -bdownload __begin_stack_init \
             -mi -exec -start=__start      \
             -tm_freq $(TM_FREQ)           \
             -mmio_base $(MMIO_BASE)       \
             -load=$(SDRAM_BASE),$(SDRAM_LIMIT)

L2_CFLAGS  = -v $(ENDIAN) -I$(TCS)/include/Win95 \
             -host nohost \
             -DMMIO_BASE_ADDR=$(MMIO_BASE)

L2_MIFLAGS = $(ENDIAN)                     \
             -bdownload __clock_freq_init  \
             -mi -exec -start=__start      \
             -tm_freq $(TM_FREQ)           \
             -mmio_base $(MMIO_BASE)       \
             -load=$(L2_LOAD_ADDR),$(SDRAM_LIMIT)
```



```
# ------------------------------------------------------------------------

l1.out: l1start.trees l1main.c
        @echo ""
        @echo making $@
        $(RM) -f l1start.t
        $(CP) l1start.trees l1start.t
        $(TMCC) -x $(L1_CFLAGS) -c l1start.t l1main.c
        $(TMLD) $(L1_LDFLAGS) -o $@ l1start.o l1main.o
        $(TMSIZE) $@

l1.mi: l1start.trees l1main.c
        @echo ""
        @echo making $@
        $(RM) -f l1start.t
        $(CP) l1start.trees l1start.t
        $(TMCC) -x $(L1_CFLAGS) -c l1start.t l1main.c
        $(TMLD) -o $@ $(L1_MIFLAGS) l1start.o l1main.o

l1.eeprom: l1.mi $(L1ROM)
        @echo ""
        @echo "Adding 47 bytes autoboot protocol header and swapping bytes"
        $(L1ROM) l1.mi

$(L1ROM): l1rom.c
        @echo ""
        @echo making $@
        $(CC) -o $@ -DSDRAM_BASE=$(SDRAM_BASE) -
DSDRAM_LIMIT=$(SDRAM_LIMIT) l1rom.c

# ------------------------------------------------------------------------

vivot.out: vivot.c
        $(TMCC) $(L2_CFLAGS) -o $@ vivot.c

vivot.mi: vivot.c
        $(TMCC) $(L2_CFLAGS) -o $@ -tmld $(L2_MIFLAGS) -- vivot.c

# ------------------------------------------------------------------------

clean:
        $(RM) -f $(L1ROM) *.o *.t *.i *.s *.eeprom *.out *.mi *.dump
```

makefile.win

```
# ------------------------------------------------------------------------
#  L2 program must be compiled to have a load address of
#  L2_LOAD_ADDR, since L2_LOAD_ADDR is used in l1main.c
# ------------------------------------------------------------------------
CP  = cp
MV  = mv
RM  = rm
CC  = cc

TCS     = C:/TriMedia
TMCC    = $(TCS)/bin/tmcc
TMLD    = $(TCS)/bin/tmld
TMSIZE  = $(TCS)/bin/tmsize
L1ROM   = l1rom.exe

MMIO_BASE    = 0xefe00000
SDRAM_BASE   = 0x0
SDRAM_LIMIT  = 0x800000
TM_FREQ      = 100000000

# ------------------------------------------------------------------------
# L1 boot program can be 2048 bytes long atmost.
# L2_LOAD_ADDR is the next cache aligned address, i.e 2112
# ------------------------------------------------------------------------
L2_LOAD_ADDR    = 2112
L2_CODE_SIZE    = 150000
L2_ROM_DEV_ADDR = 0xffc00000

ENDIAN     = -el
L1_CFLAGS  = -v $(ENDIAN) -host nohost            \
             -DL2_LOAD_ADDR=$(L2_LOAD_ADDR)       \
             -DL2_CODE_SIZE=$(L2_CODE_SIZE)       \
             -DL2_ROM_DEV_ADDR=$(L2_ROM_DEV_ADDR)

L1_LDFLAGS = $(ENDIAN) -btype boot         \
             -bdownload __clock_freq_init  \
             -bdownload __MMIO_base_init   \
             -bdownload __begin_stack_init \
             -exec -start=__start

L1_MIFLAGS = $(ENDIAN)                     \
             -bdownload __clock_freq_init  \
             -bdownload __MMIO_base_init   \
             -bdownload __begin_stack_init \
             -mi -exec -start=__start      \
             -tm_freq $(TM_FREQ)           \
             -mmio_base $(MMIO_BASE)       \
             -load=$(SDRAM_BASE),$(SDRAM_LIMIT)

L2_CFLAGS  = -v $(ENDIAN) -I$(TCS)/include/Win95 \
             -host nohost \
             -DMMIO_BASE_ADDR=$(MMIO_BASE)

L2_MIFLAGS = $(ENDIAN)                     \
             -bdownload __clock_freq_init  \
             -mi -exec -start=__start      \
             -tm_freq $(TM_FREQ)           \
             -mmio_base $(MMIO_BASE)       \
             -load=$(L2_LOAD_ADDR),$(SDRAM_LIMIT)
```



```
# ------------------------------------------------------------------------

l1.out: l1start.trees l1main.c
        @echo ""
        @echo making $@
        $(RM) -f l1start.t
        $(CP) l1start.trees l1start.t
        $(TMCC) -x $(L1_CFLAGS) -c l1start.t l1main.c
        $(TMLD) $(L1_LDFLAGS) -o $@ l1start.o l1main.o
        $(TMSIZE) $@

l1.mi: l1start.trees l1main.c
        @echo ""
        @echo making $@
        $(RM) -f l1start.t
        $(CP) l1start.trees l1start.t
        $(TMCC) -x $(L1_CFLAGS) -c l1start.t l1main.c
        $(TMLD) -o $@ $(L1_MIFLAGS) l1start.o l1main.o

l1.eeprom: l1.mi $(L1ROM)
        @echo ""
        @echo "Adding 47 bytes autoboot protocol header and swapping bytes"
        $(L1ROM) l1.mi

$(L1ROM): l1rom.c
        @echo ""
        @echo making $@
        $(CC) -o $@ -DSDRAM_BASE=$(SDRAM_BASE) -
DSDRAM_LIMIT=$(SDRAM_LIMIT) l1rom.c

# ------------------------------------------------------------------------

vivot.out: vivot.c
        $(TMCC) $(L2_CFLAGS) -o $@ vivot.c

vivot.mi: vivot.c
        $(TMCC) $(L2_CFLAGS) -o $@ -tmld $(L2_MIFLAGS) -- vivot.c

# ------------------------------------------------------------------------

clean:
        $(RM) -f $(L1ROM) *.obj *.o *.t *.i *.s *.eeprom *.out *.mi *.dump
```

l1main.c

```
/* Copyright (c) 1995,1996,1997 by Philips Semiconductors.
 * L1 boot code. Copies L2 code from a PCI-slave UVEPROM */

#include <tm1/mmio.h>

/* downloader symbols                                    */
/* Patched when creating a memory image file using tmld */

extern long          _clock_freq_init  [];
extern unsigned int  _begin_stack_init [];
extern unsigned int  _MMIO_base_init   [];

/* MACROS  */
```

```
#define  CACHE_BL_SIZE         64
#define  VO_FREQUENCY    27000000.0    /* 27 MHz */

/* globals */
unsigned long    _clock_freq = (unsigned long) _clock_freq_init;
volatile UInt32 *_MMIO_base  = (volatile UInt32 *) _MMIO_base_init;

custom_op void dcb      (unsigned, int);
custom_op void dinvalid (unsigned, int);
custom_op void iclr     (void);

/* local variables */
volatile static unsigned int dummy;

/* copyback_dcache (unsigned addr, int nbytes)
 * Addr must be cache aligned.
 * This function flushes nbytes starting at addr to memory.
 *
 * L1 boot code copies L2 code from some device. This needs to be flushed
 * to memory before jumping to the L2 load address */
static void
copyback_dcache( unsigned addr, int n ){
   int i;
   for( i = 0; i < n; i = i + CACHE_BL_SIZE ) dcb(0, addr + (unsigned) i);
}

/* iclr is in a separate funct. to ensure that it is in a dtree by itself */
static void
clear_icache(void){
  iclr();
}

/* Copies L2 code via JTAG to SDRAM */
unsigned int L1main (){
   int i;
   unsigned char  byte;
   unsigned int  *base_addr = (unsigned int *)  L2_ROM_DEV_ADDR;
   unsigned char *load_addr = (unsigned char *) L2_LOAD_ADDR;

#if 0
/* Not needed for TM1s 1.1 chip. In previous versions, autoboot leaves IIC in
 * stuck state. Steps 1, 2, and 3 will reset IIC. */

/* Step 1: Set up VO clock  */
   MMIO(VO_CLOCK) = (unsigned int)
                    (0.5 + (1431655765.0 * VO_FREQUENCY/_clock_freq));
   MMIO(VO_CTL) = 0x02700000;
/* and wait for vo clock to stabilize */
   for (i = 0; i < 1000 * 1000; i++) dummy++;

/* Step 2. Toggle I2C control */
   MMIO(IIC_CTL) = 0;
   MMIO(IIC_CTL) = 0x03c00001;

/* Step 3. Single I2C read and throw away */
   MMIO(IIC_AR) = 0x71000100;
   dummy = MMIO(IIC_DR);
#endif

