

Booting TM-1000 in Stand-alone Mode

Topic	See Page
Introduction	2
Overview of Stand-alone boot	2
Sample Programs	5

Introduction

Bringing up TM1000 in stand-alone mode involves a number of steps. This note outlines the essential steps common to different stand-alone configurations. It also includes sample programs that may be modified to suit your needs.

This note assumes that you are familiar with the TM1000 architecture and that you have read Chapter 12 of the TM1000 Preliminary Data Book (April 1997): “System Boot”, which is the official document on both stand-alone and host-assisted boot procedure.

All the examples in this note refer to TCS software tools released in August 1997.

Overview of Stand-alone boot

During power-on reset, TM1000's boot block reads some configuration information from the EEPROM via I2C. The contents of the EEPROM determine, among other things, whether TM1000 continues to boot from the EEPROM or expects another processor (such as a PC or a Mac) to complete the TM1000's boot sequence. In a host-assisted boot, the EEPROM contains just 10 bytes that set a few parameters such as `TRI_CLKIN`, `PCI Sub-system Id`, `Vendor Id`, `MM_CONFIGS`, and `PLL_RATIOS`. The task of downloading an application to SDRAM and taking TM1000 out of reset is left to a host-based program (such as **tmmon** on the PC or Mac).

In a stand-alone boot, the EEPROM contains, in addition, the initial boot program whose size is restricted to 2K bytes. This initial boot program, called *L1 boot program*, is transferred by the TM 1000 boot block from EEPROM to SDRAM and then executed. It is the responsibility of the L1 boot program to load subsequent programs (we will call them L2 boot programs) from any attached device such as on-board UVEPROMs (or flash) or network and execute them.

Creating an EEPROM image

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

EEPROM Header

Contents of the EEPROM header are documented in Chapter 12 of the TM1000 Preliminary Data Book (April 1997 edition): “System Boot” and the memory system parameters are documented in Chapter 11 of the TM1000 Preliminary Data Book (April 1997 edition): “SDRAM Memory System”.

This note includes a sample program *l1rom.c* that creates an EEPROM image file (binary) of the L1 boot program. The *l1rom.c* program adds a 47-byte header to the given L1 boot

program and swaps the bytes of the L1 boot program when creating the EEPROM image. `l1rom.c` uses fixed values for `TRI_CLKIN`, `PLL` clock ratios, and so on. Stand-alone system developers need to examine and change the first 8 bytes of the EEPROM header in `l1rom.c`, if necessary, to suit their system. The EEPROM header bytes are documented in Chapters 11 and 12 of the TM1000 Preliminary Data Book (April 1997 edition).

L1 Boot Program

L1 boot code needs to do some initialization of TM1000 such as setting the PCSW, `BIU_CTRL`, setting up stack and frame pointers, initializing PCI devices (if any) and copying the L2 boot code to SDRAM. It then jumps to the beginning of L2 boot code.

The sample L1 boot program consists of 2 files:

- `l1start.trees`

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

- `l1main.c`

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

Note

If you are using TM1000 chips earlier than revision 1s1.1, I2C may be in some stuck state after autoboot. `l1main.c` contains a simple workaround.

On the TM1000 debug board, the UVEPROM is located at (PCI) address `0xFFC00000`. The sample L1 boot code loads the sample L2 code from (PCI) address `0xFFC00000` to (SDRAM) address `0x840` (the first cache aligned address after 2 K, since L1 code can be at most 2 K 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=150000 \
      -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 (a simple video-in video-out test program) fits within 150000 bytes. L1 boot code sets up SP and

FP starting at `MEMORY_SIZE` (defined to be 8 MB, since 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.