/* Load L2 code from an attached PCI device  */

/* start copying of L2 code to sdram. Assumes TM1 debug board schematics.
```



```
 * Assumes L2 program is in a single UVEPROM plugged into byte 3 slot. The
 * other 3 slots (which supply bytes 0, 1, and 2 of a word loaded from PCI)
 * are empty. */
   for( i=0; i < L2_CODE_SIZE; i++ ){
#ifdef __BIG_ENDIAN__
      byte = base_addr[i] & 0xFF;
#else
      byte = (base_addr[i] >> 24) & 0xFF;
#endif
      load_addr[i] = byte;
   }

/* flush data cache */
   copyback_dcache(L2_LOAD_ADDR, L2_CODE_SIZE);

/* clear any interrupts */
   MMIO(ICLEAR)  = 0xffffffff;

   clear_icache();
/* Return from L1main() causes L2 code to be executed. */
   return L2_LOAD_ADDR;
}
```

l1rom.c

```
/* copyright (c) 1995,1996,1997 by Philips Semiconductors
 * Generates an EEPROM image (binary file)
 *
 * Input:
 *   f.mi - generated using -mi option of tmld
 *
 * Output:
 *    f.eeprom
 *    f.eeprom contains 47 header bytes as required by TM1 autoboot protocol,
 *    followed by the program bytes (bytes are swapped as required by boot)
 *
 * Assumption:
 *    1. f.mi contains less than 2001 bytes, divisible by four (as required
 *       by the boot protocol).
 *    2. short is 2 bytes.
 */
#if defined(__sun)
#include <unistd.h>
#endif
#include <sys/stat.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

#if !defined(SDRAM_BASE) || !defined(SDRAM_LIMIT)
#error "Macros SDRAM_BASE and SDRAM_LIMIT must be defined for this to build"
#endif

#define MAX_FILE_SIZE      2000
#define MAX_EPROM_SIZE     (1024 * 2)
#define BUF_SIZE           MAX_EPROM_SIZE
#define NUM_HEADER_BYTES   47

#define MSB_1ST(n)   (unsigned char)(((n) >> 24) & 0xff)
#define MSB_2ND(n)   (unsigned char)(((n) >> 16) & 0xff)
#define MSB_3RD(n)   (unsigned char)(((n) >> 8) & 0xff)
#define MSB_4TH(n)   (unsigned char)((n) & 0xff)

static void
basename( char *fname, char *bname ){
   char *ptr, *ptr2;
   int i;

   if( (ptr=strrchr(fname,'.')) == NULL ){
      strcpy(bname, fname);
   }else{
      for( ptr2 = fname, i = 0; ptr2 != ptr; ptr2++, i++ ){
         bname[i] = *ptr2;
      }
      bname[i] = '\0';
   }
}
int
read_file( unsigned char *buffer, char *filename ){
  FILE       *L1fp;
  int         L1fd, n, nbytes;
  struct stat file_stat;
```



```
   if( (L1fd = open(filename, O_RDONLY)) == -1 ){
      fprintf(stderr, "Unable to open file: %s\n", filename);
      fprintf(stderr, "File doesn't exist or not readable\n");
      exit(1);
   }
   if( fstat(L1fd, &file_stat) ){
      fprintf(stderr, "Unable to fstat file: %s\n", filename);
      exit(1);
   }
   nbytes = (unsigned long)file_stat.st_size;
   close(L1fd);

   if( (L1fp = fopen(filename, "rb")) == NULL ){
     fprintf(stderr, "Unable to open file: %s\n", filename);
     fprintf(stderr, "File doesn't exist or not readable\n");
     exit(1);
   }

   if( nbytes > MAX_FILE_SIZE ){
      fprintf(stderr, "File has %5d bytes. must be less than %5d bytes\n",
              nbytes, MAX_FILE_SIZE);
      exit(1);
   }
   n = fread(buffer, 1, nbytes, L1fp);
   if (n != nbytes) {
      fprintf(stderr, "Unable to read %5d bytes, error no: %5d\n",
              nbytes, errno);
      exit(1);
   }
   fprintf(stderr, "      Program Size: %5d bytes\n", nbytes);
   return nbytes;
}
/* Header bytes are hard-coded. Read the TM-1 boot block paper
 * to see what needs to go in here for AUTO boot. */
int
output_eeprom_header( int nbytes, unsigned char obuffer[] ){
  int i = 0;

/* Output eeprom header bytes 0 thru 46, as per Chapter 12 of TM 1000 Data
 * Book (April 1997 edition). These go into output array index 0 onwards. */

/* 0xc8 for 50 and 40 MHz TRI_CLKIN. 0xcc for 33 MHz */
   obuffer[i++] = 0xc8;     /* 0 */

/* Sub-system Id */
   obuffer[i++] = 0x00;     /* 1 */
   obuffer[i++] = 0x03;     /* 2 */

/* Sub-system Vendor Id */
   obuffer[i++] = 0x11;     /* 3 */
   obuffer[i++] = 0x31;     /* 4 */

/* Bytes 5 6 7: MM Config register */

/* Byte 6 and 4 bits of byte 7 determine refresh rate. The refresh rate is
 * 4c4 for 80MHz sdram clock, 384 for 60 Mhz. Use Table 11-10 Refresh
 * Intervals of TM 1000 Preliminary Data for other SDRAM clock speeds and
 * interpolate for speeds not mentioned in that table. */
   obuffer[i++] = 0x00;     /* 5 */
   obuffer[i++] = 0x4c;     /* 6 */
   obuffer[i++] = 0x44;     /* 7 */
```

```
/* Byte 8: PLL Ratios */
   obuffer[i++] = 0x00;     /* 8 */

/* Byte 9:  Most significant bit is 1 for stand-alone boot Least 3 bits of
 * byte 9 and 8 bits of byte 10 determine L1 boot program code size.
 * 11 bits == 2K bytes at most. */
   obuffer[i++] = (0x80 | ((nbytes >> 8) & 0x7));
   obuffer[i++] = (nbytes & 0xfc);

/* MMIO base register address, MSB first */
   obuffer[i++] = 0xef;                  /* 11 */
   obuffer[i++] = 0xf0;                  /* 12 */
   obuffer[i++] = 0x04;                  /* 13 */
   obuffer[i++] = 0x00;                  /* 14 */
/* MMIO base register value, MSB first */
   obuffer[i++] = 0xef;                  /* 15 */
   obuffer[i++] = 0xe0;                  /* 16 */
   obuffer[i++] = 0x00;                  /* 17 */
   obuffer[i++] = 0x00;                  /* 18 */

/* DRAM base register address, MSB first */
   obuffer[i++] = 0xef;                  /* 19 */
   obuffer[i++] = 0xf0;                  /* 20 */
   obuffer[i++] = 0x00;                  /* 21 */
   obuffer[i++] = 0x00;                  /* 22 */
/* DRAM base register value, MSB first */
   obuffer[i++] = MSB_1ST(SDRAM_BASE);   /* 23 */
   obuffer[i++] = MSB_2ND(SDRAM_BASE);   /* 24 */
   obuffer[i++] = MSB_3RD(SDRAM_BASE);   /* 25 */
   obuffer[i++] = MSB_4TH(SDRAM_BASE);   /* 26 */

/* DRAM limit register address, MSB first */
   obuffer[i++] = 0xef;                  /* 27 */
   obuffer[i++] = 0xf0;                  /* 28 */
   obuffer[i++] = 0x00;                  /* 29 */
   obuffer[i++] = 0x04;                  /* 30 */
/* DRAM limit register value, MSB first */
   obuffer[i++] = MSB_1ST(SDRAM_LIMIT);  /* 31 */
   obuffer[i++] = MSB_2ND(SDRAM_LIMIT);  /* 32 */
   obuffer[i++] = MSB_3RD(SDRAM_LIMIT);  /* 33 */
   obuffer[i++] = MSB_4TH(SDRAM_LIMIT);  /* 34 */

/* DRAM cacheable limit reg address, MSB first */
   obuffer[i++] = 0xef;                  /* 35 */
   obuffer[i++] = 0xf0;                  /* 36 */
   obuffer[i++] = 0x00;                  /* 37 */
   obuffer[i++] = 0x08;                  /* 38 */
/* DRAM cacheable limit reg value, MSB first */
/* (39) assumes to be the same as SDRAM_LIMIT */
   obuffer[i++] = MSB_1ST(SDRAM_LIMIT);  /* 39 */
   obuffer[i++] = MSB_2ND(SDRAM_LIMIT);  /* 40 */
   obuffer[i++] = MSB_3RD(SDRAM_LIMIT);  /* 41 */
   obuffer[i++] = MSB_4TH(SDRAM_LIMIT);  /* 42 */

/* DRAM base reg value, MSB first */
   obuffer[i++] = MSB_1ST(SDRAM_BASE);   /* 43 */
   obuffer[i++] = MSB_2ND(SDRAM_BASE);   /* 44 */
   obuffer[i++] = MSB_3RD(SDRAM_BASE);   /* 45 */
   obuffer[i++] = MSB_4TH(SDRAM_BASE);   /* 46 */

   if( i != NUM_HEADER_BYTES ){
      fprintf(stderr, "Error: header bytes count = %5d, shd be %5d\n",
```



```
         i, NUM_HEADER_BYTES );
      exit(1);
   }
   fprintf(stderr, "EEPROM Header Size: %5d bytes\n", NUM_HEADER_BYTES);
   return i;
}
int
main(int argc, char **argv){
   int            i, j, file_size;
   int            header_bytes;
   FILE          *fp;
   char          *o_file_name, *cp;
   unsigned char ibuffer[BUF_SIZE] = {0};
   unsigned char obuffer[BUF_SIZE] = {0};

   if (argc < 2) {
      fprintf(stderr, "Usage: l1prom file.mi \n");
      exit(1);
   }

/* find output file name */
   i = strlen(argv[1]);

/* .eeprom extension needs 7+1 chars */
   o_file_name = (char *)malloc(i+8);
   if( o_file_name == NULL ){
      fprintf(stderr, "unable to malloc\n");
      exit(1);
   }

/* skip all directory names */
   if( (cp = strrchr(argv[1], '/')) == NULL ){
      cp = argv[1];
   }
   basename(cp, o_file_name);
   i = strlen(o_file_name);
   o_file_name[i++] = '.';
   o_file_name[i++] = 'e';
   o_file_name[i++] = 'e';
   o_file_name[i++] = 'p';
   o_file_name[i++] = 'r';
   o_file_name[i++] = 'o';
   o_file_name[i++] = 'm';
   o_file_name[i++] = '\0';

   if( (fp = fopen(o_file_name,"wb")) == NULL ){
      fprintf(stderr, "Could not open (binary) file %s for write\n",
              o_file_name);
      exit(1);
   }
   file_size = read_file(ibuffer, argv[1]);

   header_bytes = output_eeprom_header(file_size, obuffer);

/* Output 4 bytes at a time. Swap the byte ordering since boot block expects
 * words in eeprom to have MSB first and LSB last. */
   for( i = header_bytes; i < file_size + header_bytes; i += 4 ){
      obuffer[i]   = ibuffer[i+3-header_bytes];
      obuffer[i+1] = ibuffer[i+2-header_bytes];
      obuffer[i+2] = ibuffer[i+1-header_bytes];
      obuffer[i+3] = ibuffer[i-header_bytes];
   }
```

```
    j = fwrite(obuffer, sizeof(char), file_size + header_bytes, fp);
    if (j != file_size + header_bytes) {
       fprintf(stderr, "Unable to write %5d bytes. Wrote %5d \n",
               file_size + header_bytes);
       exit(1);
    }
    close(fp);
    return 0;
}
```


l1start.trees

```
(* Copyright (c) 1995,1996,1997 by Philips Semiconductors. *)

(* L1 startup code *)
(* Copy this file to l1start.t and then compile as tmcc -x l1start.t *)

(* Compile this file with tmcc -x ....
 * The -x flag tells tmcc to run cpp on this file before assembly.
 * The -el or -eb option causes tmcc to define cpp flag
 *          __LITTLE_ENDIAN__ or __BIG_ENDIAN__
 * and the right INITIAL_PCSW_VALUE and INITIAL_BIU_CTL_VALUE get used.
 *
 * Running cpp on this file (via tmcc) also causes symbolic constants such as
 * BIU_CTL to be resolved. These are defined in TCS_INSTAL_DIR/tm1/mmio.h.*)

#define __TMAS__
#include <tm1/mmio.h>
(*------------------------------------------------------------------------*)
#ifdef __BIG_ENDIAN__
#define INITIAL_PCSW_VALUE     0x0800   (* S *)
#define INITIAL_BIU_CTL_VALUE  0x0200   (* Host Enable *)
#else
#define INITIAL_PCSW_VALUE     0x0A00   (* CS + Byte Sex *)
#define INITIAL_BIU_CTL_VALUE  0x0201   (* Host Enable + Byte Swap Enable *)
#endif
(*------------------------------------------------------------------------*)
   .text
   .global __start
   .global _L1main                 (* defined in l1main.c *)

__start:
___start_DT_0:
entree(0)
.treeinfo regmask "0x00000000000000000fffffffffffffff";

(* iclr just to be sure *)
   10 iclr;

   20 uimm (INITIAL_PCSW_VALUE);
   21 uimm (-1);
   22 writepcsw 20 21;

(* set up stack: FP and SP *)
   30 uimm (__begin_stack_init);
   33 wrreg (3) 30;
   34 wrreg (4) 30;

(* set up return pointer *)
   40 uimm(___start_DT_1);
   41 wrreg (2)  40;

(* configure BIU CTL *)
   50 uimm (BIU_CTL);
   51 uimm (__MMIO_base_init);
   52 iadd 50 51;
   53 uimm (INITIAL_BIU_CTL_VALUE);
   54 st32 52 53;

   gotree {_L1main}
endtree
```

```
(* Control returns to ___start_DT_1 when L1main() is done with loading
 * L2 code into SDRAM. Jump to L2_LOAD_ADDR returned in register 5 *)

___start_DT_1:
entree(0)
.treeinfo regmask "0x00000000000000000fffffffffffffff";
   12 rdreg (5);  (* L2 Load Address *)
   cgoto 12
endtree
```


Chapter 8

Bootstrapping TriMedia in Host-Assisted Mode

TriMedia Initialization in Host-Assisted Mode

The purpose of this document is to explain how the IREF board gets initialized in host assisted mode. In this mode the program is downloaded using the PCI bus. Host processor control over the program is ensured by writing to the PCI bus using the MMIO registers. Host assisted mode corresponds to a **tmcc** command line with the **-host** Win95, **-host** WinNT, or **-host** MacOS options.

The TriMedia processor begins in reset and is initialized as a result of actions by the host. The initial state of the processor is defined by the first 10 bytes of the EEPROM.

The processor state is initialized as the result of actions in several places. These include: the plug and play BIOS, the OS configuration manager, a kernel driver, and the user program. On Windows 95, the kernel driver is vtmman.vxd and the user program is tmgmon.exe.

The information in this document is useful for anyone that needs to understand the TriMedia processor at a systems level.

For more information about the TriMedia implementation of PCI, refer to the appropriate TriMedia data book. Figure 10-2 explains the PCI configuration registers. Chapter 12 describes the boot process. You may want to refer to the sections on the host assisted boot and on the EEPROM format.

You may also want to refer to the document PCI design Issues for Windows 95, and Windows NT, Microsoft Corporation, 1/25/95 (rev 1.0) for more information on PCI configuration in a PC environment.

For more information about the PCI local bus, refer to the PCI Local Bus Specification, version 2.1, available from the PCI consortium, tel: (503) 797 4207, fax (503) 234 6762.

For information about Microsoft Visual C++ (MSVC++) command line options, type:

```
cl /help
```

For information about Microsoft LINK command line options, type:

```
link /help
```

For information about the downloader library, see <tmlib/TMDownloader.h>.

For information about the TMMAN API, see Chapter 14, *TriMedia Manager API for Windows*, of Book 5, *System Utilities*, Part A. You can also refer to the <Win95/tmman32.h> header file in the release.

For information about the object file format and section types, refer to *tmld Options* in Chapter 11, *Linking TriMedia Object Modules*, of Book 4, *Software Tools*, Part B.

Overview

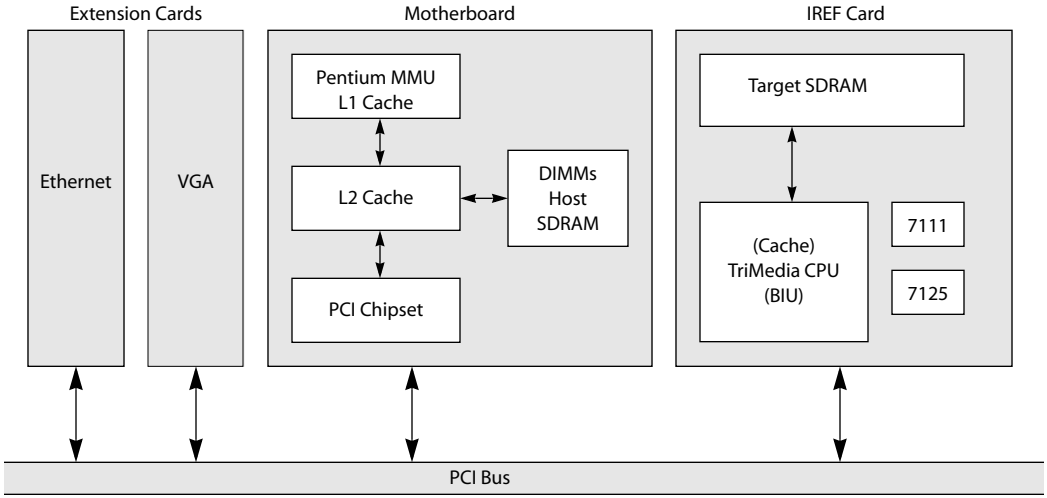

Figure 1 Overview of a Typical Host-Assisted System

Figure 1 gives an overview of a typical host-assisted system. The BIU (Bus Interface Unit) and the PCI chipset on the IREF board are equivalents.

In a host assisted system, the TriMedia processor is initialized over the PCI bus. A Pentium is being used in the example above.

Essentially, booting TriMedia in host assisted mode requires nothing more than loading a boot image into memory and taking the processor out of reset. This can be done by clearing BIU set reset and setting clear reset. However, several things do complicate matters.

First, the base address of the DRAM on the board and the MMIO registers is not fixed but determined at system startup. This is because of the plug and play nature of the PCI bus and it is done in the BIOS and the OS. Finding out the actual addresses assigned requires querying the OS configuration manager. Under Windows this is done by **tmman**.

Second, the DRAM and the MMIO on the board needs to be mapped in virtual memory for the Pentium processor to access it. Under Windows, this requires a kernel mode driver.

Third, the TriMedia downloader library must be used to construct the boot image. There are three reasons for this.

- The linker output is relocatable and needs to be made absolute.
- Symbols in the boot image needs to be patched for it to work. For example, for the processor to access its own registers MMIO_base needs to be patched.

- Special treatment is needed for cache locked and uncacheable memory and for shared sections for multiprocessors.

The downloader library depends on the object format library to read the executable. The same downloader is used to load boot images, applications and TriMedia dynamic linked libraries (dlls). For multiprocessors, processors are loaded individually and a shared section table is used for global addresses.

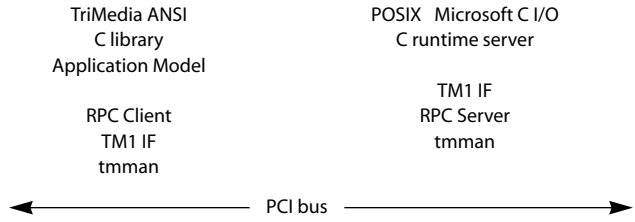

Figure 2 How System Calls on the Host are Implemented

Fourth, more functionality is required for system services (files, I/O) shows how system calls on the host are implemented.

I/O calls in the ANSI C library are mapped to system calls. They are transmitted as remote procedure calls (RPCs) using tmman to the host. On the host, the call is executed using the C run time server. Microsoft I/O is used for access to console windows and for files that can be redirected. Implementing RPC requires the ability to install an interrupt handler on the host.

The IPENDING, IMASK, and ICLEAR MMIO registers can be programmed on the host to generate a host to TriMedia interrupt dynamically. The interrupt vectors can be reprogrammed also. Interrupt pin A is used for TriMedia to host interrupts. For more information, see Chapters 3 and 10 of the data book.

The way downloading works means that a boot image that has been constructed in memory can be written to disk and executed simply by restoring and clearing reset.

Plug-and-Play BIOS

The PC BIOS allocates base addresses and interrupts for all cards using a technique called "plug and play." The interrupt vector is allocated by the plug-and-play BIOS also.

The following elements of the PCI configuration are significant. For retrieving these parameters, please refer to the PCI specification. The command **DOSPCI** in the bin directory of the Win95 release can be used to read the PCI registers. Here is an example.