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

You cannot use the compiler driver `tmcc` to link the L1 boot code, since `tmcc` adds a number of options and libraries by default to the linker command line. This step is used just to verify that the sum of text, data, data1 and bss section sizes is less than 2K bytes. *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 `-mi` option to `tld` which will concatenate the text, data, data1 and bss sections and produce a memory image. You must also define `__clock_freq_init`, `__MMIO_base_init`, and `__begin_stack_init` as download parameters (`-bdownload __clock_freq_init` etc.) and then define their values (`-tm_freq 100000000` defines the TM1000 clock frequency as 100 MHz). If you use a TM1 IREF board with an 80 MHz TM1.1 chip, change this option to `-tm_freq 80000000`. Ensure that `l1start.o` is the first file in the list of files linked. This is because TM 1000 starts execution at address 0 (in the stand-alone case SDRAM BASE is 0) and you want the startup code `__start` to be located at that address.

```
tld -eb -o "l1.mi" -bdownload __clock_freq_init -mi \
    -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.

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 L1 boot program is always needed because of the way the boot block transfers bytes from EEPROM to SDRAM. The sample program `l1rom.c` has hard-coded values for the 47 byte header. You may want to modify `l1rom.c` and change the first 8 bytes to suit your system. The command `l1rom l1.mi` produces the EEPROM image file `l1.eeprom`, which is a binary file that may be used to program an EEPROM part such as ATML646 24c16, using an EEPROM programmer such as BP 1200. In TriMedia, we have access to another EEPROM programmer called SEEVAL, made by Microchip Technologies. This

programmer does not accept binary input files. Instead it uses its own proprietary input format. There is a sample program called seeval which takes l1.eeprom (produced at step 4) and adds the header and trailer required by the SEEVAL device programmer.

Note

The seeval program has worked for us but is neither supported nor guaranteed to work. It is better to use BP 1200 or another EEPROM programmer which accepts binary input files. ♦

Sample Programs

This note includes the following sample programs:

File	Description
Makefile	Works on SunOs (and may be on HP-UX) used to create l1 and l2 boot code, EEPROM images and so forth. You need to define TCS appropriately in the Makefile.
l1start.trees	These 2 files form the L1 boot code.
l1main.c	
vivot.c	This file forms the L2 code (plus standard device libraries).
seeval.c	If you are using seeval EEPROM programmer, you need this. If not, ignore this file.

Note

The video-in, video-out test program has been tested in little and big endian mode in a stand-alone system. ♦

Make File

The following Makefile automates the generation of sample L1 boot and L2 boot programs. Currently, the L2 program `vivot.c` shows how to use video in and video out device library.

```
#
# L2 program must be compiled to be loaded at address
# L2_LOAD_ADDR since L2_LOAD_ADDR is used in l1main.c
#

CP          = /bin/cp
MV          = /bin/mv
CC          = /t/lang/acc

TCS         = /t/qasoft/build/tcs1.1f0803SunOS
TMCC        = ${TCS}/bin/tmcc
TMLD        = ${TCS}/bin/tmld
TMSIZE      = ${TCS}/bin/tmsize

L1ROM       = l1rom
L2ROM       = l2rom
SEEVAL      = seeval

MEMORY_SIZE = 0x800000
TM_FREQ     = 100000000

# L1 boot program can be 2048 bytes 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
MMIO_BASE    = 0xEFE00000

ENDIAN       = -el
#ENDIAN      = -eb

L1_CFLAGS    = -v ${ENDIAN} -DL2_LOAD_ADDR=${L2_LOAD_ADDR} \
              -DL2_CODE_SIZE=${L2_CODE_SIZE}
-DL2_ROM_DEV_ADDR=${L2_ROM_DEV_ADDR} \
              -DMEMORY_SIZE=${MEMORY_SIZE}

L1_LDFLAGS   = ${ENDIAN}
```

Chapter 3: Booting TM-1000 in Stand-alone Mode

```
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 ${MMIO_BASE} \
                 -load=0,${MEMORY_SIZE}

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

L2_MIFLAGS      = -bdownload __clock_freq_init -mi -exec -start=__start \
                 -tm_freq ${TM_FREQ} -mmio_base ${MMIO_BASE} \
                 -load=${L2_LOAD_ADDR},${MEMORY_SIZE}

#
# -----

l1.out: llstart.trees llmain.c
    @echo ""
    @echo making $@
    ${CP} llstart.trees llstart.t
    ${TMCC} -x ${L1_CFLAGS} -c llstart.t llmain.c
    ${TMLD} ${L1_LDFLAGS} -o l1.out llstart.o llmain.o
    ${TMSIZE} l1.out

l1.mi: llstart.trees llmain.c
    @echo ""
    @echo making $@
    ${CP} llstart.trees llstart.t
    ${TMCC} -x ${L1_CFLAGS} -c llstart.t llmain.c
    ${TMLD} -o "$@" ${L1_MIFLAGS} llstart.o llmain.o
    @echo ""

l1.tst: ${SEEVAL} ${L1ROM} l1.mi
    @echo "Adding 47 bytes autoboot protocol header and swapping
bytes"
    ${L1ROM} l1.mi
    @echo ""
    @echo "Making SEEVAL input file"
    ${SEEVAL} l1.eeprom 2
    @echo ""