```
[C:/Trimedia/bin] dospci

PCI Configuration Tool - Copyright (c) Philips Semiconductors 1996
PCI VendorID [1131] : DeviceID [5400] : Bus#[00] : Dev#[0e] : Func#[0]
PCI Reg#[00] : Offset[00] : Value [54001131]
PCI Reg#[01] : Offset[04] : Value [02000116]
```



```
PCI Reg#[02] : Offset[08] : Value [04800091]
PCI Reg#[03] : ...

[C:/Trimedia/bin]

PCI Register 0,  bits 0  .. 16 : vendor ID
                 bits 16 .. 31 : Device ID
```

The PCI specification identifies peripherals using a device **ID**, vendor **ID**. The device **ID** identifies the silicon. The device for TM-1000 is 5400 and the vendor ID is 1131 (Philips).

```
Register 2,  bits 0  .. 7  : Revision ID register
```

The 8 lower bits identify the CPU version (TM1, TM1S). Bits 7-6 indicate the fab. ST (Crolles) is 00, MOS4 is 01, TSMC, is 10, and 11 is unused. Bits 5-4 indicates the all layer revision. CTC/TM1 is 00, TM1S is 01, TM1C is 10, and 11 is unused. The four last bits indicates the metal layer revision. 0000 is revision 0.

```
Register 4,  bits 0  .. 31 : SDRAM Base Physical Address
Register 5,  bits 0  .. 31 : MMIO Base Physical Address
```

During startup, the card is accessed using the slot number of the PCI board in PCI configuration space. This is because the address is not allocated yet. PCI cards can have up to six base addresses. The TM-1000 IREF card has two (**MMIO_BASE**) and **DRAM_BASE**, corresponding to registers 4 and 5, above).

The necessary address range is determined as follows. The BIOS writes all 1's to these registers. The values that are read back tell the BIOS how much memory needs to be allocated, and the alignment to use. For example, writing FFFFFFFF and reading back FF000000 means that 16 megabytes need to be allocated. Natural alignment is enforced (e.g. 16 megabytes need to be allocated on a 16 megabyte boundary).

Win16 and DOS apps can query the PCI configuration space registers using the call int 1A. For more information, refer to the PCI BIOS specification. Kernel mode applications can query the Win95 configuration manager. **tmman** provides an API to query the address ranges (**tmDSPGetCaps**).

```
Register B,  bits 0  .. 15 SubSystem Vendor ID
             bits 16 .. 31 Subsystem ID
```

These two fields identify the manufacturer and subsystem ID (board ID). They correspond to bytes 1–4 of the EEPROM. They can be used by software to distinguish different boards. Board manufacturers should request a SubSystem Vendor ID from Philips.

To find out how to obtain a board ID, contact TriMedia Customer Support. Once an ID has been maintained, management of the subsystem space is the board manufacturers responsibility.

```
Register f,  bits 0  .. 7  : Interrupt line register.
```

This determines the value to use for host interrupts. Note that TMMAN does not support sharing of interrupts.

The value in registers 4 and 5, and F are allocated by the PCI BIOS as part of the setup. The values allocated by the BIOS are used by the Win95 configuration manager. The

exact way this is done is documented in *PCI Design Issues for Windows 95 and Windows NT* (Microsoft Corporation, 1/25/95) (document is available from Microsoft).

Depending on the BIOS, and exact PC configuration, plug and play may not always work. In this case, the configuration needs to be changed so that these are not allocated automatically. This is done using the Windows 95 Device Manager (Start–> Settings–> Control Panel–> System–> Device Manager). To disable automatic selection of the base address, the Resources menu needs to be selected and changed. Both the base address and the interrupt number may need to be allocated manually.

BIU and Interrupt Initialization

The TM-1 processor comes up in big endian mode. Depending on the endianness of the host processor, the BIU control register needs to be reconfigured. After this write, all further accesses should be done in big endian format.

On a little endian processor, this write has to be done in big endian format. The **BYTESWAP** macro converts the ordering.

```
#define BYTESWAP(x)
    ((x) << 24 | ((x) &0xFF00) << 16 | ((x) & 0xFF0000) >> 8 |    \
                 ((x) & 0xFF000000) >> 24 )

VOID halRegisterInit ( PVOID pvObject, DWORD dwSDRAMPhys,
                       DWORD dwSDRAMCacheLimit, DWORD dwMMIOPhys ){
   MMIO.pVIC      = dwMMIOBase + 0x100800;
   MMIO.pTimers   = dwMMIOBase + 0x100c00;
   MMIO.pDebug    = dwMMIOBase + 0x101000;
   MMIO.pBIU      = dwMMIOBase + 0x103004;
   MMIO.pAudioIn  = dwMMIOBase + 0x101c00;
   MMIO.pAudioOut = dwMMIOBase + 0x102000;
   MMIO.pCache    = dwMMIOBase + 0x100000);
```

This initializes pointers to the MMIO registers of the different peripherals.

```
FirstTimeReset = !(MMIO.pBIU->dwBIUControl & (BIU_SE|BIU_HE));
```

The Windows 95 driver checks whether the BIU control register HE and SE bits are set. If these bits are not set, it assumes that we are just doing a reboot and that none of the registers needs to be initialized.

```
if( FirstTimeReset)
   MMIO.pBIU->dwBIUControl = BYTESWAP (BIU_SE |  BIU_HE | BIU_SR );
```

This turns on the BIU byte swap enable bit, host enable, and set reset bits.

```
MMIO.pCache->dwDRAMCacheableLimit = dwSDRAMPhys + dwSDRAMSize;
MMIO.pCache->dwDRAMLimit          = dwSDRAMPhys + dwSDRAMSize;
```

The DRAM Limit and DRAM Cacheable Limit registers are set to the end of memory.

```
MMIO.pVIC->dwIMask  = 0;
MMIO.pVIC->dwIClear = 0xFFFFFFFF;
```

Writing zeroes to the **IMASK** registers ensures that all interrupts are off. Note that if interrupts need to be generated from the host to the TM processor, then the relevant bits in

IMASK need to be set. Writing all ones to the **ICLEAR** register ensures that all the pending interrupts are cleared. For more information. see Figure 3-7 of the data book.

Putting the Processor in Reset

In what follows, figure references are to the databook.

The following code from <tmhal.c> puts the processor in a reset state.

```
VOID   halDSPStop ( PVOID pvObject ) {
   MMIO.pVIC->dwIMask       = (0x0);
   MMIO.pVIC->dwIClear      = 0xffffffff;
   MMIO.pBIU->dwBIUControl &= (~BIU_CR);
   MMIO.pBIU->dwBIUControl |= BIU_SR;
```

Interrupts are masked and pending interrupts are cleared (Figure 3-7 in the data book). Turning off CR (clear reset) and turning on set reset clears the reset.

```
*((PDWORD)(MMIO.pSpace + AO_CTL )) = 0x80000000;
*((PDWORD)(MMIO.pSpace + AO_FREQ)) = 0;
```

This resets audio out. PDWORD is a Windows type for a pointer to a double word (32 bits). See Figure 9-6 of the data book.

```
/* audio in AI_CTL */
*((PDWORD)(MMIO.pSpace + AI_CTL )) = 0x80000000;
*((PDWORD)(MMIO.pSpace + AI_FREQ)) = 0;
```

This resets audio in. Generally speaking, the most significant bit in the control register for a peripheral is reset. See Figure 8-5 of the data book.

```
*((PDWORD)(MMIO.pSpace + VI_CTL  )) = 0x00080000;
*((PDWORD)(MMIO.pSpace + VI_CLOCK)) = 0;
*((PDWORD)(MMIO.pSpace + VO_CTL  )) = 0x80000000;
*((PDWORD)(MMIO.pSpace + VO_CLOCK)) = 0;
```

This resets video in and video out. See Figures 6-11 and 7-26 of the data book.

```
*((PDWORD)(MMIO.pSpace + SSI_CTL)) = 0xc0000000;
*((PDWORD)(MMIO.pSpace + SSI_CTL)) = (1 << 18);
```

The first instruction resets the Synchronous Serial Interface (SSI). The two upper most bits correspond to Transmitter reset and receiver reset. The second leaves the phone on hook after reset. See figure 16-1 in the data book.

The following code resets the ICP. The loop is executed 10 times, just to make sure.

```
for ( Idx= 0 ; Idx < 10 ; Idx ++ ) {
   if( (*((PDWORD)(MMIO.pSpace + ICP_SR )) & 0x01) ) break;
   ( *((PDWORD)(MMIO.pSpace + ICP_SR )) ) = 0x80;
}
```

The least significant bit (LSB) corresponds to ICP busy. If the ICP is busy executing microcode there is no reset. The assignment resets the ICP internal registers on reset. See Figure 13-17 in the data book.

```
*((PDWORD)(MMIO.pSpace + IIC_CTL )) = 0;
```

The IIC bus is disabled (figure 15-2 in the data book).

```
*((PDWORD)(MMIO.pSpace + VLD_COMMAND )) = 0x00000401;
```

The writes a reset command with a count of 1 to the VLD. See Chapter 14 in the data book.

```
*((PDWORD)(MMIO.pSpace + TIMER1_TCTL  )) &= ~0x1;
*((PDWORD)(MMIO.pSpace + TIMER2_TCTL  )) &= ~0x1;
*((PDWORD)(MMIO.pSpace + TIMER3_TCTL  )) &= ~0x1;
*((PDWORD)(MMIO.pSpace + SYSTIMER_TCTL)) &= ~0x1;
```

The RUN bit is turned off in the four timer control registers.

```
*((PDWORD)(MMIO.pSpace + BICTL)) = 0;
*((PDWORD)(MMIO.pSpace + BDCTL)) = 0;
```

The instruction and data breakpoints are turned off. See Figures 3-10 and 3-13 in the data book.

```
  *((PDWORD)(MMIO.pSpace + JTAG_DATA_IN))= 0x0;
  *((PDWORD)(MMIO.pSpace + JTAG_DATA_OUT))= 0x0;
  *((PDWORD)(MMIO.pSpace + JTAG_CTL))= 0x04;
```

The JTAG data registers and full bits are cleared. The JTAG interface is put in sleepless mode. See figure 17-3 in the data book.

Taking the Processor Out of Reset

Once the program has been loaded, the following code from tmhal.c will begin initialization.

```
VOIDhalDSPStart ( PVOID pvObject )
{

  MMIO.pVIC->dwIMask =  0;
  MMIO.pVIC->dwIClear = 0xFFFFFFFF;
```

This disables interrupts and clear all pending interrupts See Figure 3-7 in the data book.

```
  MMIO.pBIU->dwBIUControl &= ~BIU_SR;
  MMIO.pBIU->dwBIUControl |= BIU_CR;
```

Turning off SR (set reset) and turning on CR (clear reset) in the Bus Interface Unit (BIU) takes the processor out of reset See Figure 10.6.4 in the data book.

The following code determines whether the TriMedia processor is running or not.

```
BOOLhalIsTMRunning (void) {
return (  ( MMIO.pBIU->dwBIUControl & BIU_SR ) ==  0 );
}
```

tmmprun—Multiprocessor Download Program

tmmprun is intended to run as an independent executable to be started from a PC command line. Its first argument is the name of a TM-1 executable to be downloaded, started, and which is passed all additional command line arguments. Between starting

the executable and receiving its termination message, the module behaves as a server for the HostCall interface.

tmmprun is the driver for the host part of the PC version of the Level 2 Remote Procedure Call Server (RPCserv). An implementation of this is needed for each particular host of a TM-1 board which uses TCS's generic ANSI C library.

The **tmmprun** source consists of one C source file, tmmprun.c.

tmmprun—Program Source Listing

Following is the listing of the program **tmmprun** through which you can know how to use the **tmman**, **tmcrt** and downloader library to download into a multiprocessor system using shared memory.

```
/* COPYRIGHT (c) 1997 by Philips Semiconductors
* This module is driver for the host part of the PC version of Level 2
* Remote Procedure Call Server. An implementation of this is needed for
* each particular host of a TM-1 board which uses TCS's generic ANSI C
* library.
*
* In its current form this module is intended to run as an independent
* executable that must be started from a PC command line. Its first
* argument must be the name of a TM-1 executable which is to be
* downloaded, started, and passed all additional command line arguments.
* Between starting the executable and receiving its termination message,
* this module behaves as a server for the HostCall interface.
*/
#include "windows.h"
#include "stdio.h"
#include "winioctl.h"
#include "tmmanapi.h"
#include "tmcrt.h"
#include "tmif.h"
#include "TMDownLoader.h"
DWORD    GlobalExitCode = (~0x0);
BOOL     Interactive = TRUE;
PCHAR    TargetExecutableName = NULL;
DWORD    TargetArgumentOffset;
BOOL     WINAPI tmrunControlHandler(DWORD dwCtrlType);

#define  MAXIMUM_NODES                4
#define  MAXIMUM_COMMAND_LINE_ARGS  100

DWORD    DSPCount = 0;
HANDLE   EventArray[MAXIMUM_NODES];
TCHAR    szTemp[2048];

TMStatus tmDSPExecutableLoadEx (
   DWORD                            DSPHandle,
   PCHAR                            pszImagePath,
   DWORD                            NumberOfDSPs,
   TMDwnLdr_SharedSectionTab_Handle SharedSections,
   PDWORD                           MMIOPhysicalAddressArray
);
void     WarningBox ( TCHAR* ErrorString );
HANDLE   tmmanGetDriverHandle ( VOID );
```

```
#define tmmanDownloaderStatusToTMManStatus(x) \
   (((x)!=TMDwnLdr_OK)?(x+0x70):statusSuccess)

/*************************************************************************/
// For details on TMDwnLdr functions, refer to the chapter "TriMedia "
// Downloader API" in the book "Software Library APIs. For details on
// tmman & cruntime functions, refer to the chapter "TriMedia Manager
// API" in the book "Software Library APIs"
int
main( int argc, char *argv[] ){
   TMStatus      Status;
   tmmanVersion  Version;
   tmmanDSPInfo  DSPCaps;
   COORD         ConsoleSize;
   DWORD         wIdxArg;
   DWORD         CRTHandle[MAXIMUM_NODES];
   DWORD         DSPHandle[MAXIMUM_NODES];
   DWORD         ArgumentCountArray[MAXIMUM_NODES];
   PVOID         ArgumentVectorArray[MAXIMUM_NODES];
   DWORD         MMIOPhysicalAddressArray[MAXIMUM_NODES];
   CHAR*         ArgumentVector[MAXIMUM_COMMAND_LINE_ARGS];
   DWORD         IdxNode;
   DWORD         ArgumentCount;

   TMDwnLdr_SharedSectionTab_Handle SharedSections;

   if( argc == 1 ){
      goto mainUsage;
   }


/* check for compatible driver version */
   Version.Major = constTMManDefaultVersionMajor;
   Version.Minor = constTMManDefaultVersionMinor;

   tmmanNegotiateVersion( constTMManDefault, &Version );

   if( (Version.Major != constTMManDefaultVersionMajor) ||
       (Version.Minor != constTMManDefaultVersionMinor) )
   {
      fprintf( stderr, "TMMPRun : ERROR : tmmprun.exe Version[%d.%d] is \
               INCOMPATIBLE With TMMan32.dll Version[%d.%d]\n",
               constTMManDefaultVersionMajor, constTMManDefaultVersionMinor,
               Version.Major, Version.Minor );
      goto mainExit1;
   }


// install a control C handler so we can perform cleanup before exit.
   if( SetConsoleCtrlHandler( tmrunControlHandler, TRUE) != TRUE){
      fprintf( stderr,
         "TMMPRun : ERROR : Win32 SetConsoleCtrlHandler failed [0x%x]",
         GetLastError());
      goto mainExit1;
   }

// count the number of DSPs into which we must download.
// this corresponds to the number of "-exec" in the command line
   for( wIdxArg = 1 ; wIdxArg < argc; wIdxArg++ ){
      // "-exec" indicates start of target executable name and arguments
      if( _stricmp ( argv[wIdxArg], "-exec" ) == 0 ){
         DSPCount++;
```



```
      }
   }

// initialize the arrays
   for ( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      DSPHandle[IdxNode] = NULL;
   }
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      CRTHandle[IdxNode] = NULL;
   }
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      ArgumentVectorArray[IdxNode]  = NULL;
   }
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      EventArray[IdxNode] = NULL;
   }

// open all the DSPs
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      Status = tmmanDSPOpen(IdxNode, &DSPHandle[IdxNode]);
      if( Status != statusSuccess ){
         fprintf(stderr,
            "TMMPRun : ERROR : tmmanDSPOpen failed [0x%x](%s)\n",
            Status, tmmanGetErrorString(Status) );
         goto mainExit2;
      }
      Status = tmmanDSPGetInfo ( DSPHandle[IdxNode] , &DSPCaps );
      if( Status != statusSuccess ){
         fprintf(stderr,
            "TMMPRun : ERROR : tmmanDSPGetInfo failed [0x%x](%s)\n",
            Status, tmmanGetErrorString(Status) );
         goto mainExit3;
      }
      MMIOPhysicalAddressArray[IdxNode] = DSPCaps.MMIO.PhysicalAddress;
   }