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

vivot.mi:       vivot.c
                $(TMCC) $(L2_CFLAGS) -o $@ -tmld ${L2_MIFLAGS} -- vivot.c
# -----
```

l1main.c

```

/*
 * +-----+
 * | Copyright (c) 1995,1996,1997 by Philips Semiconductors. |
 * | |
 * | This software is furnished under a license and may only be used |
 * | and copied in accordance with the terms and conditions of such a |
 * | license and with the inclusion of this copyright notice. This |
 * | software or any other copies of this software may not be provided |
 * | or otherwise made available to any other person. The ownership |
 * | and title of this software is not transferred. |
 * | |
 * | The information in this software is subject to change without |
 * | any prior notice and should not be construed as a commitment by |
 * | Philips Semiconductors. |
 * | |
 * | This code and information is provided "as is" without any |
 * | warranty of any kind, either expressed or implied, including but |
 * | not limited to the implied warranties of merchantability and/or |
 * | fitness for any particular purpose. |
 * +-----+
 *
 * Module name           : l1main.c
 *
 * Module type           : IMPLEMENTATION
 *
 * Title                 : L1 boot code
 *
 * Last update          : 15 July 1997
 *
 * Description           :
 *
 *                       L1 boot code.
 *                       Copies L2 boot code from a PCI-slave
UVEPROM
 */

```

```
#include <tml/mmio.h>

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

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

/* MACROS */

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

/* globals */

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

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

/* local variables */

volatile static unsigned int   dummy;
```

```

/*
 * copyback_dcache (unsigned addr, int nbytes)
 * 1. addr must be cache aligned.
 * This function flushes nbytes starting at addr to memory.
 *
 * L1 boot code copies L2 boot code from some device
 * This needs to be flushed to memory before jumping to the
 * L2 load address
 *
 */

static void
copyback_dcache(unsigned addr, int n)
{
    int i;

    for (i = 0; i < n; i = i + CACHE_BL_SIZE)
        dcb(0, addr + (unsigned) i);
}

/*
 * iclr is in a separate function to ensure that it is in a
 * dtree by itself
 */

static void
clear_icache(void)
{
    iclr();
}

/* Copies L2 code to SDRAM */

unsigned int Llmain ()
{
    int i;
    unsigned char byte;
    unsigned int *base_addr = (unsigned int *) L2_ROM_DEV_ADDR;
    unsigned char *load_addr = (unsigned char *) L2_LOAD_ADDR;

#ifdef 0

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

```

Chapter 3: Booting TM-1000 in Stand-alone Mode

```
/* 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 boot 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;
}
```

l1start.trees

```

(*
 * +-----+
 * | Copyright (c) 1995,1996,1997 by Philips Semiconductors. |
 * | |
 * | This software is furnished under a license and may only be used |
 * | and copied in accordance with the terms and conditions of such a |
 * | license and with the inclusion of this copyright notice. This |
 * | software or any other copies of this software may not be provided |
 * | or otherwise made available to any other person. The ownership |
 * | and title of this software is not transferred. |
 * | |
 * | The information in this software is subject to change without |
 * | any prior notice and should not be construed as a commitment by |
 * | Philips Semiconductors. |
 * | |
 * | This code and information is provided "as is" without any |
 * | warranty of any kind, either expressed or implied, including but |
 * | not limited to the implied warranties of merchantability and/or |
 * | fitness for any particular purpose. |
 * +-----+
 *)

(*
 * Module name           : l1.trees
 *
 * Title                 : L1 startup code
 *
 * Last update           : Tue Jul 15 09:53:10 PDT 1997
 *
 *)

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

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

#define __TMAS__
#include <tml/mmio.h>

```

Chapter 3: Booting TM-1000 in Stand-alone Mode

```
(*-----*)
#ifdef __BIG_ENDIAN__
#define INITIAL_PCSW_VALUE    0x0800    (* S *)
#define INITIAL_BIU_CTL_VALUE 0x0200    (* Host nable *)
#else
#define INITIAL_PCSW_VALUE    0x0A00    (* CS + Byte Sex *)
#define INITIAL_BIU_CTL_VALUE 0x0201    (* Host Enable + Byte Swap
Enable *)
#endif
(*-----*)