// process the generic command line parameters here
   for( wIdxArg = 1; wIdxArg < argc; wIdxArg++ ){
      // "-exec" indicates start of target executable name and arguments
      if( _stricmp(argv[wIdxArg],"-exec") == 0 ){
         // the next argument should be the name of the executable.
         wIdxArg++;    // Point to it.
         break;
      }
      switch( toupper(argv[wIdxArg][1]) ){
      // interactive off option
         case 'B':         Interactive = FALSE;
                           break;
      // no of lines in console window
         case 'W': {
            DWORD   dwWindowLines;
            COORD   ConsoleSize;
            if( sscanf(&argv[wIdxArg][2], "%d", &dwWindowLines) != 1 ){
               goto mainUsage;
            }
            ConsoleSize.X = 80;
            ConsoleSize.Y = (USHORT)dwWindowLines;
            SetConsoleScreenBufferSize(
               GetStdHandle(STD_OUTPUT_HANDLE), ConsoleSize);
         }
         break;
      // add other command line options here
```

```
         case '?':
         default:
            goto mainUsage;
      }
   }

////////////////////////////////////////////////////////////////////////
// create an empty shared section table for use in multiprocessing
// downloading
   TMDwnLdr_create_shared_section_table(&SharedSections);

// initialize the C Run Time server to serve multiple nodes.
   cruntimeInit();

// initialize the C runtime for each DSP
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      CRunTimeParameterBlock CRTParam;
      DWORD                  i;

   // create an event for this node
      EventArray[IdxNode] = CreateEvent(
         NULL,     //default security descriptor in NT, not used in Win95
         FALSE,    //auto reset event
         FALSE,    //initial state is not signalled
         NULL);
      if( EventArray[IdxNode] == INVALID_HANDLE_VALUE ){
         fprintf(stderr,
            "TMMPRun : ERROR : Win32 CreateEvent failed [0x%x]\n",
            GetLastError());
         goto mainExit4;
      }
      CRTParam.OptionBitmap = 0;
      CRTParam.StdInHandle  = (DWORD) GetStdHandle(STD_INPUT_HANDLE );
      CRTParam.StdOutHandle = (DWORD) GetStdHandle(STD_OUTPUT_HANDLE);
      CRTParam.StdErrHandle = (DWORD) GetStdHandle(STD_ERROR_HANDLE );

   // process target specific command line arguments here
   // this parameter should be the executable filename
      TargetExecutableName = argv[wIdxArg];
      ArgumentCount = 0;
      ArgumentVector[ArgumentCount++] = argv[wIdxArg];
      for( wIdxArg++ ; wIdxArg < argc; wIdxArg++){
         // "-exec" indicates start of target executable name and
         // arguments
         if( _stricmp(argv[wIdxArg],"-exec") == 0 ){
            wIdxArg++;
            break;
         }
         ArgumentVector[ArgumentCount++] = argv[wIdxArg];
      }
      ArgumentVector[ArgumentCount] = NULL;

   // we have to allocate persistent storage for these values
      ArgumentCountArray[IdxNode] = ArgumentCount;
      if( (ArgumentVectorArray[IdxNode] =
         malloc(sizeof(PVOID) * (ArgumentCount + 1) ) ) == NULL )
      {
         fprintf(stderr,
         "TMMPRun : ERROR : malloc(Argument Array) failed\n" );
         goto mainExit5;
      }
      memcpy( ArgumentVectorArray[IdxNode],
```



```
         ArgumentVector,
         sizeof(PVOID) * (ArgumentCount + 1) );

   // PERFORM RELOCATION HERE ////////////////////////
      Status = tmDSPExecutableLoadEx (
         DSPHandle[IdxNode],
         TargetExecutableName,
         DSPCount,
         SharedSections,
         MMIOPhysicalAddressArray );
   ////////////////////////////////////////////////////

      if( Status != statusSuccess ){
         fprintf(stderr,
            "TMMPRun : ERROR : tmDSPExecutableLoadEx(%s) failed \
            [Ox%x](%s)\n",
            TargetExecutableName, Status, tmmanGetErrorString(Status));
         goto mainExit6;
      }
      CRTParam.OptionBitmap |= constCRunTimeFlagsUseSynchObject;
      CRTParam.SynchronizationObject = (DWORD) EventArray[IdxNode];
      CRTParam.VirtualNodeNumber = IdxNode;

      if( !Interactive ){
         CRTParam.OptionBitmap |= constCRunTimeFlagsNonInteractive;
      }

   // allocate resources for this TriMedia processor.
      if( cruntimeCreate(
          IdxNode,                      /* the  physical DSP Number */
          ArgumentCountArray[IdxNode],
          ArgumentVectorArray[IdxNode],
          &CRTParam,
          &CRTHandle[IdxNode] ) != True )
      {
         fprintf(stdout,
                "TMMPRun : ERROR : Cannot Initialize C Run Time Server : CRT \
                I/O calls will not work\n");
         goto mainExit6;
      }
   }
// At this point, code has been copied to the memory of the TriMedia boards
// and C runtime has been created. Start executing the code on each DSP.
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      Status = tmmanDSPStart( DSPHandle[IdxNode] );
      if( Status != statusSuccess ){
         fprintf(stderr,
            "TMMPRun : ERROR : tmmanDSPStart failed [Ox%x](%s)\n",
            Status, tmmanGetErrorString(Status) );
         goto mainExit7;
      }
   }

// wait until all objects in the EventArray array are signaled.
   WaitForMultipleObjects (DSPCount, EventArray, TRUE, INFINITE );

/* mainExit7: */
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      if( DSPHandle[IdxNode] ) tmmanDSPStop ( DSPHandle[IdxNode] );
   }
mainExit7:
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
```

```
      if( CRTHandle[IdxNode] )
         cruntimeDestroy(CRTHandle[IdxNode], &GlobalExitCode);
   }
mainExit6:
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      if( ArgumentVectorArray[IdxNode] )
         free ( ArgumentVectorArray[IdxNode] );
   }
mainExit5:
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      if( EventArray[IdxNode] )
         CloseHandle( EventArray[IdxNode] );
   }

mainExit4:
   cruntimeExit();
   TMDwnLdr_unload_shared_section_table(SharedSections);

mainExit3:
   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      if( DSPHandle[IdxNode] )
         tmmanDSPClose ( DSPHandle[IdxNode] );
   }

mainExit2:
   SetConsoleCtrlHandler(tmrunControlHandler, FALSE);

mainExit1:
   if( Interactive ){
      fprintf(stdout, "\nTMMPRun:Press [ENTER] to close server >>");
      getchar();
   }
   return (GlobalExitCode);

mainUsage:
   fprintf(stderr, "usage:TMMPRun [-wWindowSize] [-b] -exec \
   ExecutableImageName [Arg1] [Arg2] ... -exec ExecutableImageName [Arg1] \
   [Arg2] ...\n");
   goto mainExit1;
}

// control C handler
   BOOL WINAPI
   tmrunControlHandler( DWORD dwCtrlType ){
   DWORD   IdxNode;

   fprintf(stderr,
      "\nTMMPRun:Control C Detected : Performing Cleanup\n");

   for( IdxNode = 0 ; IdxNode < DSPCount ; IdxNode++ ){
      SetEvent(EventArray[IdxNode]);
   }
   return TRUE;
}

// function to download the image in DSP
TMStatus   tmDSPExecutableLoadEx (
   DWORD                              DSPHandle,
   PCHAR                              pszImagePath,
   DWORD                              NumberOfDSPs,
   TMDwnLdr_SharedSectionTab_Handle   SharedSections,
   PDWORD                             MMIOPhysicalAddressArray )
```



```
{
   tmmanDSPInfo             DSPInfo;
   tmmanDSPInternalInfo     DSPInternalInfo;
   TMStatus                 Status = statusSuccess;
   tmifDSPLoad              TMIF;
   TMDwnLdr_Object_Handle   ObjectHandle;
   UInt32                   ImageSize;
   UInt32                   Alignment;
   DWORD                    AlignedDownloadAddress;
   UInt32                   DownLoaderStatus;
   Endian                   endian;
   UInt32                   HostType = tmWinNTHost;
   UInt32                   BytesReturned;
   UInt32                   TargetVersion;

   UInt32   ClockSpeed  = 100000000;                          // default
   UInt32   CacheOption = TMDwnLdr_LeaveCachingToDownloader;  // default

   UInt32   INIEndianess = True;     // little endian
   HKEY     RegistryHandle;
   HKEY     RegistryHandleDevice;
   TCHAR    szDeviceName[0x10];

// map SDRAM into the Operating System and Process virtual address space
   if( (Status = tmmanDSPMapSDRAM(DSPHandle)) != statusSuccess ){
      OutputDebugString ( TEXT("TMMPRun:tmmanDSPMapSDRAM:FAIL\n"));
      goto tmDSPExecutableLoadExit1;
   }
   tmmanDSPGetInfo        ( DSPHandle, &DSPInfo         );
   tmmanDSPGetInternalInfo( DSPHandle, &DSPInternalInfo );

   if( ERROR_SUCCESS == RegOpenKeyEx(
                        HKEY_LOCAL_MACHINE,
                        constTMManRegistryPath,
                        0,
                        KEY_READ,
                        &RegistryHandle) )
   {
      ULONG   BytesXfered;