        .text
        .global __start
        .global _Llmain                (* defined in llmain.c *)

__start:
__start_DT_0:
entree(0)
.treeinfo regmask "0x0000000000000000ffffffff";

        (* iclr just to be sure *)

        10 iclr;

        20 uimm (INITIAL_PCSW_VALUE);
        21 uimm (-1);
        22 writepcsw 20 21;
        (* set up stack: FP and SP *)

        30 uimm (__begin_stack_init);
        33 wrreg (3) 30;
        34 wrreg (4) 30;

        (* set up return pointer *)

        40 uimm(__start_DT_1);
        41 wrreg (2) 40;

        (* configure BIU CTL *)

        50 uimm (BIU_CTL);
        51 uimm (__MMIO_base_init);
        52 iadd 50 51;
        53 uimm (INITIAL_BIU_CTL_VALUE);
        54 st32 52 53;

        gotree {_Llmain}

entree
```

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

___start_DT_1:
entree(0)
.treeinfo regmask "0x0000000000000000fffffffffffffff";

        12 rdreg (5); (* L2 Load Address *)
        cgoto 12
enttree
```

l1rom.c

```

/*
 * copyright (c) 1995,1996,1997 by Philips Semiconductors
 *
 * +-----+
 * | This software is furnished under a license and may only be used |
 * | and copied in accordance with the terms and conditions of such |
 * | a license and with the inclusion of the this copy right notice. |
 * | this software or any other copies of this software may not be |
 * | provided or otherwise made available to any other person. The |
 * | ownership and title of this software is not transferred.      |
 * +-----+
 *
 * Module name           : llrom.c
 *
 * Description           : 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.
 */

#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

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

void basename      (char *fname, char *bname);

```

```

int
read_file (unsigned char *buffer, char *filename)
{
    FILE          *Lfp;
    int           Lfd, n, nbytes;
    struct stat   file_stat;

    if ((Lfd = 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 (Lfd, &file_stat)) {
        fprintf (stderr, "Unable to fstat file: %s\n", filename);
        exit(1);
    }
    nbytes = (unsigned long) file_stat.st_size;
    close(Lfd);

    if ((Lfp = 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, Lfp);
    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);
}

```

Chapter 3: Booting TM-1000 in Stand-alone Mode

```
/*
 * 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_eeeprom_header (int nbytes, unsigned char obuffer[])
{
    int tmp;
    int i = 0;

    /* Output eeeprom 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++] = 0x01;    /* 2 */    /* make it 0x01 for IREF board.
different DENC */

    /* 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 Book
     * 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++] = 0x00; /* 23 */
obuffer[i++] = 0x00; /* 24 */
obuffer[i++] = 0x00; /* 25 */
obuffer[i++] = 0x00; /* 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++] = 0x00; /* 31 */
obuffer[i++] = 0x80; /* 32 */ /* assume 8 MB of sdram */
obuffer[i++] = 0x00; /* 33 */
obuffer[i++] = 0x00; /* 34 */

/* DRAM cacheable limit reg address, MSB first */
obuffer[i++] = 0xef; /* 35 */
obuffer[i++] = 0xf0; /* 36 */
obuffer[i++] = 0x00; /* 37 */
obuffer[i++] = 0x08; /* 38 */

```

```

/* DRAM cacheable limit reg value, MSB first */
obuffer[i++] = 0x00;    /* 39 */
obuffer[i++] = 0x80;    /* 40 */    /* assume 8 MB of sdram */
obuffer[i++] = 0x00;    /* 41 */
obuffer[i++] = 0x00;    /* 42 */

/* DRAM base reg value, MSB first */
obuffer[i++] = 0x00;    /* 43 */
obuffer[i++] = 0x00;    /* 44 */
obuffer[i++] = 0x00;    /* 45 */
obuffer[i++] = 0x00;    /* 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);
}

main (int argc, char **argv)
{
    int          i, j, file_size;
    int          header_bytes;
    FILE         *fp;
    char         *o_file_name, *cp;
    unsigned char ibuffer[BUF_SIZE] = {0};
    unsigned char obuffer[BUF_SIZE] = {0};

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

    /* find output file name */

    i = strlen(argv[1]);

    /* .eprom 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);
}

```

```
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';
    }
}
```