      BytesXfered = sizeof ( ULONG );
      if( ERROR_SUCCESS != RegQueryValueEx(
                           RegistryHandle,
                           TEXT("DefaultEndianness"),
                           NULL,
                           NULL,
                           (BYTE*)&INIEndianess,
                           &BytesXfered) )
      {
         INIEndianess = True;
      }
      wsprintf( szDeviceName, TEXT("Device%x"), DSPInfo.DSPNumber );
      if( ERROR_SUCCESS == RegOpenKeyEx(
                           RegistryHandle,
                           szDeviceName,
                           0,
                           KEY_READ,
                           &RegistryHandleDevice) )
      {
         BytesXfered = sizeof ( ULONG );
         if( ERROR_SUCCESS != RegQueryValueEx(
                              RegistryHandleDevice,
```

```
                              TEXT("ClockSpeed"),
                              NULL,
                              NULL,
                              (BYTE*)&ClockSpeed,
                              &BytesXfered) )
         {
            ClockSpeed = 100000000;
         }
         BytesXfered = sizeof ( ULONG );
         if( ERROR_SUCCESS != RegQueryValueEx(
                              RegistryHandleDevice,
                              TEXT("CacheOption"),
                              NULL,
                              NULL,
                              (BYTE*)&CacheOption,
                              &BytesXfered) )
         {
            CacheOption = TMDwnLdr_LeaveCachingToDownloader;
         }
         RegCloseKey ( RegistryHandleDevice );
      }
      RegCloseKey ( RegistryHandle );
   }
// read the executable from file into memory,
// and use the handle returned for subsequent operations
   if( (DownLoaderStatus = TMDwnLdr_load_object_from_file (
        pszImagePath,
        SharedSections,
        &ObjectHandle ) ) != TMDwnLdr_OK )
   {
      Status = tmmanDownloaderStatusToTMManStatus(DownLoaderStatus);
      goto tmDSPExecutableLoadExit2;
   }

// BEGIN symbol patching
   if( (DownLoaderStatus = TMDwnLdr_resolve_symbol(
        ObjectHandle,
        "_TMManShared",
        DSPInternalInfo.TMManSharedPhysicalAddress ) ) != TMDwnLdr_OK )
   {
      OutputDebugString( TMDwnLdr_get_last_error (DownLoaderStatus) );
      HostType = tmNoHost;
      Status = tmmanDownloaderStatusToTMManStatus(DownLoaderStatus);
      WarningBox(TEXT("Target Executable has not been linked with \
[-host \ WinNT] or [-host Windows]\n"));
   }


// get the extracted image size, and its required alignment in SDRAM
   if( (DownLoaderStatus = TMDwnLdr_get_image_size(ObjectHandle,
        &ImageSize, &Alignment)) != TMDwnLdr_OK )
   {
      Status = tmmanDownloaderStatusToTMManStatus(DownLoaderStatus);
      OutputDebugString ( TMDwnLdr_get_last_error (DownLoaderStatus) );
      goto tmDSPExecutableLoadExit3;
   }
   AlignedDownloadAddress =
   ( (DSPInfo.SDRAM.PhysicalAddress + Alignment-1) & ( ~(Alignment-1) ) );

// relocate the loaded executable into a specified TM1 address range,
// with specified values for MMIO_base and TM1_frequency.
   if( (DownLoaderStatus = TMDwnLdr_multiproc_relocate (
```



```
        ObjectHandle,
        HostType,
        (Address*)MMIOPhysicalAddressArray,
        DSPInfo.DSPNumber, // NodeNumber
        NumberOfDSPs,// NumberOfNodes
        ClockSpeed,
        (Address)AlignedDownloadAddress,
        DSPInfo.SDRAM.Size,
        CacheOption ) ) != TMDwnLdr_OK )
   {
      OutputDebugString ( TMDwnLdr_get_last_error (DownLoaderStatus) );
      Status = tmmanDownloaderStatusToTMManStatus(DownLoaderStatus);
      goto tmDSPExecutableLoadExit3;
   }

// Get the endianness of the specified loaded object
   if( (DownLoaderStatus = TMDwnLdr_get_endian(
        ObjectHandle,
        &endian ) ) != TMDwnLdr_OK )
   {
      OutputDebugString( TMDwnLdr_get_last_error (DownLoaderStatus) );
      Status = tmmanDownloaderStatusToTMManStatus(DownLoaderStatus);
      goto tmDSPExecutableLoadExit3;
   }
   if( INIEndianess != endian ){
      Status = statusExecutableFileWrongEndianness;
      goto tmDSPExecutableLoadExit3;
   }
   TMIF.DSPHandle = DSPHandle;
   TMIF.Endianess =
      (endian==LittleEndian) ?
      constTMManEndianessLittle : constTMManEndianessBig;

   if( (TMDwnLdr_get_contents(
        ObjectHandle,
        "__TMMan_Version",
        &TargetVersion ) ) != TMDwnLdr_OK )
   {
      TMIF.PeerMajorVersion = constTMManDefaultVersionMajor;
      TMIF.PeerMinorVersion = constTMManDefaultVersionMinor;
   } else {
      // major version = __TMMan_Version[31:16]
      // minor version = __TMMan_Version[15:0]
      TMIF.PeerMajorVersion = ( (TargetVersion & 0xffff0000 ) >> 16 );
      TMIF.PeerMinorVersion = TargetVersion & (0x0000ffff);
      if( (TMIF.PeerMajorVersion != constTMManDefaultVersionMajor ) ||
          (TMIF.PeerMinorVersion != constTMManDefaultVersionMinor ) )
      {
         wsprintf ( szTemp, TEXT("Target Executable Version [%d.%d] is \
INCOMPATIBLE with TriMedia Driver Version [%d.%d]\n"),
         TMIF.PeerMajorVersion, TMIF.PeerMinorVersion,
         constTMManDefaultVersionMajor, constTMManDefaultVersionMinor );
         switch(
            (MessageBox(NULL,
            szTemp,
            TEXT("TriMedia Manager : TMMPRun.exe : Continue ? "),
            MB_OKCANCEL|MB_ICONQUESTION|MB_DEFBUTTON1|MB_APPLMODAL)) )
         {
            case IDOK:
               break;
            case IDCANCEL:
            default:
```

```
                Status =
                (TMIF.PeerMajorVersion != constTMManDefaultVersionMajor) ?
                statusMajorVersionError : statusMinorVersionError;
                goto tmDSPExecutableLoadExit3;
            }
        }
    }
// call to kernel mode driver
    if( DeviceIoControl ( tmmanGetDriverHandle(),
         constIOCTLtmmanDSPLoad,
         (PVOID)&TMIF, sizeof( tmifDSPLoad),
         (PVOID)&TMIF, sizeof( tmifDSPLoad),
         &BytesReturned, NULL) != TRUE )
    {
       Status = TMIF.Status;
       goto tmDSPExecutableLoadExit3;
    }
    if( (DownLoaderStatus = TMDwnLdr_get_memory_image(
         ObjectHandle,
         (UInt8*)tmmanPhysicalToMapped( &DSPInfo. SDRAM,
         AlignedDownloadAddress))) != TMDwnLdr_OK )
    {
       OutputDebugString ( TMDwnLdr_get_last_error (DownLoaderStatus) );
       Status = tmmanDownloaderStatusToTMManStatus(DownLoaderStatus);
       goto tmDSPExecutableLoadExit3;
    }

tmDSPExecutableLoadExit3:
    TMDwnLdr_unload_object ( ObjectHandle );

tmDSPExecutableLoadExit2:
    tmmanDSPUnmapSDRAM ( DSPHandle );

tmDSPExecutableLoadExit1:
    return Status;
}

void   WarningBox ( TCHAR* ErrorString ){
    MessageBox( NULL,
                ErrorString,
                TEXT("TriMedia Manager : tmmprun.exe : WARNING"),
                MB_OK|MB_ICONWARNING|MB_DEFBUTTON1|MB_APPLMODAL );
}
```


Chapter 9

Bootstrapping TriMedia from Flash

Note
The Flash File System is not included with the basic TriMedia SDE, but is available under a separate licensing agreement. Please contact your TriMedia sales representative for more information.

Introduction

Using the TriMedia Flash File System Manager for storing boot images on flash memory, a simple modification to the L1 boot procedure is sufficient to let TriMedia boot from flash. (The standard L1 boot procedure is described in Chapter 7.)

This chapter describes this modified boot procedure, and how to set up the flash memory so that it can be used for autonomous booting. The TriMedia Flash File System Manager is fully described in Chapter 10, *TriMedia Flash File System API*, of Book 5, System Utilities, Part A.

Setting Up Flash-Based Booting

Setting up a board so that it boots from flash generally involves the following steps:

1. Choose the endian in which the board will run; the flash file system format is endian dependent. For example:

   ENDIAN= -el

2. Choose a flash driver that corresponds with the board. A flash driver defines some basic flash characteristics, like flash block size, number of flash blocks, and how to write to- and read from flash, and how to erase flash blocks. Directory *$TCS/examples/flash_file_system/sample_drivers* contains a number of flash drivers for standard Philips boards, and new drivers can best be based upon these sample drivers. The chosen flash driver must, at a minimum, correspond to requirements given in *Flash Driver Boot Specification* in Book 5, *System Utilities*, Part A. For example:

   $TCS/examples/flash_file_system/sample_drivers/FlashSpecificGomad.c

3. Create the flash-based L1 boot image that is to be stored in the L1 boot EEPROM. This is a simple modification of the standard L1 boot image that obtains its L2 boot image from flash using function *Flash_boot* exported by the flash file system library. An L1 flash booter is demonstrated in *$TCS/examples/flash_file_system/autoboot*. In this example directory, the flash-based L1 image can be built as follows:

   ```
   make -f Makefile.Unix l1.mi \
   CC=/t/lang/acc \
   ENDIAN=el \
   BOARD=Gomad \
   MMIO_BASE=0xefe00000 \
   SDRAM_BASE=0, SDRAM_LIMIT=0x800000 \
   TCS=/local/bin/tcs
   ```

   The location at which the flash-based L1 stage loads the L2 boot image is yet unspecified; this is to be defined for each L2 image individually.

   Function *Flash_boot* assumes that the datacache is disabled; this means that the cacheable limit address defined in l1rom.c of the autoboot sample directory should be set to the SDRAM base.

The Gomad driver has its bank size set to 6 megabytes to avoid a hardware problem in accessing the higher 2 megabytes of each flash bank. This might cause the L1 image size to exceed its size limit of 2 kilobytes; a work around for this problem is using a power of two as bank size.

4. Using the **-mi** option to the linker tmld, create the L2 boot image. In case this boot image itself needs to access the flash file system, then also pass the option **-u FlashFS** to tmld so that it fetches the Flash File System component from library *libio.a*.

```
tmcc -el my_app.c -tmld -mi -load 0x100000,0x800000 -u _FlashFS --
```

   CAVEAT: Avoid loading at **SDRAM_BASE**, since the L1 loader itself is executing there.

5. Optionally compress this L2 load image into a self-extracting image; refer to Chapter 11, *General Purpose Compression API*, of Book 5, *System Utilities*, Part A.
6. Put the standard L1 boot image into the boot EEPROM, as described in Chapter 7.
7. Create a file system on flash, and put the L2 boot image, and optionally other files, on to flash. The next section details the strategies for transferring this data to flash.
8. Replace the standard L1 book image with the flash-based L1 boot image. The system is now ready for booting.

Initializing Flash Contents

Flash memory has to be formatted into an empty file system before it can be used by the TriMedia Flash File System Manager. After that, files and directory structures can be arbitrarily created and deleted, and boot images can be written and overwritten without restriction.

In the examples provided in this chapter, it is assumed that the relevant BSP is linked by default. You can do this by changing the **BOARD_LIST_EL** and **BOARD_LIST_EB** lines in tmconfig. The compression tools use the BSP as the flash file system hardware interface by default.

The following tools in the TCS example directory facilitate the process of setting up initial flash contents, and of later maintenance:

$TCS/examples/flash_file_system/mkfs
: Format flash to an empty file system.

$TCS/examples/compression/zlib/utilities/tmWRB
: Embed a boot image into an executable that writes this boot image to flash memory. In case the flash contained an old boot image, this will be replaced.

$TCS/examples/compression/zlib/utilities/tmSEA
: Embed a directory structure into an executable that writes this directory structure to flash memory. Unrelated files that existed on flash are left untouched, while others are overwritten. Executing an application that has been created using tmSEA is similar to running an *untar* command on Unix.

These utilities can be used in succession to create the final flash image. There are two possible, subtly different ways of dealing with them:

1. The flash can be gradually filled by successively running the executables produced by the above utilities on the board on which the flash itself resides. The utilities can be downloaded via a JTAG connection until the flash is ready for standalone booting; after this, the boot EEPROM can be changed to boot from flash. This method programs one particular board, and is useful in a development setup.
2. It is also possible to create a flash image off-line, to be programmed later into one or more flash chips without intervention of the board that contains (or will contain) the programmed flash chips. This method is more useful in a production environment, in which a large number of flash-based systems have to be created.

Flash image pre-creation is possible by using the previous method on a reference system, after which the resulting flash image can be copied. A second solution does not even require a TriMedia processor: using tmsim and the flash simulator driver in *$TCS/examples/flash_file_system/sample_drivers/FlashSpecificSim.c*, the constructed simulated flash contents are maintained in a file FLASH image, and are directly available after running the 'flash' filling executables. When using the flash simulator with the purpose of building flash images, then it is advisable to disable the simulation of flash errors to avoid simulated bad sectors.

Simulating flash on tmsim, and successively running executables, thereby building up flash state, is demonstrated in *$TCS/examples/flash_file_system/all_together*.

Creating an Empty Flash File System Using mkfs

The following shows how to generate an executable that formats the flash to an empty file system:

```
cd $TCS/examples/flash_file_system/mkfs
make ENDIAN=el HOST=nohost BOARD= FLASH.SPECIFIC= mkfs.out
```

Executable *mkfs.out* can be downloaded and executed, and leaves an initialized flash. Progress is written to standard output.

Writing a Boot Image Onto Flash Using tmWRB

The following shows how to generate an executable that writes a boot image onto flash. It is performed by first generating the tool *tmWRB* for the chosen board, then generating the (L2) executable, and finally compressing the boot image derived from this into a boot image writer using *tmWRB*. Note that *tmWRB* implicitly converts a boot executable into a boot image:

```
cd $TCS/examples/compression/zlib/utilities/tmWRB
make -f Makefile.Solaris
tmcc -host nohost boot.c -o boot.out
tmWRB -el boot.out -load 0x100000,0x800000 -flashbsp -o write_boot.out
```


Executable *write_boot.out* can be downloaded and executed, and installs the image of executable boot.out as the flash's boot image, thereby overwriting the previous one when this existed.

Transferring Files to Flash Using tmSEA

The following shows how to embed a directory structure into an executable that writes this directory structure onto flash when it is run. First, generate the tool tmSEA for the chosen board. Then embed the chosen directory structure into a self-extracting archive. In this example, the TCS system dlls are transferred to directory */flash/dlls*:

```
## Input Commands
cd $TCS/examples/compression/zlib/utilities/tmSEA
make -f Makefile.Solaris
mkdir dlls
cp $TCS/lib/el/*.dll dlls
tmSEA -el dlls -od /flash/dlls -flashbsp -o write_dlls.out

## Output to stdio by tmSEA executable
D /flash
D /flash/dlls
F /flash/dlls/libPCI.dll
F /flash/dlls/libam.dll
F /flash/dlls/libc.dll
F /flash/dlls/libdma.dll
F /flash/dlls/libgeneric.dll
F /flash/dlls/libintpins.dll
F /flash/dlls/libm.dll
F /flash/dlls/libreg.dll
F /flash/dlls/libsem.dll
F /flash/dlls/libxio.dll
```

Executable *write_dlls.out* can be downloaded and executed, and writes the listed files to flash, thereby leaving all other flash files (and the boot image, when existent), unmodified.

Compressing TriMedia Boot Images

This chapter describes how boot images can be reduced in size, by first compressing them and then embedding the result in a new boot image. When this new boot image is started, it decompresses the original one, places it at a specified range in SDRAM, and transfers control to it. After this decompressing stage, the unpacker image is discarded, and the original image runs exactly as it would have run if it had been used for booting directly (i.e. without the compression/decompression stages). Note that this procedure reduces the required amount of space for *storing* the image; because it is decompressed before it is started, it does *not* reduce the required amount of SDRAM for *executing* it.

Using the full public domain compression library that has been provided in the $TCS/example directory, boot image size reductions of over 50% for large executables can be achieved.

Using tmSEI for Compressing Boot Images

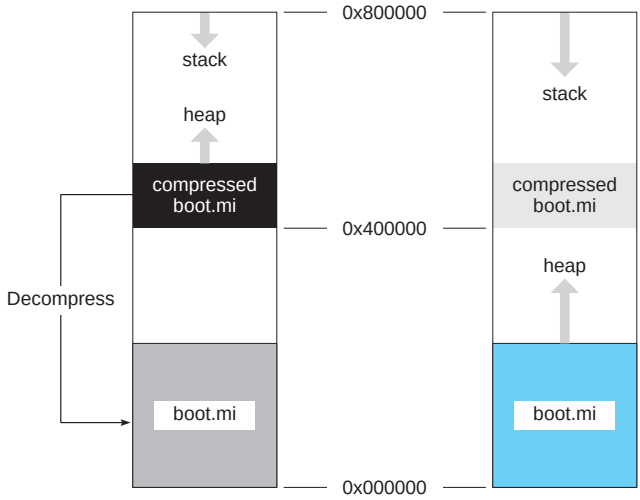

Figure 3 A self-extracting load image

The following shows how to compress an executable into a self-extracting image. It is performed by first generating the tool *tmSEI* for the chosen board, then generating the boot executable, and finally compressing the boot executable into a self extracting image using *tmSEI*. Note that *tmSEI* implicitly converts a boot executable into a boot image. Note also that *compressed_boot.mi*, which places the load image of *boot.mi* at the start of

SDRAM, is itself loaded in the second half of SDRAM to avoid interference with boot.mi during the decompression stage:

```
cd $TCS/examples/compression/zlib/utilities/tmSEI
make -f Makefile.Solaris
tmcc -host nohost boot.c -o boot.out
tmSEI -el boot.out \
      -load 0,0x800000 \
      -sei 0x400000,0x800000 \
      -flashbsp \
      -o compressed_boot.mi
```

Cascading tmSEI and tmWRB

Cascading of tmSEI and tmWRB can be achieved by letting tmSEI generate an executable object file instead of a memory image, and applying tmWRB on the result. Leaving the output of tmSEI as an executable can be achieved by omitting the -sei option:

```
tmcc -host nohost boot.c -o boot.out
tmSEI -el boot.out \
      -load 0,0x800000 \
      -flashbsp \
      -o compressed_boot.out
tmWRB -el compressed_boot.out \
      -load 0x400000,0x800000 \
      -flashbsp \
      -o write_compressed_boot.out
```

